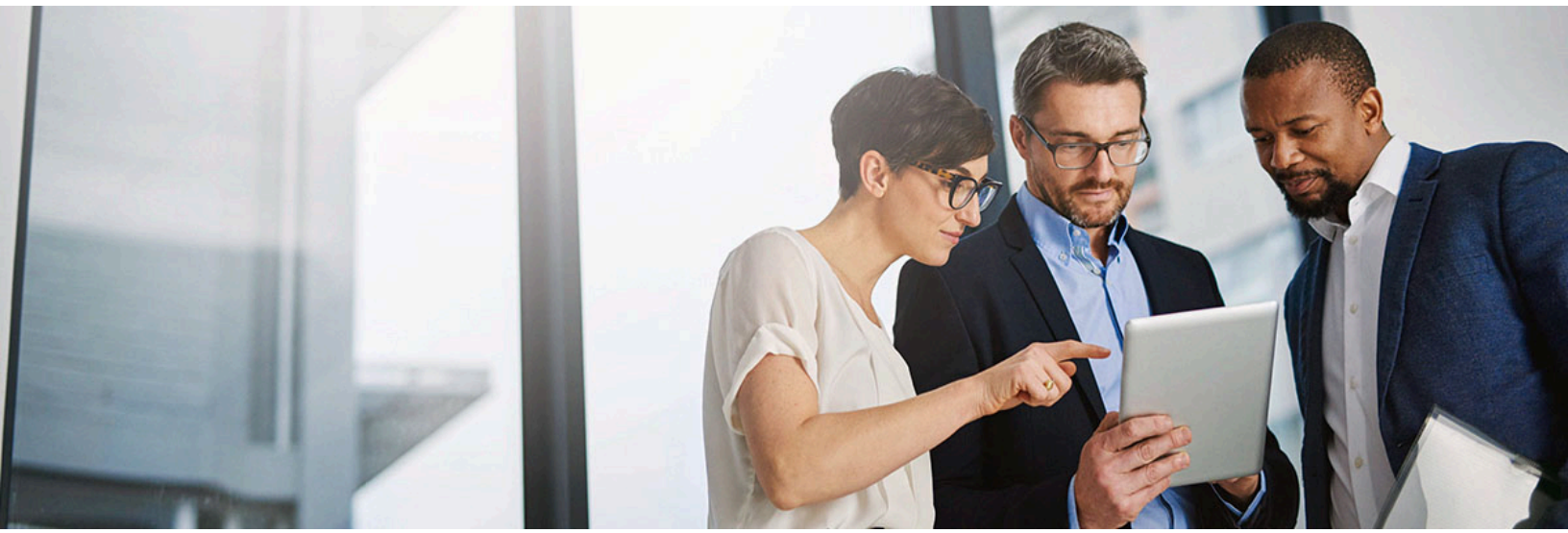


Atalasoft DotImage
Developer's Guide

Version: 11.4.0

Date: 2022-05-16

Atalasoft
A Kofax Company

© 2022 Kofax. All rights reserved.

Kofax is a trademark of Kofax, Inc., registered in the U.S. and/or other countries. All other trademarks are the property of their respective owners. No part of this publication may be reproduced, stored, or transmitted in any form without the prior written permission of Kofax.

Table of Contents

Preface **12**
- Related documentation 12
- Resources 12

Chapter 1: Atalasoft DotImage **13**
- Atalasoft DotImage Photo 13
  - Scenarios 14
  - Sample code 14
- Atalasoft DotImage Photo Pro 14
  - Scenarios 15
- Atalasoft DotImage Document Imaging 15
  - Scenarios 16
- Atalasoft DotImage add-ons 17
  - Barcode Reader 17
  - Barcode writing 18
  - OCR 18
  - Text translator 20
  - PDF Translator 20
  - PDF Reader 21
  - JPEG2000 Codec 22
  - DICOM Codec 23
  - Advanced Document Cleanup 23
  - Advanced Photo Effects 27
  - DWG decoder 28
  - Office Decoder 28
  - OfficeAdapterDecoder 29
  - Atalasoft DotImage ISIS 32

Chapter 2: Use Atalasoft DotImage **34**
- Atalasoft DotImage documentation 34
- Atalasoft DotImage NuGet Packages 34
  - Visual Studio Activation Wizard Extension 34
- Getting Started with Web Capture 35
  - Atalasoft DotImage demos 35
  - Set up a new project 35
  - Add the Web Document Viewer handler 37

Add the Web Capture handler....37
Set up the scanning controls and viewer....40
Wrap-up....41
Deploy on multiuser environment....41
Configure Kofax Import Connector....43
Upload sizes and limits....44
Document management....46
Capture and view documents with a scanning client....46
Submit captured documents to a central database repository....53
View documents from a central database repository in a browser....60
Collaborate on documents using AJAX-enabled annotations....63
Chapter 3: Program with Atalasoft DotImage....67
Getting started....67
Core Atalasoft DotImage editions....67
Add-on modules....68
.NET assemblies....69
Create images....72
Open images....73
Save images....74
Process and clean up images....75
Image controls for Windows Forms applications....76
Image controls for web applications....77
TWAIN scanning....79
Imaging....80
Dithering....80
Compression....81
Asynchronous image processing....82
Resampling....83
Morphology....85
Best practices....90
Memory management....90
Pixel memory....91
Image Source....98
Access images....114
Read and write images to a database....114
Access multipage images....115
Add support for RAW images....115
Get image information....116

View images....116
ASP.NET WebForm controls....116
Windows form control....179
Work with WPF images....185
Use ASP.NET WebForm controls....186
Image Capture....194
Web scanning....194
TWAIN scanning....195
ISIS scanning....202
Image processing and cleanup....206
Manipulate colors with Lookup Tables....206
Resize images....207
Process an image using a Workspace object....209
Process an image using the Apply method....210
Process a Bitmap image....210
Extend ImageCommands....210
Upload an image to a server....216
Annotations....218
Annotations....218
Import and export annotations....219
Create an annotation enabled control....226
Annotation assemblies....227
Create a template annotation....228
Print annotations....228
Serialize to XMP And WANG data....229
Work with unit systems....230
Render custom grips....231
Respond to events....233
Add a context menu to an annotation....234
Highlight a document....234
Work with layers and groups....235
Create a custom annotation....235
Password-based authentication....239
Create a sticky note....240
Work with unit systems....241
Annotate multipage documents....241
Metadata....242
Supported metadata types....242

Image formats supporting metadata....243
IPTC metadata....243
EXIF metadata....245
COM text....246
TIFF tags....246
XMP....249
Save metadata with an image....249
Retrieve metadata from an image....250
Set metadata values....250
Attach metadata to objects....252
Retrieve a document title....253
Read EXIF information....253
Read EXIF thumbnails....253
Store EXIF information....254
Add an object to document metadata....254
Obtain DPI information from a .PSD File....255
Losslessly copy metadata....255
Document and image formats....256
Introduction to PDF technology....256
.TIF files....272
DICOM....284
JPEG2000 - encoding images....287
Introduction to JBIG2....291
Custom codecs - image codec....292
Barcode reading....300
Upgrade tips....301
Use the BarcodeReader....301
Read a bar code with options set....302
Render a bar code into an AtalaImage....302
Verify a bar code can represent a string....302
Barcode writing....303
Deployment....303
Use the Barcode writer....303
OCR document design considerations....304
Use the factory property to construct a document....305
Load OCR resources....305
Stages in OCR translation....308
Translator types....310

Page region types....311
OCR engine....312
Glyph reader engine....314
Tesseract engine....316
OmniPage Engine....318
Use an OCR engine....323
Get and set engine options....323
Determine if an engine supports a mime type....324
Alter the interpretation of page elements....324
Determine translation type....325
Distinguish between OCR region types....325
Clean up after translation....326
Traverse an OCR document....326
Cancel OCR in progress....327
Track page progress in a UI....327
Use page deskew events....328
Deploy an OCR engine....329
Access document information properties....331
Color management....332
Color profiles....332
CMYK images....332
Draw on the canvas....333
The canvas....333
Atalasoft DotImage drawing versus GDI+....333
Draw text....334
Draw shapes....334
Draw with rubber bands....335
Interoperability - Work with GDI+ images....335
Create a copy of the data....336
Pass the data directly to the AtalaImage constructor....336
Interoperability - Work with WPF images....336
Forms processing....338
Align an image to a template....338
Disable alignment rejection heuristics....338
Use the OMR engine to recognize marks on a page....339
Create and save an OMR template....339
Perform additional preprocessing in the OMR engine....340
Cancel OMR in progress....340

Track OMR engine page progress in a UI....341
Web Document Viewer....342
Chapter 4: Deploy Atalasoft DotImage....343
Visual C++ Runtime dependencies....343
Deploy Atalasoft DotImage in ASP.NET....343
Dependencies using Atalasoft DotImage class library....343
Dependencies using Atalasoft DotImage with WebControls....343
Generating licenses....344
Chapter 5: Program with DotPdf....346
Mathematical model....347
Transformations....348
PdfGeneratedDocument....350
Pages....350
Standard page sizes....351
Create stationery....351
Clipping....353
Colors....354
Rendering....355
Resources....355
Font resources....356
Type 1 symbol font encoding....357
Embed fonts....360
Color space resources....360
Image resources....361
Template resources....362
Shapes....362
PdfPath....362
PdfRectangle....365
PdfRoundedRectangle....366
PdfCircle....366
PdfArc....366
PdfImageShape....366
PDF text shapes....368
PdfTable....369
PdfTemplateShape....370
PostnetBarcodeShape....373
GSave / GRestore....373
Transform....374

Marked content....374
Make custom shapes....375
Round trip documents....378
Integrate with DotImage....379
Actions....380
PdfAction....381
Go To View actions....381
URI actions....382
JavaScript actions....382
Sound actions....382
Show/Hide action....383
Named actions....384
Submit Form Actions....385
Reset Form Action....385
Annotations....385
Properties common to all annotations....386
Properties common to all mark up annotations....389
Properties common to all widget annotations....390
General annotations....391
Markup annotations....394
Widget annotations....405
Use annotations....414
Place an annotation....414
Create an annotation with a custom border....415
Add a pop-up to a markup annotation....416
Create an annotation with transparency....417
Skin an annotation....418
Make an annotation with a rollover appearance....419
Make a sticky note annotation....420
Add a review state to a sticky note....420
Make a highlight annotation....421
Set a redaction area....425
Use JavaScript to calculate values....426
PDF Forms....428
PdfForm....429
Node form fields....430
Leaf form fields....430
Visiting nodes....430

Form actions....431
Merge PDF forms....432
Import pages....433
Merge forms....433
Default merging....434
DotPdf repair....434
DotPdf repair process....434
Detect errors....435
Repair errors....436
Repair events....436
Repair filtering....437
Structure options....438
Array options....440
Property repair....440
Digital signatures....441
Certify documents....442
Get signer information....444
Document signing operations....447
Customize signature appearance....450
Certify a document with PdfDocument....451
Determine if a document is certified or signed....451
Fill fields of a certified document....452
Sign a document with an existing signature....452
Add a signature to a document....452
Linearized PDF....453
PdfDocument and PdfGeneratedDocument integraton....453
PdfEncoder integration....454
PDF/A....454
PDF/A in PdfDocument....454
PDF/A data in PdfDocumentMetadata....457
PDF/A in PdfGeneratedDocument....457
PDF 2.0....465
Document upgrade to PDF 2.0....466
Chapter 6: DotTwain....467
About DotTwain....467
Acquire images....468
Document feeder control....468
Navigating files in a camera....469

ActiveX control API reference....469
Class events....470
Getting started....470
Add DotTwain to the toolbox....470
Set application information....471
Model acquisition....471
Set up events....471
Show the Select Source dialog....472
Properties....472
Acquire an image....472
Acquire images with TWAIN....473
Acquire a select region of the device....473
Acquire and save images directly to a file....474
Detect a camera device....475
Upload an image to a server....476
Deploy DotTwain....477
Chapter 7: OCR engine....478
OCR document design considerations....479
Use the factory property to construct a document....480
Tesseract engine....480
Features....480
Supported languages....481
Supported output formatters....481
Deployment....481

Preface

The *Atalasoft DotImage Developer's Guide* contains information about how to install and customize your Atalasoft DotImage installation. This guide explains how to:

- Use .NET assemblies to acquire, read, write, display, annotate, or process images
- Use WebForms controls to scan, display, and manipulate images and documents
- Add .NET controls to WinForms, WPF, and WebForms projects

Related documentation

In addition to this guide, the Atalasoft DotImage documentation set includes the following:

- API Reference: Gives the complete Atalasoft DotImage class library in online help format.
- API Reference (.chm file): Gives the complete Atalasoft DotImage class library for offline use.
- *Atalasoft DotImage Release Notes:* Contains late-breaking product information not included in this guide. You can download the release notes from the Atalasoft Web site at www.atalasoft.com.
- Atalasoft DotImage Demos

Resources

The following resources are available for Atalasoft products:

- Purchase Atalasoft DotImage
- Knowledgebase
- Atalasoft DotImage Feature Matrix
- Atalasoft DotImage Support
- Sample applications
- Atalasoft DotImage Dev Team Blog
- Contact information.

Chapter 1

Atalasoft DotImage

Atalasoft DotImage is a suite of .NET assemblies to use in projects to acquire, read, write, display, annotate or process images. Atalasoft DotImage assemblies can be used in desktop and ASP.NET projects and the various objects and methods are accessible from any .NET-compliant language including C# and Managed C++.

Atalasoft DotImage WebForms controls include innovative AJAX controls for scanning, displaying and manipulating images and documents, all with a minimal or no client-side footprint.

Images used in Atalasoft DotImage controls or objects can come from files or databases, as well as from scanners or cameras.

In addition to standard objects, Atalasoft DotImage includes .NET controls you can add to WinForms, WPF, and WebForms projects. These controls make it easy to create applications that display or edit images.

Atalasoft DotImage Photo

Atalasoft DotImage Photo is a raster imaging toolkit that can add advanced image viewing, editing, and printing to an application. Atalasoft DotImage compliments GDI+ in .NET by adding more advanced codecs, image processing, and UI controls. Atalasoft DotImage Photo includes an imaging class library, and a Windows Forms control library. The Atalasoft DotImage class library is used for all image processing, encoding, and decoding, in Windows Forms or ASP.NET WebForms applications. It is broken up into logical namespaces such as Imaging, Codecs, Drawing, and ImageProcessing. The Windows Forms library includes visual UI controls for viewing, printing, and rubberbanding images.

The toolkit includes the following features:

- Read write, and convert to popular image formats such as JPEG, PNG, BMP, DIB, TIFF, GIF, PCX, TGA, PSD, WBMP, EMF, WMF, TLA, PCD, PCD, with a plug-in interface to add new ones.
- Compress images using algorithms such as JPEG, LZW, Deflate/ZIP, RLE, Packbits, CCIT Group 3/4, and Huffman.
- Over 100 image processing commands to apply filters, effects, and transforms to images.
- Over 20 different resampling commands for fast or high quality enlargements or thumbnail creation.
- Advanced image viewing, scrolling, zooming, and rubberbanding with the BitmapViewer, ImageViewer, and WorkspaceViewer controls.
- Control over printing single or multiple images with the ImagePrintDocument component including properties to center, fit to margins, and fit to edges.

- Support for streaming images to any .NET Stream or byte arrays to support saving and retrieving images from SQL, Access, Oracle, and other databases using binary streams.
- Load or save images from HTTP or FTP. Includes an HTTP Post object for posting images directly to a server.
- A Workspace object for easily handling and processing images with no need to worry about memory cleanup.
- Full alpha transparency support.

Scenarios

Atalasoft DotImage Photo is used in a variety of imaging applications and industries. some sample scenarios include:

- Digital imaging applications
- Stock photo Web sites
- Employee photo ID application
- Advanced photo resizing

Sample code

This example demonstrates how to open a TIFF image, apply a marble transform, save it as a JPEG with a quality of 90, and stream to the browser for display.

C#

```
myWorkspace.Open("myimage.tif");
myWorkspace.ApplyCommand(
new MarbleTransform(1.4, new Size(5, 5)));
Response.ContentType = "image/jpeg";
myWorkspace.Save(Response.OutputStream,
new JpegEncoder(90));
```

Atalasoft DotImage Photo Pro

Atalasoft DotImage Photo Pro is an advanced raster imaging toolkit that can add image viewing, editing, printing, and metadata support to an application. Atalasoft DotImagee compliments GDI+ in .NET by adding more advanced codecs, image processing, and UI controls. Atalasoft DotImage Photo Pro includes all the features of Atalasoft DotImage Photo as well as high-end photographic/prepress support, and an AJAX enabled ASP.NET Server- Side Image Viewer. Full documentation is integrated into Microsoft Visual Studio .NET. See Atalasoft DotImage Document Imaging for document imaging functionality and support. For the high-end photography and pre-press market, the Atalasoft DotImage Advanced Photo Effects Module is an add-on that can be used to automatically enhance digital photos without laborious manual touch-ups.

In addition to the features included in Atalasoft DotImage Photo , Photo Pro includes the following:

- Support for 32-bit CMYK images.
- Support for 16-bit grayscale, 48-bit RGB, and 64-bit RGBA color depths.

- An ASP.NET AJAX-enabled server-side image viewer to view, pan, zoom, process, and rubberband images without the need for client applets or plug-ins. New AJAX controls for viewing thumbnails and editing images on the with full scrolling support and on-demand tiled loading.
- Support for reading over 100 RAW digital camera formats including support for decoding the DNG RAW image format.
- A multi-threaded Thumbnail List Viewer control to display thumbnails from custom sources, or list thumbnail images from a directory with horizontal or vertical scrolling support, and ability to customize thumbnail size, position, and style.
- A composite printing component that will print multiple images as a photo composite or contact sheet.
- Support for reading metadata such as EXIF, IPTC, XMP, TIFF Tags, COM Markers, Adobe Photoshop Resources, and Eastman WANG annotations.
- Support for writing metadata such as EXIF, IPTC, XMP, TIFF Tags, COM Markers, Adobe Photoshop Resources, and Eastman WANG annotations as well as creating EXIF Thumbnails in TIFF images.
- Full color management support including the ability to extract and embed ICC color profiles, automatically adjust colors based on target and destination color profiles and create virtual proofs.
- AutoLevels, AutoColor, Curves, and Levels commands that automatically adjust colors, just like the Adobe Photoshop equivalent functions.
- Red Eye Removal.
- Dust and Scratch Removal.
- Region of Interest Processing.

Scenarios

Atalasoft DotImage Photo Pro is used in a variety of imaging applications and industries. In addition to the features included in Atalasoft DotImage Photo, Photo Pro includes the following scenarios.

- Pre-press printing application.
- Online image viewing application.

Atalasoft DotImage Document Imaging

Atalasoft DotImage Document Imaging is an advanced document imaging toolkit that can add high performance image viewing, editing, printing, scanning, and annotating support to an application. The toolkit includes an imaging class library, a Windows Forms control library, an ASP.NET AJAX-Enabled Server-Side Image Viewer, DotTwain for advanced TWAIN scanning, and advanced annotations support for both Windows Forms and ASP.NET WebForms.

In addition to all features in Atalasoft DotImage Photo Pro and the AJAX-enabled thin-client image viewer, the toolkit includes the following features:

- Multipage TIFF encode and decode with support for many obscure flavors of TIFF (including most Old Style JPEG-In-TIFF images).
- Multipage PDF Encoding Support (embedded images only) l TWAIN Scanning support with the included DotTwain SDK* l Advanced Windows Forms Annotations support webForms Annotations support featuring AJAX technology*.

- TIFF Directory manipulation controls for inserting, removing, and reordering TIFF pages in a multipage TIFF without re-encoding the entire image.
- Advanced TIFF Tag editing support to edit standard and arbitrary TIFF Tags.
- Optimized Scale to Gray display, fastest in the industry.
- Optimized Scale to Gray display when viewing 1-bit documents using the AJAX- enabled thin client image viewer.
- Document Processing Functions including:
  - Fast Auto-Deskew for binary images.
  - Despeckle binary images.
  - Remove noise from binary images.
  - Auto border detection and removal
  - Adaptive and Global Thresholding to convert grayscale or color images to binary.
  - Binary Morphological Filters (Erode, Dilate, Boundary Extraction, Thinning).
- Render FAX images with different X and Y resolutions properly (both WinForm and WebForm controls).
- Thresholding Commands optimized for multi-core processors.
- The Advanced Document Cleanup (ADC) add-on module providing the ability to auto- matically remove specks, borders, lines, blobs, hole punches, and halftones for improved image quality using unique proprietary algorithms.
- The ADC add-on also provides blank page detection, as well as the ability to auto-negate inverted text and inverted documents.
- High quality character recognition with the GlyphReader OCR add-on module, a highly accurate and cost-effective OCR engine. Provides a generic interface for OCR with integration with other industry leading OCR engines.
- A Searchable PDF add-on module for OCR that generates high-quality searchable PDF documents from any Atalasoft DotImage-supported OCR engine. Produces documents with hidden text behind image for color, gray, or bitonal images. Supports JBIG2 and JPEG2000 compression.
- View PDF documents with the PDF Reader add-on module.
- Recognize 1D and 2D barcodes with the BarcodeReader add-on module.

*Requires additional licensing for production server distribution.

Scenarios

Atalasoft DotImage Document Imaging is used in a variety of imaging applications and industries. Some sample scenarios include:

- Medical records
- Insurance documents Management Solution
- Mailroom document application
- Check management in financial/banking industries
- Online Mechanical Drawing Viewer

See our online AJAX Image Viewer for an example on how to use Atalasoft DotImage on the to view and cleanup multipage documents.

Atalasoft DotImage add-ons

You can enhance the capabilities of Atalasoft DotImage by using add-ons such as Barcode Reader, Barcode Writer, OCR, PDF Reader, JPEG2000 Codec, JBIG2 Codec, DICOM Codec, DWG Decoder, and Atalasoft DotImage ISIS.

Barcode Reader

The Atalasoft DotImage BarcodeReader Add-On provides advanced bar codeimage recognition for your .NET applications. This component is very easy to use and designed specifically for Microsoft .NET.

ℹ Licensing is runtime royalty free for desktop applications.

Features

- Recognizes all barcodes in an image
- Returns the string value of each bar coderecognized
- Reads twenty-one industry 1D symbologies as well as QR Code, PDF417 and DataMatrix 2D symbologies
- Automatically detects orientation of bar code(East, South, West, North)
- Returns the bounding rectangle of all recognized barcodes
- Returns the coordinates of the start and end lines, can be used to construct a polygon encompassing the bar codearea
- Detects the type of bar coderecognized
- Integrates with Atalasoft DotImage with the ability to include an image viewer and pre-processing capabilities such as deskew, despeckle, and annotations. Returns position of checksum character (if present)
- Returns any supplemental barcodes
- Deploys as a single managed assembly alongside Atalasoft DotImage dependencies

Supported symbologies

1D Barcodes			
Codabar	Code93	Patch code	RM4SCC (Royal Mail)
code 11	EAN-13	Planet	Telepen
code 128	EAN-8	Plus 2	UPC-A
code 32	Interleaved 2 of 5	Plus 5	UPCE-E
code 39	ITF-14	Postnet	

2D Barcodes
Aztec
DataMatrix

2D Barcodes
PDF417
QR Code

Deployment

When using the BarcodeReader, the assemblies that need to be copied with your application include:

Assembly	Description
`Atalasoft.Shared.dll`	Shared classes such as licensing management
`Atalasoft.dotImage.Lib.dll`	DotImage low level library assembly
`Atalasoft.dotImage.dll`	DotImageAtalasoft DotImage class library assembly
`Atalasoft.dotImage.Barcoding.Reading.dll`	Barcode Recognition Engine

Barcode writing

The DotImage Barcode Writing assembly is designed to be a simple set of classes that make it easy to create barcodes within a .NET application. There are objects that are designed for writing directly into Graphics objects and corresponding Win Forms Controls that allow barcodes to appear in window-based applications.

The Barcode Writing classes can be used for many common 1-D bar codetypes as well as PDF417 and DataMatrix barcodes.

Deployment

When using Barcode Writing, the assemblies that need to be copied with your application include:

Assembly	Description
`Atalasoft.Shared.dll`	Shared classes such as licensing management
`Atalasoft.dotImage.Barcoding.Reading.dll`	Barcode Writing Engine

OCR

Atalasoft DotImage OCR is an adapter module for Microsoft .NET developers that allows programmers to add character recognition (OCR) to their applications.

The Atalasoft.Ocr namespace contains a set of classes for managing and processing documents to be processed for optical character recognition (OCR). These classes provide the functionality necessary for OCR, without being tied directly to any particular OCR engine. This allows client code to be insulated from changes in the underlying engine, and makes it easy to test and evaluate different OCR engines.

OCR engines are supported through engine-specific add-ons that fit into the Atalasoft DotImage OCR framework.

Engine Add-Ons

You can add these corresponding OCR engines:

- GlyphReader Engine Add-on
- Tesseract Engine Add-on
- OmniPage Engine Add-on

Features

- Fully extensible file and stream export
- OCR Engine neutral, open API
- Built-in image preprocessing
- Fully overridable image preprocessing
- Easy event model for tracking progress and reporting/modifying document layout
- Fully extensible document and page model
- Font management and abstraction
- Confidence level provided at region, line, word, and glyph levels
- OCR capability for any image that can be read by Atalasoft DotImage
- Easy integration with DotTwain
- Images can come from any source, not just files
- Output formats specified by MIME standard
- Built-in Text Translator for formatted text output
- Searchable PDF module for outputting results in highly compressed JBIG2 Adobe PDF as Text Only, or Hidden Text Underneath Image
- Supports engines that automatically identify regions (or zones) of an image, or manually zone images yourself

Licensing

To use OCR functionality, you must have an Atalasoft DotImage license, licenses for any OCR Engine Add-On(s) you use, and a license for any OCR engines you use.

ℹ A license to use an Atalasoft DotImage OCR Add-On is not a license to use the corresponding 3rd-party OCR engine.

There is also an add-on module to generate searchable PDF documents from OCR.

Output Formats

Atalasoft DotImage OCR provides a flexible means for translating OCR output to alternative document formats. Client code requests a mime type that represents the desired output format and specifies whether the output should go to a file or stream. If the engine can provide the requested output format, it goes ahead with the work. The output translation mechanism is fully extensible and changeable by client code.

Broadly speaking, there are two different types of translators: native and foreign. Native translators are built into a particular OCR engine, foreign translators are those that are supplied from outside of an engine. Typically, native translation avoids the overhead of constructing an OcrDocument and

foreign translation always requires construction of an OcrDocument. Foreign translators are like to be flexible in their configuration and output styles but can be less efficient when used with some engines because they operate at a significantly higher level than native translators.

This distinction allows us to publish low-level engine translators and to provide a means of adding other translators that are treated as first class objects.

Text translator

A TextTranslator class implements a foreign translator that is used to generate text files from an OCR engine. The TextTranslator has properties to control how much the TextTranslator attempts to mimic the layout and format of the original document.

PDF Translator

The PdfTranslator class allows client applications to generate high quality PDF documents from scanned documents.

Features

- Ability to set PDF Metadata fields
- High quality thumbnail images
- Accurate text placement
- Text-Under-Image placement
- Optional placement of picture regions
- Automatic or client-controlled image compression
- Advanced codec support (JBIG2, JPEG 2000)
- Insertion of client synthesized pages
- Generation of PDF/A-1b and PDF/A-2b compliant documents with embedded fonts and color profiles

Font Building

Atalasoft DotImage OCR defines an IFontBuilder interface which is used to construct fonts in documents. Since the elements within a document page can come from many different sources, it is advantageous to have a central resource location for fonts. The FontBuilder allows fonts to be retrieved for every text element without the need to construct any more font objects than are needed for the page. If a document needs to enumerate all the font resources on a page, it is possible to do so by scanning every font in the document and then enumerating those objects within the FontBuilder.

Atalasoft DotImage OCR provides a simple implementation of the IFontBuilder interface called BasicFontBuilder. This object caches font building requests and only returns one instance of a font for every unique font requested.

Font Mapping

When recognizing a document, it is necessary to create fonts associated with text

elements. There is, however, no guarantee that an identified font in a given document is available on the computer that is doing the recognition. Further, some engines might not be able to do any better than "this font looks like it has serifs" or "this font looks like it is monospaced." For those reasons, every engine build contains a FontMapper object which can be used to map one set of

font characteristics to another. This makes it possible to turn a generic "sans serif" font into, for example, Gill Sans or Helvetica, rather than Arial. It also makes it possible to strip out all "strike out" font characteristics.

When constructing an OcrDocument, the engine's FontMapper determines which font should be used for a page element. You may supply your own FontMapper to replace the standard font mapper.

Atalasoft DotImage provides a simple implementation of the IFontMapper interface called BasicFontMapper. This object attempts to map a requesting font to a matching system font, using Arial as the default when no font matches are present.

Metadata

The OcrPage object includes a property called Metadata which is reserved for holding metadata associated with a given page. At present, only the PdfTranslator uses the Metadata property, but the same conventions applied to the OcrDocument metadata should be honored: client code should only use add values using non-integral keys, such as strings.

PDF Reader

Atalasoft DotImage PDF Reader (formerly PDF Rasterizer) quickly decodes PDF documents into color or grayscale raster images. It seamlessly integrates into Atalasoft DotImage and associated image viewing components as an image decoder.

PDF Reader supports simple image-only PDF's as well as complicated PDF's with vectors, text, patterns, and transparency. You can:

- Rasterize any PDF to an image
- Extract images from a PDF document
- Extract text and text locations from a PDF document
- Search for text in a PDF document
- Render a PDF directly to a printer

Atalasoft DotImage PDF Reader provides a PdfDecoder which seamlessly integrates into Atalasoft DotImage and associated image viewing components.

To use PDF Reader to rasterize images:

1. Add an instance of the decoder to the Decoders collection.
2. Open the PDF with Workspace.Open(), new AtalaImage(), or any Atalasoft DotImage method that takes a stream containing an image.

For further details, see How to: View a PDF Image.

Features

- Add high speed viewing of PDF documents to your applications
- Convert PDF's to raster images in the Atalasoft DotImage Image Viewers without the Adobe Acrobat SDK
- Extracts and searches for text in a PDF document
- Print PDF Documents using the Atalasoft DotImage Printing components or by rendering onto a printer graphics object.

- Convert PDF Documents into any supported Atalasoft DotImage image format (multipage TIFF, JPEG, etc).
- Integrates with our AJAX enabled Web Image viewer for server-side viewing, panning, and zooming of PDF documents
- Extract all images from a specified PDF page to their native bit depth
- Specify a resolution to use when rasterizing the PDF for instant PDF thumbnails
- Supports CCIT G3, G4, Flate, and JPEG embedded images
- Support for JBIG2 and JPEG2000 compressed images
- Supports PDF Specification versions 1.3, 1.4, 1.5, and 1.6
- Supports all font types: Type1, TrueType, Type0, Type3
- Supports all text rendering modes (full, stroke, clipping)
- Supports tiling patterns and shadings
- Supports all colorspaces: RGB, Gray, CMYK, ICCBased, Lab, Indexed, and Separation
- Runtime Royalty Free Desktop Licensing

Deployment

When using PDFReader, the assemblies you need to copy with your application include these:

Assembly	Description
`Atalasoft.Shared.dll`	Shared classes such as licensing management
`Atalasoft.dotImage.Lib.dll`	Low level Atalasoft DotImage Library
`Atalasoft.dotImage.dll`	Atalasoft DotImage class library
`Atalasoft.dotImage.PdfReader.dll`	PDF Decoder

When using other Atalasoft DotImage functionality, such as WinControls, those assemblies must be distributed as well.

JPEG2000 Codec

Atalasoft DotImage JPEG2000 codec can be used to decode and encode JPEG2000 images using the Microsoft .NET Framework. It uses wavelet compression technology to compress photographic images further then any other available compression scheme. The codec is available as a plug-in and integrates with Atalasoft DotImage seamlessly. Atalasoft DotImage JPEG2000 is based off of Luratech's Lurawave.jp2 wavelet compression technology.

ℹ Licensing is runtime royalty free for desktop applications.

Features

Standard Edition

- Ability to decompress JPEG2000 images stored in any compatible jp2, j2k, or jpc code stream
- Supports decoding directly to 8-bit grayscale, 24-bit RGB, 16-bit grayscale, and 48-bit RGB
- Supports encoding 8-bit grayscale and 24-bit RGB to jp2 code stream
- Specify compression ratio (0 - 100)
- Lossless compression

- Integrated with Atalasoft DotImage's PdfEncoder to encode PDF images with JPEG2000 Compression.
- Integrated withAtalasoft DotImage's searchable PDF module for OCR to encode PDF images with JPEG2000 Compression
- Runtime Royalty Free Desktop Licensing

Professional Edition

- Supports decoding to CMYK colorspace
- Supports encoding 16-bit grayscale and CMYK images
- Full Metadada support, such as IPTC, XMP, XML Box, and UUID Boxes
- Precise control over the decoder and encoder settings
- Progressive Decoding Support

DICOM Codec

Atalasoft DotImage DICOM codec can be used to decode DICOM images and metadata using the Microsoft .NET Framework. DICOM stands for Digital Imaging and Communications in Medicine and is typically used to hold medical images and information about a patient. Our DICOM Decoder has the ability to decode images compressed with JPEG2000, JPEG, JPEG Lossless, and RLE.

ℹ Licensing is runtime royalty free for desktop applications.

The Atalasoft DotImage DICOM Codec is an add-on module to Atalasoft DotImage for viewing medical images and requires a Atalasoft DotImage license. Add DICOM viewing to your applications when integrating with our AJAX-Enabled Image Viewing technology for ASP.NET.

Features

- Views both old and new style DICOM Formats
- Decodes images into 8-bit Gray, 16-bit gray, 24-bit RGB, or 48-bit RGB colorspaces
- Supports 10-bit, 12-bit, and 14-bit gray promoting them to 16-bit gray for viewing
- Read all patient metadata from the image
- Supports advanced JPEG2000 compression
- Reads a specified frame in a multi-framed DICOM image
- Full Annotations support
- Integrates with the Atalasoft DotImage image processing and windows based or based component features
- Supports access to raw images for efficiently applying repeated window and leveling operations

Advanced Document Cleanup

The Advanced Document Cleanup (ADC) add-on provides document cleanup algorithms you can apply applied to scanned documents to clean them up for better compression and archival, increased readability, and for improved OCR accuracy.

Atalasoft DotImage ADC uses proprietary algorithms developed by our research and development team that are designed to automatically select the best parameters to apply for fast and accurate processing. The commands included in ADC extend the command interfaces in Atalasoft DotImage and can easily be applied to an existing application built with Atalasoft DotImage.

ADC features

Atalasoft DotImage Advanced Document Cleanup (ADC) provides the features listed below. These commands can all be applied to an entire image or to a region of interest.

Binarize

Binarize uses your choice of three thresholding methods (adaptive, global, or simple) to efficiently and accurately convert a color or grayscale image to binary.

Binarize often results in better bi-tonal documents than those obtained when scanning directly to black and white.

Advanced Border Removal

Advanced Border Removal removes black borders from a bi-tonal image using a sophisticated object detection algorithm. This is useful because scanning documents often results in an unnecessary border around the image.

CONTENTS

ILLUSTRATIONS

CONTENTS

ILLUSTRATIONS

Margin Crop

Removes margins (white space) from each side of an image. Useful for decreasing image size, and isolating relevant features on an image for processing.

Auto Border Crop

Auto Border Crop is similar to Advanced Border Removal, but crops the border as opposed to removing the feature. It uses a higher speed algorithm than Advanced Border Removal.

Speck Removal

Removes specks of a specified size from a bi-tonal image. Useful for removing noise from an image introduced by a scanner or when thresholding from a color or grayscale image.

Hole Punch Removal

Detects and removes hole punch artifacts from a bi-tonal image. Will automatically remove round hole punches from bi-tonal images from any of the 4 sides that appear when these documents are scanned.

Blob Removal

Removes arbitrary blobs of a specified size from a bi-tonal image.

Blank Page Detection

Automatically detects if a bi-tonal document is blank using a sophisticated object detection algorithm. Useful when scanning duplex (both sides) when the back side of the document is blank.

Automatic Page Inversion

Automatically detects and optionally inverts an image when the background is black and the foreground is white. Useful when a scanner mistakenly inverts image data, or if the image was saved with incorrect tags resulting in inverted data.

Automatic Text Inversion

Detects inverted text regions in a bi-tonal image and automatically inverts them so that OCR engines can recognize the text.

Line Removal

Automatically removes horizontal or vertical lines from a bi-tonal image and reconnects broken characters. Useful for pre-processing a form prior to OCR.

Halftone Removal

Detects and removes shaded regions in a bitonal image, created when thresholding color or gray images to bitonal with shaded image areas. Often referred to as Dot Shade Removal.

Auto Deskew

Detects skew in a bi-tonal document and corrects the image by rotating it. Useful for deskewing scanned documents.

Advanced Photo Effects

The Atalasoft DotImage Advanced Photo Effects (APE) add-on automatically corrects digital photographs. Using proprietary algorithms, functions that typically require many manual steps when carried out in a photo editing tool such as Adobe Photoshop can be accomplished with a single line of code.

Features

- Automatic color correction
- Automatic skin tone correction
- Manual/Automatic levels adjustment
- Portrait enhancement effect
- Color warming and cooling
- ND Gradient filters

- Photo Color Magic *

* Automatically enhances the colors in a photographic image. Designed to improve the colors in any photograph - Results are nearly always better than Automatic levels adjustment.

Examples of these effects can be found in the Photo Effect Demo on the Atalasoft DotImage website.

DWG decoder

Use DWG codec to decode AutoCad DWG and DXF images using the Microsoft .NET Framework.

DWG and DXF files are vector images that are used in CAD/CAM applications for designing or visualizing objects, architecture, maps, or other drawings.

This is an Atalasoft DotImage add-on module for viewing CAD images. In addition to the DWG codec licenses, you must also have a license for Atalasoft DotImage Document Imaging.

Features

- Views both DWG and DXF format files
- Decodes into 24 bit images with arbitrary image resolution
- Decodes at any zoom factor, including automatic "fit-to-page" viewing
- Selectable background color
- Reads individual layouts within a file as frames
- Provides access to names and dimensions of available layouts within a file
- Integrates with the Atalasoft DotImage image processing and windows based or based component features

Office Decoder

The OfficeDecoder codec can be used to decode MS Office documents using the Microsoft .NET Framework. Unlike OfficeAdapterDecoder, it does not require that MS Office be installed.

OfficeDecoder derives the ImageDecoder class and acts like any other Atalasoft DotImage decoder in that it has a Read() method which returns an AtalaImage of the decoded document. It can be included in the RegisteredDecoders collection which allows to open office documents using the AtalaImage constructor.

OfficeDecoder supports Microsoft Office Word, Excel, and PowerPoint documents using the following formats:

- Office 97-2003 Document (.doc)
- Office 97-2003 Document Template (.dot)
- Office Open XML Document (.docx)
- Office Open XML Document Template (.dotx)
- Office Word XML Document (.xml)
- Rich Text Format (.rtf)
- Open Document Text Format (.odt)

- Office 97-2003 Spreadsheet (.xls)
- Office 97-2003 Spreadsheet Template (.xlt)
- Office XML Spreadsheet (.xlsx)
- Office XML Spreadsheet Template (.xltx)
- Microsoft Powerpoint (.ppt)
- Office 97-2003 Presentation (.ppt)
- Office 97-2003 Presentation Template (.pot)
- Office XML Presentation (.pptx)
- Office XML Presentation Template(.potx)

In order to work properly the following modules should be installed along with Atalasoft.dotImage.Office assembly:

- `ISYS11df.dll`
- `ISYSreaders.dll`
- `ISYSreadershd.dll`
- `Perceptive.DocumentFilters.dll`

The modules are available for both x86 and x64 platforms.

OfficeAdapterDecoder

Atalasoft DotImage Office decoders can be used to decode Word, Excel, and PowerPoint documents. The codec is available as a plug-in that integrates with Atalasoft DotImage seamlessly.

The Office assembly includes the OfficeAdapterDecoder class which derives from the ImageDecoder. This class implements the Read method to use Microsoft Office to render pages from the document. It acts like any other Atalasoft DotImage decoder in that it has a Read() method which returns an AtalaImage of the decoded document. It also can be included in the RegisteredDecoders collection which is used when opening images using the AtalaImage constructor.

It supports Microsoft Office Word and Excel, and Powerpoint using the following formats:

- Office 97-2003 Document (.doc)
- Office 97-2003 Document Template (.dot)
- Office Open XML Document (.docx)
- Office Open XML Document Template (.dotx)
- Office Word XML Document (.xml)
- Rich Text Format (.rtf)
- Open Document Text Format (.odt)
- Office 97-2003 Spreadsheet (.xls)
- Office 97-2003 Spreadsheet Template (.xlt)
- Office XML Spreadsheet (.xlsx)
- Office XML Spreadsheet Template (.xltx)
- Microsoft Powerpoint (.ppt)
- Office 97-2003 Presentation (.ppt)

- Office 97-2003 Presentation Template (.pot)
- Office XML Presentation (.pptx)
- Office XML Presentation Template(.potx)

Example how to register the OfficeAdapterDecoder:

```
RegisteredDecoders.Decoders.Add(new OfficeAdapterDecoder());
```

OfficeSession

For situations where the OfficeAdapterDecoder will be used to read a batch of documents, it is recommended that the OfficeSession class be used. The OfficeSession class will keep Office loaded in the background ready to render documents and will remain open until its Close method is called or it is disposed.

If an OfficeSession is not provided to the OfficeAdapterDecoder, the Read method will create and release a new instance of Office each time it is called, which can have a negative impact on performance.

An OfficeSession is created by the static Open method. Alternately, the OpenCached method will create an OfficeSession which will cache the documents it reads, consuming more memory but providing faster performance for subsequent reads on the same stream.

How to: Convert an Office document to an AtalaImage

The OfficeAdapterDecoder class can be used in the same was as any other ImageDecoder. The following C# code demonstrates using the OfficeAdapterDecoder to create a method which will convert a document stream and page index to an AtalaImage.

```
AtalaImage RenderDocument(Stream documentStream, int pageIndex)

{

OfficeAdapterDecoder decoder = new OfficeAdapterDecoder();

return decoder.Read(documentStream, pageIndex, null);

}

How to: Convert a collection of Office documents to AtalaImages using an OfficeSession

The OfficeAdapterDecoder has an overloaded constructor which takes an OfficeSession to
 use to perform rendering.

IEnumerable<AtalaImage> RenderDocuments(Stream[] documentStreams)

{

using(OfficeSession officeSession = OfficeSession.Open())

{

OfficeAdaptorDecoder decoder = new OfficeAdaptorDecoder(officeSession);



foreach(Stream stream in documentStreams)
```

```
{
int frameCount = decoder.GetFrameCount(stream);
for(int frameIndex = 0; frameIndex < frameCount; ++frameIndex)
{
yield return decoder.Read(stream, frameIndex, null);
}
}
}
}
```

How to: Create a WebDocumentRequestHandler configured to handle Office documents using an OfficeSession.

This sample implements the WebDocumentRequestHandler to decode Office documents using an OfficeSession in the WDV.

```
internal sealed class OfficeWebDocumentRequestHandler : WebDocumentRequestHandler
{
private readonly OfficeSession officeSession = OfficeSession.Open();
public OfficeWebDocumentRequestHandler()
{
OfficeAdapterDecoder officeDecoder = new OfficeAdapterDecoder(this.officeSession);
this.DocumentInfoRequested += (o, e) =>
{
string fileName = HttpContext.Current.Request.MapPath(e.FilePath);
if(File.Exists(fileName))
{
using(Stream stream = File.OpenRead(fileName))
{
if(officeDecoder.IsValidFormat(stream))
{
ImageInfo imageInfo = officeDecoder.GetImageInfo(stream);
int dpi = officeDecoder.Resolution;
e.Resolution = new Dpi(dpi, dpi, ResolutionUnit.DotsPerInch);
e.PageCount = imageInfo.FrameCount;
e.ColorDepth = imageInfo.ColorDepth;
```

```
e.PageSize = imageInfo.Size;
}
}
}
};
this.ImageRequested += (o, e) =>
{
string fileName = HttpContext.Current.Request.MapPath(e.FilePath);
if(File.Exists(fileName))
{
using(Stream stream = File.OpenRead(fileName))
{
if(officeDecoder.IsValidFormat(stream))
{
e.Image = officeDecoder.Read(stream, e.FrameIndex, null);
}
}
}
};
}
~OfficeWebDocumentRequestHandler()
{
this.officeSession.Close();
}
}
```

Atalasoft DotImage ISIS

Atalasoft DotImage ISIS is a .NET component for capturing images from scanners that use an ISIS driver. It takes advantage of the speed and stability of ISIS drivers available from most scanner manufacturers.

Supported Features

- Direct in-memory scanning
- Scanning directly to a variety of file formats (provided through ISIS drivers)
- Access to dozens of scanner property values

- Automatic Document Feeder support
- Supports custom interface creation or, use the default driver interface
- Support for saving and restoring scanner settings to a file or stream
- Barcode detection (when supported by the scanner)
- In-memory images can be returned as a .NET Bitmap or an AtalaImage
- Use the IsisController for more direct lower-level scanner control

Chapter 2

Use Atalasoft DotImage

This chapter describes information resources that are available to help you learn about Atalasoft DotImage.

Atalasoft DotImage documentation

The Atalasoft DotImage documentation provides detailed information on how to develop Windows Form and ASP.NET WebForm applications using the Atalasoft DotImage SDK.

We recommend that you read the Getting Started section before beginning your first project. Atalasoft DotImage has many classes and namespaces. This section should help you gain an understanding of where to start and of the classes that are relevant to your specific imaging needs.

The documentation covers the Atalasoft DotImage Photo, Atalasoft DotImage Photo Pro, and Atalasoft DotImage Document Imaging products as well as all Atalasoft DotImage Add-ons. The Atalasoft DotImage documentation is available as an HTML Help collection and as a PDF suitable for printing.

Atalasoft DotImage NuGet Packages

NuGet is a deployment mechanism for development artifacts (.NET assemblies, JavaScript files, etc.). It is the standard deployment option for Microsoft products, and even for the .NET Framework itself.

By introducing NuGet packages for Atalasoft DotImage, it becomes possible for users to install Atalasoft DotImage components right from within Visual Studio, eliminating the need to provide a complete Atalasoft DotImage installer. This streamlines the whole development process. NuGet packages also simplify upgrading to newer versions of the packages.

Note that you cannot use the NuGet infrastructure to generate a Atalasoft DotImage license. Historically, Activation Wizard tools were part of the complete installation package and were not available otherwise. To accommodate Atalasoft DotImage NuGet packages and simplify the license generation process, Activation Wizard is provided as an extension for Visual Studio, so clients can request trial or generate Atalasoft DotImage licenses without a complete installation of Atalasoft DotImage SDK in their environment.

Visual Studio Activation Wizard Extension

For convenience, the Activation Wizard is ported to a VSIX, so NuGet package consumers do not have to download a full installer for the Atalasoft DotImage SDK to get the Activation Wizard.

The Activation Wizard extension can be installed or upgraded right from within Visual Studio, using Tools -> Extension and Updates.

After installing Atalasoft DotImage or downloading the Atalasoft DotImage NuGet packages, you need to activate your serial number. You can do this using the "Activate Atalasoft DotImage.." item from the Visual Studio Tools menu.

The activation process varies with your situation:

- If you have not yet purchased Atalasoft DotImage, select the **Request 30-day Evaluation** radio button and follow the instructions in Requesting an Evaluation License.
- If you have purchased Atalasoft DotImage, or related toolkits, you need your serial number to continue with the activation process.

Your Atalasoft DotImage serial number belongs only to you and must be kept in a safe place. You need this serial number to request license files. The activation wizard will generate the necessary license files (.lic) in your Local Application Data folder, located at:

```
%LOCALAPPADATA%\Atalasoft\DotImage X.Y
```

Where X and Y are the major and minor versions of the release covered by the license.

Getting Started with Web Capture

Follow these steps to create a new capture-enabled Web project. Topics include adding the document viewer and scanning controls to your Web page, and handling uploaded content on the server. Several steps will contain cross-references to other sections with more detailed information.

This guide is intended to be followed exactly, but it is not intended to give you a solution that is ready to deploy. Once you have succeeded building the example project, you can begin modifying it to fit your organization.

Atalasoft DotImage demos

The demo programs provided at our demo gallery demonstrate the wide range of capabilities available to you while developing applications with Atalasoft DotImage.

These demos are designed as a reference and an evaluation tool, and are provided as compiled executables, as well as Visual Studio projects in C# and VB.NET in Visual Studio 2012. The executables generally run without a license, but licenses are required to compile the source code.

To view a complete list of demos, go to: http://www.atalasoft.com/Support/Sample-Applications.

Set up a new project

A capture-enabled application requires these basic elements:

- A client-side ASPX page containing the scanning controls and document viewer.
- A server-side ASHX handler for the Web Document Viewer.
- A server-side ASHX handler for the Web Capture back end.

- WebCapture and WebDocumentViewer resources files.
- An upload location for scanned documents.

Start by creating a new ASP.NET Web Application in Visual Studio.

ℹ In the following instructions the project is called BasicWebCapture.

Visual Studio automatically gives you Default.aspx as a page, which we will use for placing the scanning controls and viewer.

Modify the MSBuild project file when using .NET 5

If you are using .NET 5, you need to modify the MSBuild project file (which has a .csproj extension) to add Windows Forms support and enable Atalasoft DotImage libraries to be imported.

Search the MSBuild project file to see if <UseWindowsForms> is already in the file. If so, change `false` to `true`. If not, add the following line to the file:

```
<UseWindowsForm>true</UseWindowsForm>
```

Add assembly references

Add the following DotImage assemblies to your project:

- Atalasoft.dotImage.WebControls
- Atalasoft.Shared

In a default installation, these assemblies can be found in the following folders:

- .NET Framework 4.5.2 (64-bit): `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\4.5.2\x64`
- .NET Framework 4.5.2 (32-bit): `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\4.5.2\x86`
- .NET Framework 3.5 (64-bit): `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\3.5\x64`
- .NET Framework 3.5 (32-bit): `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\3.5\x86`

There may be further dependencies on any of the remaining DotImage assemblies. Include all DotImage assemblies in your project if there are problems resolving them.

Copy resources

DotImage comes with two sets of resources: WebCapture and WebDocumentViewer. In a default .Net installation, these directories are located in `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\WebResources`.

Copy the WebCapture and WebDocumentViewer directories into the root of your project.

Create the upload location

Create a new directory in the root of your project called `atala-capture-upload`. This is the default path that will be used for storing images uploaded by the scanning controls.

If you need to change the location of the upload path (for example, to place it in a location outside of your document root), you can set an `atala_uploadpath` value in the `appSettings` section of either your `web.config` or `app.config`.

```
<appSettings>
  <add key="atala_uploadpath" value="c:\path\to\location"/>
```

```
</appSettings>
```

Add the Web Document Viewer handler

The Web Document Viewer handler is responsible for communicating with the Web Document Viewer embedded in your page, and is separate from the capture handler.

Add a new Generic Handler to your project. For the purposes of this guide, it is assumed this file will be called `WebDocViewerHandler`.

Change the class definition to extend `WebDocumentRequestHandler` (part of Atalasoft.Imaging.WebControls). Your handler should resemble the following example.

C#

```
using Atalasoft.Imaging.WebControls;
namespace BasicWebCapture
{
public class WebDocViewerHandler : WebDocumentRequestHandler
{ }
}
```

There is no need for further modification to your handler.

Add the Web Capture handler

The Web Capture handler is responsible for handling file uploads from the scanning controls embedded in your page, and routing them to their next destination along with any necessary metadata. It is also responsible for supplying the scanning controls with the available content and document types, and status information.

For this guide, we will create a custom handler that provides a few static content and document types, and saves uploaded files to another location. Using this baseline, you can continue modifying the handler to suit your own document handling needs.

If your organization uses Kofax Import Connector (KIC), DotImage ships with handlers to connect to the service.

ℹ Kofax Import Connector handlers are only supported with .NET Framework 3.5 and 4.5.2.

Create a handler

Add a new Generic Handler to your project. For the purposes of this guide, it is assumed this file will be called `WebCaptureHandler.ashx`.

The handler should be modified to extend from WebCaptureRequestHandler (part of Atalasoft.Imaging.WebControls.Capture), and should not implement the IHttpHandler interface, as is done when a generic handler is first created. Instead your handler will need to override several methods of WebCaptureRequestHandler. Your handler should resemble the following example.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Web;
```

```
using Atalasoft.Imaging.WebControls.Capture;

namespace BasicWebCapture
{
    public class WebCaptureHandler : WebCaptureRequestHandler
    {
        protected override List<string> GetContentTypeList(HttpContext context)
        {
            // ...
        }

        protected override List<Dictionary<string, string>>
 GetContentTypeDescription(HttpContext context, String contentType)
        {
            // ...
        }

        protected override Dictionary<string, string> ImportDocument(HttpContext
 context, string filename,
            string contentType, string contentTypeDocumentClass, string
 contentTypeDescription)
        {
            // ...
        }
    }
}
```

The three stubs represent the minimum number of methods that must be implemented for basic functionality, but there are other methods available in the public API that can also have their behavior overridden, such as methods to generate IDs or query the status of documents. Refer to the accompanying object reference for the complete WebCaptureRequestHandler API.

GetContentTypeList

This method returns the collection of available content types that can be used to organize scanned and uploaded documents. Content types are the top-level organizational unit, and each one has its own collection of document types (also called document classes) below it.

For this example, GetContentTypeList will be implemented to return a fixed list of two types: Accounts and HR. In a real system, this would probably query a database or other data source instead. In the KIC handler, this method queries the system for these values.

C#

```
protected override List<string> GetContentTypeList(HttpContext context)
{
    return new List<string>() { "Accounts", "HR" };
}
```

GetContentTypeDescription

This method returns a collection of data describing all the document types under a single content type. The return data is a list of dictionaries, where each dictionary contains a set of properties describing a single document type. In this example, the only property returned for a document type is its documentClass, which serves as its name.

C#

```
protected override List<Dictionary<string, string>>
 GetContentTypeDescription(HttpContext
 context, String contentType)
    {
        switch (contentType)
```

```
        {
            case "Accounts":
                return CreateDocumentClassDictionaryList(new string[]
{ "Invoices",
                "Purchase Orders" });
            case "HR":
                return CreateDocumentClassDictionaryList(new string[]
{ "Resumes" });
            default:
                return base.GetContentTypeDescription(context, contentType);
        }
    }

private List<Dictionary<String, String>>
createDocumentClassDictionaryList(String[] docList)
{
 return docList.Select(doc => new Dictionary<String, String> {{"documentClass",
doc}}).ToList();
}
```

A helper method is provided to produce the actual list of document types, while GetContentTypeDescription switches on a given content type to determine what document types should be included in the list. As with content types, it is expected that this data will originate from another data source, instead of being hard-coded.

ImportDocument

This method is responsible for actually moving a document and its metadata to its real destination, which could be a directory, database, or system such as KIC.

C#

```
protected override Dictionary<string, string> ImportDocument(HttpContext context,
 string filename,
    string contentType, string contentTypeDocumentClass, string
 contentTypeDescription)
{
    string docId = Guid.NewGuid().ToString();
    string importPath = @"C:\DocumentStore";

    importPath = Path.Combine(importPath, contentType);
    importPath = Path.Combine(importPath, contentTypeDocumentClass);
    importPath = Path.Combine(importPath, docId + "." +
 Path.GetExtension(filename));

    string uploadPath = Path.Combine(UploadPath, filename);

    File.Copy(uploadPath, importPath);

    return new Dictionary<string, string>()
    {
        { "success", "true" },
        { "id", docId },
        { "status", "Import succeeded" },
    };
}
```

In this example, imported documents are copied into a directory tree rooted at C:\DocumentStore, using the content type and document class as subdirectories for organizing files. The imported file is copied and given a new name based on a GUID, which is also passed back to the client in the "id" field of a dictionary. The id could be used by the client to query the handler at a future time for the status of the imported document, but this functionality is not included in the guide.

Set up the scanning controls and viewer

The setup for scanning just requires placing some JavaScript, CSS, and HTML into your page. The page itself could be HTML, ASPX, JSP, or anything else, as the client-side technology is not directly tied to .NET or IIS. For this guide however, we will update the document Default.aspx, which was originally included in the new project.

Include the resources

Include the following script and link tags in your page's head section to include the necessary Web Document Viewer and Web Capture code and dependencies.

HTML

```
<!-- Script includes for Web Viewing -->
<script src="WebDocViewer/jquery-3.4.1.min.js" type="text/javascript"></script>
<script src="WebDocViewer/atalaWebDocumentViewer.js" type="text/javascript"></
script>

<!-- Style for Web Viewer -->
<link href="WebDocViewer/jquery-ui-1.12.1.custom.css" rel="Stylesheet" type="text/
css" />
<link href="WebDocViewer/atalaWebDocumentViewer.css" rel="Stylesheet" type="text/
css" />

<!-- Script includes for Web Capture -->
<script src="WebCapture/atalaWebCapture.js" type="text/javascript"></script>
```

Configure the controls

The scanning and viewing controls need to be initialized and configured to set up connections to the right handlers, specify behavior for events, and so forth. This can be done with another block of JavaScript, either included or pasted directly within your page's head somewhere below the included dependencies.

JavaScript

```
<script type="text/javascript">
    // Initialize Web Scanning and Web Viewing
    $(function() {
        try {
            var viewer = new Atalasoft.Controls.WebDocumentViewer({
                parent: $('.atala-document-container'),
                toolbarparent: $('.atala-document-toolbar'),
                serverurl: 'WebDocViewerHandler'
            });

            Atalasoft.Controls.Capture.WebScanning.initialize({
                handlerUrl: 'WebCaptureHandler',
                onUploadCompleted: function(eventName, eventObj) {
                    if (eventObj.success) {
                        viewer.OpenUrl("atala-capture-upload/" +
 eventObj.documentFilename);
                        Atalasoft.Controls.Capture.CaptureService.documentFilename
 = eventObj.documentFilename;
                    }
                },
                scanningOptions: { pixelType: 0 }
            });

            Atalasoft.Controls.Capture.CaptureService.initialize({
```

```
                handlerUrl: 'WebCaptureHandler.'
            });
        }
        catch (error) {
            alert('Thrown error: ' + error.description);
        }
    });
</script>
```

Note that the URL for the WebDocViewer handler is specified once and the URL for the WebCapture handler is specified twice, since two capture services must be initialized.

There are several additional options and handlers that can be specified in the initialization routines for scanning and viewing. This example represents the minimal configuration necessary for scanning with an integrated document viewer.

Add the UI

Add the following HTML to your project to create a basic viewer UI. This includes the Web Document Viewer, drop-down boxes to choose scanners, content types, and document types, and buttons to drive the UI. The scanning demos included with DotImage also include more complete examples.

HTML

```
<p>Select Scanner:
  <select class="atala-scanner-list" disabled="disabled" name="scannerList"
 style="width: 22em">
    <option selected="selected">(no scanners available)</option>
  </select>
</p>
<p>Content Type:
  <select class="atala-content-type-list" style="width:30em"></select>
</p>
<p>Document Type:
  <select class="atala-content-type-document-list" style="width:30em"></select>
</p>
<p>
  <input type="button" class="atala-scan-button" value="Scan" />
  <input type="button" class="atala-import-button" value="Import" />
</p>
<div>
  <div class="atala-document-toolbar" style="width: 670px;"></div>
  <div class="atala-document-container" style="width: 670px; height: 500px;"></div>
</div>
```

Wrap-up

Your project should be ready to deploy to an app server. It is also ready to run from your developing environment, for testing purposes.

Web server Upload size limits

By default, IIS limits uploads to 30MB. Estimate the maximum upload size your application could generate, and adjust the server limits accordingly.

Deploy on multiuser environment

There are scenarios where Web Capture Service is used on multiuser environments (MS Terminal Server, Citrix). On these environments, multiple users work with Web Capture Service at the same

time from different Windows logon sessions. We need to support such environments and provide the same experience as on single-user machine.

Terminal server

When using a terminal server, users can connect to the scan server simultaneously and perform scanning tasks or import files in parallel.

In this case, the Web Capture Service Host determines who exactly has made a request to it, and forwards the request to the appropriate Web Capture Service Worker which, in turn, works with devices and files that are available to the specific user. For the end user, this detection process is transparent, and takes the same as in the simple single-user environment.

Web Capture Service can work only with scanners attached to a remote Terminal Server. Locally connected scanners are not available in this scenario. The same goes for file import – Web Capture Service provides access to files on a Terminal Server.

Citrix

The major difference, in comparison with the standalone scenario, when both the Browser app and Web Capture Service are installed on client machine, is that Web Capture Service is physically running on a remote Citrix server, while a scanner is connected to the client user’s computer. This works transparently for Web Capture Service when Citrix TWAIN Redirection is enabled.

Installation

Web Capture Service can be installed as a Windows Service, enabling the multiuser support features described above by using the INSTALLASSERVICE command line option as shown below:

```
msiexec /I Kofax.WebCapture.Installer.msi INSTALLASSERVICE=1
```

The same command line parameter should be passed to upgrade Web Capture Service installed as Windows Service.

Administrator rights are required to deploy and upgrade Web Capture Service installed as Windows Service; therefore it is the responsibility of server Administrator to deploy/upgrade it.

Upgrade

You cannot upgrade Web Capture Service installed as a Windows Service to the standalone version. The following error message is shown if you try to do so:

```
This application can't be installed because you already have Web Capture
Service install as Windows service.
```

However, upgrading from the standalone installation to Windows Service is supported and works as expected.

Configure Kofax Import Connector

This is not intended to be a full set of instructions to install, set up, and maintain a Kofax Import Connector server. The following information provides the minimum amount of configuration needed for the DotImage Web Scanning Control to successfully connect, and import into Kofax Import Connector.

For information on connecting to an already configured Kofax Import Connector server, see Connect to Kofax Import Connector (KIC) Web Services.

ⓘ Kofax Import Connector handlers are only supported with .NET Framework 3.5 and 4.5.2.

Required license

For the KIC server to accept documents imported from the DotImage assembly, a **KIC – Electronic Documents – Web Service interface**.

The license must be installed on your KIC server.

To verify that the correct minimum license has been installed go to the Message Connector Monitor, which by default is located on the KIC server at https://localhost:25086/file/index.html where under the **Status->license** section.

Configure the service

The DotImage Web scanning control connects via KIC's service via a server-side handler that extends the KicHandler found in the Atalasoft.dotImage.WebControls assembly.

Once in the message connector, go to the "General" section, and verify that the ".

1. From the App Programs list, select **Kofax** > **KIC Electronic Documents** > **Message Connector Configuration**.

   The message connector opens.
2. in the **General** section, verify the **Own Computer Name** is filled in with the current server's domain qualified name.
3. Next, go to the **Web-Service Input** section.
   - If only a HTTP based connection is desired set the HTTPS port to 0

     This will be the port which the endpoint in the application's `web.config` will point to. If HTTPS is desired, then enter the port which will be used.
   - If HTTPS is enabled the HTTP port will not be able to be connected to, and the endpoint in the application's `web.config` will need to point at the URL using the HTTPS port.
4. Once all of the desired changes to the KIC Message Connector have been made save, and restart the Message Connector service.

Configure the Electronic Documents plugin

In the Kofax Capture (KC) Administration application, open the 'Electronic Documents->Configuration' window, and configure the necessary Connections, and Destinations.

When finished, stop and start the service.

Test the configuration

To test that the KIC server has been minimally configured correctly in a browser either on the server, or at a client that might connect to the server enter the following URLs (all on one line of course):

HTTP enabled webservice

```
http://[kic_servername]:[http_port]/soap/tsl/Import?<OwnerReference>myref</
OwnerReference>
<Address>importaddr</Address><Part><ContentType>text/plain</ContentType>
<Content><Text>hello</Text></Content></Part>
```

HTTPS enabled webservice

```
https://[kic_servername]:[https_port]/soap/tsl/Import?<OwnerReference>myref</
OwnerReference>
<Address>importaddr</Address><Part><ContentType>text/plain</ContentType>
<Content><Text>hello</Text></Content></Part>
```

Upload sizes and limits

By default, IIS limits uploads to 30MB. If your application may sometimes generate larger uploads, you will need to adjust this limit for the server, or at least for your application.

Estimate upload sizes

The size of an upload is approximately the sum of the compressed sizes of the uploaded images x 4/3 (1.333).

The calculations below are for images. Remember that duplex scanning generates two images per page, minus any blank sides discarded by setting discardBlankPages:true.

Raw uncompressed image size

Uncompressed image size in bytes = (width x DPI x height x DPI x depth) / 8

Where depth is 24 for color, 8 for grayscale, and 1 for B&W images.

Example, an 8.5" x 11" color page, scanned at 200 DPI: (8.5 x 200 x 11 x 200 x 24) / 8 = 11,220,000 bytes (~11MB)

Compression ratios

Typical office documents in B&W will compress by ~10X. White space increases the compression, lots of text or detailed graphics of any kind decreases the compression. 50KB per compressed B&W image is not a bad average, 70KB is conservative.

Grayscale and color images will compress by 20X-30X, sometimes more. As with B&W, blank paper compresses more, detailed content compresses less.

For our example 8.5" x 11" color page scanned at 200 DPI, with a raw size of 11MB we estimate a compressed size in the range 374KB - 560KB.

Factor in Base64 encoding

We multiply by 4/3 (1.33) because uploads are encoded in Base64, which encodes 3 binary bytes as 4 text characters.

Adjust the IIS upload limit

IIS, by default, limits any single upload to 30MB. If you attempt a larger upload, the server will (oddly) return a 404 error.

If you expect to upload larger files, you will need to increase this limit

Edit `web.config`

(from `http://www.webtrenches.com/post.cfm/iis7-file-upload-size-limits`)

Add to `web.config`

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="524288000"/>
    </requestFiltering>
  </security>
</system.webServer>
```

If you add the above code to the `web.config` file for your site, you can control the maximum upload size for your site. In many cases, the `system.webServer` node will already be in the file, so just add the security node within that.

The `maxAllowedContentLength` is in BYTES not kilobytes.

You may also need to restart your Web site (not the whole server) to enable the setting.

Configure interactivity

The limit can also be changed interactively (quoting from the same blog post)

1. Open **IIS 7 SnapIn**.
2. Select the Web siteyou want enable to accept large file uploads.
3. In the main window double-click **Request filtering**.
   The Request filtering window opens.
   The tab list shows options such as file name extensions, rules, and hidden segments.
4. Select one fo the tabs. and in the main window right-click and select **Edit Feature Settings**.
5. Modify the **Maximum allowed content length (bytes)**.

In-Memory limitation

Note that you should not expect Atalasoft DotImage to collect and upload a set of images that exceeds (approximately) 500MB of memory when uncompressed, whether the uploaded file is compressed or not. This corresponds to approximately 20 pages of 200 DPI full-color US Letter or A4 size. Grayscale images use 1/3 the space of color images, and B&W images use 1/24.

Server timeouts

With larger uploads, you may need to also increase the Params.serverTimeout: Integer value, which is 20 seconds by default.

Document management

The following set of tutorials are designed to teach you how to use Atalasoft DotImage by taking you through the development of a real world application, an enterprise document management solution.

To complete the tutorial, you need the following:

- Atalasoft DotImage installed on your computer
- Visual Studio 2012 or later
- .NET 3.5

Off-the-shelf document management solutions for your business may never meet all of your company's requirements, can be painful to deploy, expensive to train your users on, and costly to license. Atalasoft DotImage Document Imaging enables .NET developers to quickly and easily build document management solutions that meet 100% of your company's requirements with reasonable licensing, and with advanced Enterprise 2.0 features typically not found in off-the-shelf solutions.

This step-by-step tutorial demonstrates how to build an n-tier enterprise client-server document capture and viewing application in just a few hours. The tutorial describes the following:

- Building an application that captures documents from a scanner.
- Submitting those documents to a server via services.
- Storing the document and associated metadata into a database.
- Viewing the documents in a zero footprint web-client.
- Adding collaboration features with web-based annotations support.

To get started, continue to the next section.

Capture and view documents with a scanning client

This tutorial builds a windows forms application that can be deployed via an MSI installer or .NET Click-Once deployment.

This tutorial consists of the following lessons:

- Create the Windows Forms project
- Design the user Interface
- Add toolstrip controls
- Open a multipage docment from a file
- Configure AutoZoom and image scaling
- Save a multipage document to a file
- Capture documents from a scanner

Create the Windows Forms project

1. Create a new Windows Forms project called CaptureClient.
2. Choose the language you wish to code in, such as C#.

Design the user interface

1. Add a ToolStrip control to the top of the form by dragging it from the Visual Studio Toolbox.
2. Drag a DocumentViewer control from the Visual Studio Toolbox Atalasoft DotImage tab.
   Use this control to open, view, navigate, and save a multipage document.
3. Select Fill from the Dock property located in the property grid of the DocumentViewer control.
   This docks the DocumentViewer to the form so that resizing the form, also resizes the control.
4. Drag an Acquisition Component from the same Atalasoft DotImage tab in the toolbox.
   This component acquires images from scanners.
   The Atalasoft DotImage references added to your project up to this point include:
   - Atalasoft.dotImage
   - Atalasoft.dotImage.WinControls
   - Atalasoft.DotTwain
   - Atalasoft.Shared

   The form created should have a Thumbnail Viewer in the left pane and an ImageViewer on the right.

Add toolstrip controls

In this lesson you add the toolstrip control that allows the user to interact with your application.

When complete, your toolstrip should have the six commands listed in the table. You also need to set the ToolStrip ImageScalingSize to 24,24 and name each control in the toolstrip as shown in the table.

Control	Name
Open from a file	tsbOpen
Save to a file	tsbSave
Choose a scanner	tsbChooseScanner
Scan	tsbScan
Fit to width	tbsFitToWidth
View full size	tbsFullSize

1. Select **ComboBox** from the Add ToolStripButton menu.
   You need the ComboBox for the "Choose a scanner" control.
2. Represent your commands using images from your own icon library or use text.

Open a multipage document from a file

After the user interface is finished, use this procedure to start coding.

1. Add "using" ("Imports" in VB) statements to the form as shown in the following C# example. This prevents you from having to declare the entire namespace for each class reference.

```
using System.IO;
using Atalasoft.Imaging;
using Atalasoft.Imaging.Codec;
using Atalasoft.Twain;
```

2. Drag an OpenFileDialog from the Visual Studio Toolbox to your form.
3. Double-click the **Open** button.
4. Add the following code in the event handler:

```
private void tsbOpen_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog(this) == DialogResult.OK)
    {
      documentViewer1.Open(new FileSystemImageSource(new string[]
     { openFileDialog1.FileName }, true));
    }
}
```

   This code presents an Open File dialog box. If an image is selected, it loads the image into the DocumentViewer control using a FileSystemImageSource. An ImageSource is designed to efficiently manage multiple images, keeping in memory only those that are necessary. This is the ideal way to navigate a multipage document such as a TIFF or PDF.

5. Build and run your project.
   a. Open a multipage TIFF such as the one found in Images\Documents under the Atalasoft DotImage installation folder.
   b. Use your toolstrip to open, view and navigate the pages of any supported image format.

Configure AutoZoom and image scaling

In this lesson, you add two buttons. Each affects document scaling by using the AutoZoom setting.

- The first button makes the entire document width visible in the viewer.
- The second button restores the document to full size view.

The first button is implemented by setting the AutoZoom property to the FitToWidth setting to make the entire document width visible in the viewer.

The Full Size View button sets the AutoZoom property to None and the Zoom property to 1.0.

ⓘ When users view a page at a zoom level less than 1.0, the control uses the fast ScaleToGray resampling on 1-bit black and white images when the AntialiasDisplay property for the DocumentViewer is set to a value other than none.

1. Double-click each button.

2. Add the following event handler to your application to configure these buttons:

```
private void tsbFitToWidth_Click(object sender, EventArgs e)
{
    documentViewer1.ImageControl.AutoZoom =
 Atalasoft.Imaging.WinControls.AutoZoomMode.FitToWidth;
}

private void tsbFullSize_Click(object sender, EventArgs e)
{
    documentViewer1.ImageControl.AutoZoom =
 Atalasoft.Imaging.WinControls.AutoZoomMode.None;
    documentViewer1.ImageControl.Zoom = 1.0;
}
```

   You now have a simple, fast, and useful multipage document viewer.

Save a multipage document to a file

Use this procedure to save a multipage document to a file.

1. Drag a SaveFileDialog from the Visual Studio Toolbox to your form.
2. As the Save() method of the DocumentViewer handles all of the complexities involved in efficiently saving a multipaged document, you need only invoke the Save() method and pass in the file name or stream as well as an instance of ImageEncoder of the format in which you want to save.
3. Double-click the **Save** button.
4. Add the following code in the event handler:

```
private void tsbSave_Click(object sender, EventArgs e)
{
    if (saveFileDialog1.ShowDialog(this) == DialogResult.OK)
    {
        documentViewer1.Save(saveFileDialog1.FileName, new TiffEncoder());
    }
}
```

Capture documents from a scanner

In this lesson, you add the capability of capturing documents from a scanner to your application. The lesson uses DotTwain to scan documents directly into the DocumentViewer control.

1. Double-click the Acquisition control that you dropped onto the form in an earlier lesson. This creates the event handler for ImageAcquired. That event is fired for each image captured while the document is being scanned.
2. DotTwain stores images as .NET bitmaps. Therefore you must translate them to an AtalaImage using AtalaImage.FromBitmap. Images are added to the Document Viewer with the Add() method in the Source property. The Add() method accepts a filename, stream, or an in-memory AtalaImage object. Add the following code to the ImageAcquired event handler.

```
privatevoid acquisition1_ImageAcquired(object sender,
AcquireEventArgs e)
{
documentViewer1.Add(AtalaImage.FromBitmap(e.Image));
}
```

3. Now that the scan handling is in place, you must initiate the scan.
   a. Double-click the Scan button in the ToolStrip.
   b. Invoke the Acquire() method of the acquisition object as shown here. The code starts the scan using the default scanner and displays the scanner's default dialog box.

```
private void tsbScan_Click(object sender, EventArgs e)
{
acquisition1.Acquire();
}
```

Select a scanner

As many systems have more than one TWAIN device configured and the default device may not be the one the user needs to use for capture, you must populate the drop-down box previously added to the ToolStrip with all the TWAIN drivers configured on the current system.

This enables the user to select the device from which images are scanned.

1. Change the DropDownStyle property in the tsbChooseScanner control to DropDownList.
2. Add code to populate the drop-down menu with the names of all configured scanners.

```
public Form1()
{
    InitializeComponent();
     if (acquisition1.SystemHasTwain)
    {
        foreach (Device dev in acquisition1.Devices)
        {
            tsbChooseScanner.Items.Add(dev.Identity.ProductName);
        }
                if (acquisition1.Devices.Count > 0)
                    tsbChooseScanner.SelectedItem =
 acquisition1.Devices.Default.Identity.ProductName;
                else
                {
                    tsbChooseScanner.Enabled = false;
                    tsbScan.Enabled = false;
                }
    }
}
```

3. Change the acquisition code to use the selected device. Loop through the devices collection, in a manner similar to that used in the preceding code.

```
private void tsbScan_Click(object sender, EventArgs e)
{
    foreach (Device dev in acquisition1.Devices)
    {
        if (dev.Identity.ProductName == tsbChooseScanner.SelectedItem.ToString())
        {
            dev.Acquire();
        }
    }
```

```
}
```

You now have a full-featured capture application that can use any installed scanner, open any multipage document, and save the captured document as a TIFF.

To see the full solution for this lesson, go to Application Source Code.

You have now completed all of the lessons in part 1 of this tutorial. For suggestions on enhancing this application, see Recommendations. To continue to the next part, go to Tutorial: Submit Captured documents to a central database repository.

Application source code

Developing a fully featured capture application such as this one is easy using Atalasoft DotImage Document Imaging. The easy to use API, and visual controls enable us to configure this application with minimal code.

The entire code for this application is as follows:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using Atalasoft.Imaging;
using Atalasoft.Imaging.Codec;
using Atalasoft.Twain;

namespace CaptureClient
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            if (acquisition1.SystemHasTwain)
            {
                foreach (Device dev in acquisition1.Devices)
                {
                    tsbChooseScanner.Items.Add(dev.Identity.ProductName);
                }
                if (acquisition1.Devices.Count > 0)
                    tsbChooseScanner.SelectedItem =
 acquisition1.Devices.Default.Identity.ProductName;
                else
                {
                    tsbChooseScanner.Enabled = false;
                    tsbScan.Enabled = false;
                }
           }
        }

        private void tsbOpen_Click(object sender, EventArgs e)
        {
            if (openFileDialog1.ShowDialog(this) == DialogResult.OK)
            {
                documentViewer1.Open(new FileSystemImageSource(new string[]
 { openFileDialog1.FileName }, true));
```

```
            }
        }

        private void tsbSave_Click(object sender, EventArgs e)
        {
            if (saveFileDialog1.ShowDialog(this) == DialogResult.OK)
            {
                documentViewer1.Save(saveFileDialog1.FileName, new TiffEncoder());
            }
        }

        private void acquisition1_ImageAcquired(object sender, AcquireEventArgs e)
        {
            documentViewer1.Add(AtalaImage.FromBitmap(e.Image), "", "");
        }

        private void tsbScan_Click(object sender, EventArgs e)
        {
            foreach (Device dev in acquisition1.Devices)
            {
                if (dev.Identity.ProductName ==
 tsbChooseScanner.SelectedItem.ToString())
                {
                    dev.Acquire();
                }
            }
        }

        private void tsbFitToWidth_Click(object sender, EventArgs e)
        {
            documentViewer1.ImageControl.AutoZoom =
 Atalasoft.Imaging.WinControls.AutoZoomMode.FitToWidth;
        }

        private void tsbFullSize_Click(object sender, EventArgs e)
        {
            documentViewer1.ImageControl.AutoZoom =
 Atalasoft.Imaging.WinControls.AutoZoomMode.None;
            documentViewer1.ImageControl.Zoom = 1.0;
        }

    }
}
```

Recommendations

You can enhance this application simply by setting some properties or by writing just a small amount of code.

For example:

- Add AutoDeskew functionality as the image is scanned.
- Set the AutoDragDrop property of the ThumbnailControl to allow for the reordering of pages.
- Add additional navigation capabilities.
- Add mouse tools such as zoom and zoom to area.
- Generate searchable PDFs.

Submit captured documents to a central database repository

In the previous tutorial, you built a scanning client using Atalasoft DotImage that saves captured images to the local file system. In an enterprise capture application, however, documents need to be stored in a central database.

In the spirit of developing a true Enterprise 2.0 application with a Service Oriented Architecture, services should be the public interface to the database repository.

This tutorial demonstrates how to configure services to transmit images from a client application to a service and into a SQL database.

You must complete the steps in Lesson 1 before continuing with this lesson. This lesson uses very little Atalasoft DotImage code: refer to the MSDN documentation and Microsoft support for issues that do not directly involve Atalasoft DotImage code.

This tutorial consists of the following lessons:

- Create the SQL database
- Create the data abstraction layer
- Create the service
- Connect the capture client to the service
- Show progress while the document loads

Create the SQL database

Be sure that SQL Express is installed on your computer before beginning this lesson.

1. Add a new ASP.NET site to the solution.
2. Add a new item to the ASP.NET site. This creates the database that stores the multipaged document and associated metadata.
3. Select SQL Database, and name it `ImageDatabase.mdf`.
4. When prompted, select **To add the database to the App Data folder** in the site.
5. Create a table with the fields and field types shown in the table.

Column Name	Data Type	Allow Nulls
ImageID	uniqueidentifier	No
ImageData	varbinary(MAX)	Yes
PageCount	int	No
DateTimeCreated	datetime	No
Name	nvarchar(50)	Yes

6. Set the default value of DateTimeCreated to getdate().
7. Set the default value of PageCount to 0.
8. Select ImageID as the primary key.

Create the data abstraction layer

In this lesson, you submit captured documents to a central database repository. Begin by adding the ImageDatabaseTableAdapter command to your ImageDatabase.

1. Use Add New Item to add a new DataSet to your Web siteproject.
2. Name the newly added item "ImageDatabase".
3. If the TableAdapter Configuration Wizard appears, click **Cancel**.
4. Drag the ImageDatabase table from the Server Explorer to the ImageDatabase.xsd window.
5. Create two new queries by right-clicking the ImageDatabaseTableAdapter and selecting AddQuery.
6. Use the following SQL code to name the first query CreateRecord.

```
INSERT INTO ImageDatabase (ImageID, Name) VALUES (@ImageID,
@Name)
```

7. Use the following SQL code to name the second query UpdateImageData.

```
UPDATE ImageDatabase
SET ImageData = @ImageData, PageCount
WHERE (ImageID = @ImageID)
```

Create the service

In this lesson, you will submit captured documents to a central database repository.

1. Right-click the Atalasoft DotImage WebServer project.
2. Select Add New Item.
3. Choose Web Service and give your service the name ImageUpLoadService.

This creates a new service in the Web site, with the code in the App_Code folder. You will use the service to submit chunks of each page of the captured document located at the client. To do this, you need to create three methods:

- StartUpload
- AppendChunk
- FinishUpload

1. Add the following skeleton code to the service. Be sure to delete the Hello World default method.

WebMethod
`public string StartUpload(string name)` `{}`
`public void AppendChunk(string uid, byte[] buffer, long offset)` `{}`

WebMethod

```
public void FinishUpload(string uid, byte[] md5hash)
{}
```

2. Before you write the service code, add a reference to Atalasoft.dotImage, which is used to determine the number of pages that are in the submitted document.
3. Create an upload folder in the website's root to store the documents as they are uploaded.
4. Name the folder as Upload.
   Ensure that the ASPNET user has modify privileges.
5. Add the following configuration setting to the configuration tag at the bottom of the `web.config` file.
   `<appSettings> <add key = "UploadPath" value="Upload"/> </appSettings>`
6. Begin by creating the required using statements as shown here.

```
using System.IO;
using Atalasoft.Imaging;
using Atalasoft.Imaging.Codec;
using System.Configuration;
using System.Security.Cryptography;
using ImageDatabaseTableAdapters;
```

7. Add a helper method to assist in error handling as shown here.

```
public Exception CustomSoapException(string message)
{
return new SoapException(message, new
System.Xml.XmlQualifiedName("ImageUploadService"));
}
```

8. In the webservice constructor, define the upload path and ensure that it exists. Throw off an error if it does not.

```
string _uploadPath;
public ImageUploadService()
{
    // check that the upload folder exists
 string uploadConfigSetting =
 ConfigurationManager.AppSettings["UploadPath"].ToString();
    if (Path.IsPathRooted(uploadConfigSetting))
        _uploadPath = uploadConfigSetting;
    else
        _uploadPath = Server.MapPath(uploadConfigSetting);
    if (!Directory.Exists(_uploadPath))
        throw CustomSoapException("Upload Folder not found");
}
```

9. Create the StartUpload() method.
   The StartUpload() method initiates the upload process. It creates a unique GUID stored as both the primary key in the database and as the temporary file name. This is also where the client defines the name of the document being uploaded. This information is stored as a database record which is later updated with the image data once uploading is complete.
   StartUpload() returns the unique GUID string that will be used to identify each chunk.

```
public string StartUpload(string name)
{
    Guid guid = Guid.NewGuid();
    string uid = guid.ToString();
    using (Stream stream = File.Create(Path.Combine(_uploadPath, uid)))
    {
    }
    using (ImageDatabaseTableAdapter ta = new ImageDatabaseTableAdapter())
    {
        ta.CreateRecord(guid, name);
    }
    return uid;
}
```

10. Call AppendChunk() from the client until the entire document is uploaded.

```
public void AppendChunk(string uid, byte[] buffer, long offset)
{
    string filename = Path.Combine(_uploadPath, uid);
    if (File.Exists(filename))
    {
        // open a file stream and write the buffer.  Don't open with
 FileMode.Append because the transfer may wish to start at a different point
 using (FileStream fs = new FileStream(filename, FileMode.Open,
 FileAccess.ReadWrite, FileShare.Read))
        {
            fs.Seek(offset, SeekOrigin.Begin);
            fs.Write(buffer, 0, buffer.Length);
        }
    }
    else
    {
        throw CustomSoapException("Error Uploading File: cached file is missing");
    }
}
```

11. Implement the FinishUpload() method, which does the following:

- Verifies that the image data has not been corrupted during transfer by verifying a hash.
- Updates the image field in the database with image data.

Connect the capture client to the service

Before proceeding with this lesson, do the following:

- Save the ImageUploadService file.
- In the CaptureClient application completed in the previous part of this tutorial, add a reference to ImageUploadService just created. Name that connection DotImageWebServer.

1. Create a new button in the ToolStrip named tsbUpload, and double click it to create the event handler.

2. Create a private constant, _chunkSize, that defines the size of each chunk to be sent to the service:

   ```
   const int _chunkSize = 16 * 1024; //in bytes (16 KB)
   ```

   You need to provide a title for each image sent to the service. To do so, add a new Windows Form, InputForm. Use the button named listed here:

   Cancel btnCancel

   OK btnOK

   textboxtxtTitle

3. Create the Title property for use in accessing the title text.

```
public string Title
{
get { return txtTitle.Text; }
}
```

Creating the upload code

You are now ready to create the upload code. The code is used at the point where the image is saved to a stream, and then uploaded, chunk by chunk, to the service. When the process is complete, a hash is calculated and sent to the service to ensure that the image data on the server is identical to the image data on the client.

This demo saves the document as a multipage TIFF. Use a temporary file stream to store the data as it is uploaded. In the handler for the upload button CaptureClient's Form1), you need to add code.

1. Begin by adding the using the statement shown here.

   ```
   using System.Security.Cryptography;
   ```

2. Continue by adding the handler shown here.

```
private void tsbUpload_Click(object sender, EventArgs e)
{
    InputForm dialog = new InputForm();
    if (dialog.ShowDialog(this) == DialogResult.OK)
    {
        string tempPath = System.IO.Path.GetTempPath();
        DotImageWebServer.ImageUploadService service = new
 DotImageWebServer.ImageUploadService();
        service.Credentials = System.Net.CredentialCache.DefaultCredentials;
        string guid = service.StartUpload(dialog.Title);
        string filename = Path.Combine(tempPath, guid);
        byte[] hash = null;
        try
        {
            TiffEncoder encoder = new TiffEncoder(TiffCompression.Default);
            using (Stream stream = File.Create(filename))
            {
                documentViewer1.Save(stream, encoder);
                stream.Seek(0, SeekOrigin.Begin);

                //get hash
                MD5CryptoServiceProvider md5 = new MD5CryptoServiceProvider();
                hash = md5.ComputeHash(stream);
                stream.Seek(0, SeekOrigin.Begin);

                //read stream
 long streamLength = stream.Length;
```

```
                byte[] buffer = new byte[_chunkSize];
                int currentOffset = 0;
                int bytesRead = 0;
                do
                {
                    bytesRead += stream.Read(buffer, 0, _chunkSize);
                    service.AppendChunk(guid, buffer, currentOffset);
                    currentOffset = bytesRead;
                } while (bytesRead < streamLength);
            }
        }
        finally
        {
            if (guid != null)
            {
                service.FinishUpload(guid, hash);
                File.Delete(filename); //delete temp file
            }
        }
    }
}
```

Running the capture application

You can now run the Capture application. Open or scan a document. The document is submitted to the service and then stored to the SQL Express database.

Show progress while the document loads

In "Connect the Capture Client to the Web Service", you may have noticed the lack of feedback when the document is uploading to the server. In this lesson, you enhance the Capture application to show progress while a document is loading. To do so you will:

- Add a progress bar.
- Add a background worker thread enabling the main user interface to continue to respond to events and to show progress as each chunk in the document is uploaded.

1. From the Toolbox, add a StatusStrip control to the main form.
2. From the StatusStrip pull-down menu, add a progress bar and a status label.
3. Set the status label text to Ready.

Add the background worker thread

1. Add a BackgroundWorker component from the toolbox to the form.
2. In the BackgroundWorker, create event handlers for DoWork and ProgressChanged.
3. Set the WorkerReportsProgress property to true in the Property Grid.

Modify the Upload code to use background worker and report progress

1. Cut and paste all the code with the `if(dialog.ShowDialog ...)` to the backgroundWorker1_ DoWork event handler.
2. Click **Upload**.
   - Invoke the backgroundWorker1 RunAsyncMethod.

- Pass in the Title.

Your code should look similar to the following:

```
private void tsbUpload_Click(object sender, EventArgs e)
{
    InputForm dialog = new InputForm();
    if (dialog.ShowDialog(this) == DialogResult.OK)
    {
        backgroundWorker1.RunWorkerAsync(dialog.Title);
    }
}
```

Update the event handler

1. In the backgroundWorker1_DoWork event handler, change the dialog.Title to (string) e.Argument.
2. Add code to report progress. Start by adding this code at the beginning of the DoWork event handler.

   ```
   backgroundWorker1.ReportProgress(0, "Connecting to Web Service...");
   ```
3. Add the following code just before the documentViewer1.Save() method.

   ```
   backgroundWorker1.ReportProgress(0, "Encoding...");
   ```
4. Add the following code after the hash calculation and immediately before the while loop (after the declarations for reading the stream).

   ```
   backgroundWorker1.ReportProgress(0, "Uploading: 0 of " +
   Convert.ToInt32((double)buffer.Length / 1024) + " KB");
   ```
5. Add the following code after service.AppendChunk is called in the while loop.

   ```
   backgroundWorker1.ReportProgress(Convert.ToInt32((double)
   bytesRead / streamLength * 100), "Uploading: " + Convert.ToInt32
   (bytesRead / 1024) + " of " + Convert.ToInt32((double)
   streamLength / 1024) + " KB");
   ```
6. Add the following code at the end of this method.

   ```
   backgroundWorker1.ReportProgress(100, "Done");
   ```
7. Add the following code in the backgroundWorker1_ProgressChanged event handler to update the user interface with progress information.

   ```
   private void backgroundWorker1_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
   {
       toolStripProgressBar1.Value = e.ProgressPercentage;
       if (e.UserState != null)
           toolStripStatusLabel1.Text = e.UserState.ToString();
   }
   ```

Test your work

When testing the application, you should see the progress bar move while the document is uploading. The resulting document appears in the SQL Express Database.

View documents from a central database repository in a browser

In the previous tutorial, you built a capture client that submits documents to a centralized server via services. You built the database in SQL Express and created the data abstraction layer using the tools in Visual Studio 2012.

In this tutorial, the capture application and associated service adds data to the database. You need to develop a viewing application to navigate and view the documents in the database.

In contrast to the capture application that will be distributed to only a handful of clients with desktop scanners, the viewing application will be used by the entire enterprise. Creating a zero footprint viewing application reduces the deployment and maintenance costs associated with installing and supporting a client application.

Be sure that you have completed the previous tutorials before proceeding with this one.

This tutorial consists of the following lessons:

- Configure a Web siteto use Atalasoft DotImage controls
- Add a webThumbnailViewer and WebAnnotationsViewer to the form
- Use a Data-Bound drop-down list to navigate documents stored in the database
- Load documents from database into the viewer
- Add No-Postback navigation controls to the viewer

Configure a Web siteto use Atalasoft DotImage controls

In this lesson, you configure a Web site to use Atalasoft DotImage controls. This requires adding the ASP.NET WebForm controls to the project and configuring the ImageCache.

1. Give the ASP.NET user modify access to the folder.
2. Add the following XML to the appSetting section of the `web.config` to define where the ImageCache exists and to determine how long to keep the images in the cache.

```
<add key="AtalasoftWebControls_Cache" value="ImageCache/"/>
<add key="AtalasoftWebControls_CacheLifeTime" value="60"/>
```

Add a WebThumbnailViewer and WebAnnotationsViewer to the form

In this lesson, you add a WebThumbnailViewer and a WebAnnotationsViewer to the form.

Begin by dragging and dropping a WebThumbnailViewer and a WebAnnotationViewer from the Visual Studio Toolbox onto the Form.

Setting the ThumbnailViewer properties

1. Position the ThumbnailViewer to the left of the Annotation Viewer using tables.
2. Set the width of the Thumbnail Viewer to 150px and the width of the AnnotationViewer to 100%. Set the height of both controls to 500px.
   The controls should now be positioned and sized properly.

Connecting the viewer with the thumbnails

To connect the Viewer with the Thumbnails, type WebAnnotationViewer1 into the ViewerID property of the WebThumbnailViewer1. By setting that property, clicking a thumbnail automatically displays the associated image in the viewer.

The HTML code for the viewers should now look similar to the following.

```
< table > < tr > <td> <cc1:WebThumbnailViewer
ID="WebThumbnailViewer1" runat="server" Width="150px"
Height="500px" ViewerID="WebAnnotationViewer1" /> </td> <td
style="width:100%"> <cc2:WebAnnotationViewer
ID="WebAnnotationViewer1" runat="server" Height="500px"
Width="100%" /> </td> </ tr > </ table >
```

Use a Data-Bound drop-down to navigate documents stored in the database

1. Add a drop-down list to the page
2. Bind the list with all documents stored in the database.
3. In the drop-down list designer, configure the Data Source to use the SQL Database, the ImageDatabaseConnectionString, and an SQL statement that selects all from the ImageDatabase.

   Here is the SQL statement: `SELECT * FROM [ImageDatabase]`
4. Set the display field to Name and the value field to ImageId.
5. Select EnableAutoPostback to get a new document whenever the drop-down value is changed. Your design should look similar to that shown below.

   If you run the project, the document names are listed in the drop-down menu but no image is loaded into the viewer. That is the task of the next lesson, Load Documents from Database into the Web Viewer.

Load documents from database into the viewer

In this lesson, you load documents from a database into the viewer.

This lesson takes advantage of an Atalasoft DotImage feature that provides the ability to load images directly from a database using the DBImageSource class. This class requires a callback to handle the database access as efficiently as possible.

The requirements for database access include:

- A connection string callback
- Table name (from Clause)
- Image field
- Primary Key
- Optional order by clause

- Frame count (required for performance reasons)

1. The first step is to write a method that loads the image into the WebThumbnailViewer from DBImageSource, an ImageSource object you need to create. Add the following using statements at the beginning of the code followed by the Load() method.

```
using Atalasoft.Imaging.ImageSources;
using Atalasoft.Imaging.ImageSources.Data;
using System.Data.SqlClient;
private void LoadImage(string guid)
{
    DbSqlImageAccessor accessor = new DbSqlImageAccessor(new
 GetConnectionCallback(GetConn),
    false, "ImageData", "ImageID", "ImageDatabase", "ImageID='" + guid + "'",
    "DateTimeCreated", "PageCount");
    DbImageSource source = new DbImageSource(accessor);
    this.WebThumbnailViewer1.Open(source);
}
```

2. Create the static callback method for getting the connection for the preceding method to work. The code for this method is shown here.

```
private static IDbConnection GetConn()
{
    return new SqlConnection(ConfigurationManager.ConnectionStrings
    ["ImageDatabaseConnectionString"].ConnectionString);
}
```

3. Call the LoadImage() method when the drop-down changes. The selected index is databound.

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
    LoadImage(DropDownList1.SelectedValue.ToString());
    WebThumbnailViewer1.SelectedIndex = 0;
}
protected void DropDownList1_DataBound(object sender, EventArgs e)
{
    LoadImage(DropDownList1.SelectedValue.ToString());
    WebThumbnailViewer1.SelectedIndex = 0;
}
```

4. When the first page loads, the first document is shown in the viewer. You can use the drop-down to load a different document into the viewer.

Add No-Postback navigation controls to the viewer

In this lesson, you add No-Postback navigation controls to the viewer.

In the Capture application, you added navigation buttons to fit image to width, and to view full size. In this lesson, you do the same thing with the Web Viewer.

One possible approach is to create server buttons that postback to the server and change the properties. A better approach is to create an HTML button and use the Viewer's Javascript API to change the settings without posting back. This lesson takes the latter approach.

Begin by adding two HTML buttons, btnFitToWidth and btnViewFullSize to your form. You can choose to represent the button as an image or as text. Double-clicking on a button adds its event

handler in JavaScript. Interacting with the JavaScript is similar to working with server-side code, but all properties begin with set or get. Your JavaScript code should look like the following sample.

```
<script language="javascript" type="text/javascript">
// <!CDATA[
 function btnFitToWidth_onclick() {
   WebAnnotationViewer1.setAutoZoom(3);
}

function btnViewFullSize_onclick() {
   WebAnnotationViewer1.setAutoZoom(0);
   WebAnnotationViewer1.setZoom(1.0);
}

// &cd;>
</script>
```

Consult the Atalasoft DotImage documentation for the full JavaScript API reference.

Collaborate on documents using AJAX-enabled annotations

The previous three tutorials demonstrated how to create a client capture application that submits multipage document to a service interface to a database. An ASP.NET application navigates and views the documents from the SQL database repository.

This tutorial consists of the following lessons:

- Configure the controls to draw and view annotations
- Configure the application to draw annotations
- Configure the SQL database to store annotations
- Store and load annotations from the database

Configure the controls to draw and view annotations

As the viewer is already functioning at this point in the tutorial, adding annotations is straightforward.

1. Set the ShowAnnotations property of the WebThumbnailViewer1 to true.
2. Set the AutoLinkThumbnailViewer in WebAnnotationsViwer1 to true.
3. Add an HTML button, btnStickyNote, for sticky note annotations.
4. Add an HTML button, btnHighlighter, for a highlighter annotation.

Configure the application to draw annotations

In this lesson, configure the application that draws annotations.

1. Add these using statements to the default.aspx.cs file.

```
using Atalasoft.Annotate;
using Atalasoft.Annotate.Formatters;
using System.Drawing;
using ImageDatabaseTableAdapters;
```

2. In the Page_Load event handler, create default annotations for StickyNote and Highlighter as shown here.

```
protected void Page_Load(object sender, EventArgs e)
{
    //add default annotations
    TextData stickynote = new TextData("Double Click to Edit");
    stickynote.Font.Size = 24;
    stickynote.Fill = new AnnotationBrush (Color.Yellow);
    stickynote.Name = "StickyNote";
    WebAnnotationViewer1.Annotations.DefaultAnnotations.Add(stickynote);

    RectangleData highlighter = new RectangleData();
    highlighter.Translucent = true;
    highlighter.Fill = new AnnotationBrush (Color.FromArgb(127, Color.Green));
    highlighter.Name = "Highlighter";
    WebAnnotationViewer1.Annotations.DefaultAnnotations.Add(highlighter);
}
```

3. In the JavaScript code:
   a. Change the InteractMode to Author.
   b. Create the annotations.
   c. In the OnAnnotationCreated event, change the InteractMode back to Modify so that normal mouse operations once again function.

   Here is a sample of these three functions.

```
function OnAnnotationCreated(e)
{
    WebAnnotationViewer1.setInteractMode(AnnotationInteractMode.Modify);
    WebAnnotationViewer1.AnnotationCreated = function(){};
}

function btnStickyNote_onclick() {
    WebAnnotationViewer1.setInteractMode(AnnotationInteractMode.Author);
    var ann = WebAnnotationViewer1.CreateAnnotation('TextData', 'StickyNote');
    WebAnnotationViewer1.AnnotationCreated = OnAnnotationCreated;
}

function btnHighlighter_onclick() {
    WebAnnotationViewer1.setInteractMode(AnnotationInteractMode.Author);
    var ann = WebAnnotationViewer1.CreateAnnotation('RectangleData',
 'Highlighter');
    WebAnnotationViewer1.AnnotationCreated = OnAnnotationCreated;
}
```

   After you add the preceding code, clicking on the StickyNote or Highlighter draws annotations onto the image. You now have annotations support, although the annotations lack persistence.

Configure the SQL database to store annotations

In this lesson you configure the SQL database to store annotations.

1. Add an annotations field to the database. This allows annotations to be stored separately form the image data as XML. To do so, add a field called Annotations with type varbinary(MAX).

2. Add a query to the ImageDatabaseTableAdapter called GetAnnotations using the following SQL:

   ```
   SELECT Annotations FROM ImageDatabase WHERE ImageID = @ImageID
   ```

3. Add a query called UpdateAnnotations using the following SQL:

   ```
   UPDATE ImageDatabase SET Annotations = @Annotations WHERE (ImageID =
   @Original_ImageID)
   ```

Store and load annotations from the database

In this lesson, you store and load annotations from the database.

Store the annotations

Use this procedure to store annotations in the database.

1. Open the toolbox and drag a button (which is called a server button)onto your form and name it btnSave.
2. Double-click the button to create the event handler.
3. Add the following code to the handler to save the annotations as XMP and store in the database using the query created in Configuring the SQL Database to Store Annotations.

```
protected void btnSave_Click(object sender, EventArgs e)
{
    XmpFormatter formatter = new XmpFormatter();
    byte[] annots = WebAnnotationViewer1.Annotations.Save(formatter);

    using (ImageDatabaseTableAdapter ta = new ImageDatabaseTableAdapter())
    {
        ta.UpdateAnnotations(annots, new Guid(DropDownList1.SelectedValue));
    }
}
```

Load the annotations

Add code to load the annotations from the database. Do this after the image is loaded into the Viewer.

1. Clear the annotations from the previous image by invoking the ClearAnnotations() method of WebAnnotateViewer.
2. Get the annotations using the Query previously added.

```
private void LoadImage(string guid)
{
    DbSqlImageAccessor accessor = new DbSqlImageAccessor(new
 GetConnectionCallback(GetConn), false, "ImageData", "ImageID", "ImageDatabase",
 "ImageID='" + guid + "'", "DateTimeCreated", "PageCount");
    DbImageSource source = new DbImageSource(accessor);
    this.WebAnnotationViewer1.ClearAnnotations();
    this.WebThumbnailViewer1.Open(source);
    //get annotation data
 byte[] annotations = null;
    using (ImageDatabaseTableAdapter ta = new ImageDatabaseTableAdapter())
    {
        annotations = (byte[])ta.GetAnnotations(new Guid(guid));
    }
    if (annotations != null)
    {
```

```
        using (MemoryStream ms = new MemoryStream(annotations))
        {
            WebAnnotationViewer1.LoadAnnotationData(ms);
        }
    }
}
```

Now the application loads the document with annotations that have persisted in the database. More annotations can be created and saved back to the database.

Chapter 3

Program with Atalasoft DotImage

Atalasoft DotImage brings a full-featured advanced imaging solution to your desktop or thin client application. With a collection of controls for ASP.NET and Windows, Atalasoft DotImage offers a light and powerful solution that matches your imaging requirements. Licensing is straightforward and runtime royalty free on the desktop. All Atalasoft DotImage and related assemblies are available as managed components and are natively built as .NET assemblies. You can choose from three distinct editions of Atalasoft DotImage as well as many add-on modules.

Getting started

Core Atalasoft DotImage editions

Atalasoft DotImage Photo

Basic color and grayscale image processing.

DotImage Photo Pro

Controls for viewing, editing, and printing images.

Color and grayscale image processing for the photographic and pre-press industry.

Support for EXIF/IPTC/XMP Metadata, Advanced Color Management, regional processing, automatic color adjustments, and RAW image support.

AJAX ASP.NET Server control for thin client image viewing and editing.

Atalasoft DotImage document imaging

Professional document imaging that includes all the features of Atalasoft DotImage Photo Pro plus:

- Fast scale to gray display
- TWAIN scanning
- Full-featured annotations SDK
- Auto-deskew
- Basic document clean-up
- Thresholding
- Multipage TIFF support
- Multipage PDF export
- Full features annotations support

AJAX ASP.NET image viewer, and thumbnail viewer for viewing documents on the without client postbacks.

Add-on modules

Atalasoft DotImage supports a number of add-on modules as detailed in the table that follows.

BarcodeReader
Read 1D and 2D barcodes from any image.

Barcode Writer
Write 1D and 2D barcodes into images.

PDF Reader
Read any PDF document as a raster image without Adobe Acrobat Reader.

JPEG2000 Codec
Read and Write JPEG2000 images. Used to add JPEG2000 support for PDF output.

JBIG2 Codec
Read and Write JPEG2000 images. Used to add JBIG2 support for PDF output

OCR module
Provides a standard interface for OCR engines and built-in translators to generate output files from the results of OCR.
Includes the open source Tesseract Engine.

GlyphReader OCR engine
Highly accurate lexicon-based OCR engine for European characters.

I.R.I.S. OCR
Interface to the I.R.I.S. iDRS multi-language ICR/OCR engine.

Searchable PDF module
Create searchable PDF files from any Atalasoft DotImage OCR engine.

Tesseract OCR engine
Interface to the open source Tesseract OCR engine.

Advanced document cleanup
Provides advanced document cleanup routines such as:

- Border removal
- Blob and hole punch removal
- Auto binarization
- Line removal
- Blank page detection
- Auto text inversion
- Auto negate
- Speck removal

Advanced photo effects

Advanced photo manipulation algorithms for automatically adjusting brightness and contrast of photos, and applying automatic effects to increase the overall quality of photos taken from both low resolution camera phones and high end professional cameras.

DICOM Codec

Read DICOM images and access image metadata.

DWG Codec

Read images from DWG and DXF CAD files.

Atalasoft DotImage ISIS

Capture images from ISIS-compatible scanners.

.NET assemblies

The following table lists the .NET assemblies provided with Atalasoft DotImage. These assemblies are compatible with .NET Framework and .NET 5 unless otherwise noted. For a list of .NET versions supported by Atalasoft DotImage, see the *Atalasoft DotImage Technical Specifications*.

Assembly	DLL	Notes
ASP.NET AJAX WebForm Controls	`Atalasoft.dotImage.WebControls.dll`	This library contains the pre-built handlers for the WebDocumentViewer, WebAnnotationViewer and WebCapture.
Advanced Document Cleanup	`Atalasoft.dotImage.AdvancedDocClean.dll`	Use AdvancedDocClean's filters and helper functions to perform advanced document cleanup. HolePunchRemoval, Despeckle, DynamicThreshold, and Deskew help clean any scanned paper document for any downstream recognition (barcode, ocr, omr, etc.).
Advanced Photo Effects	`Atalasoft.dotImage.AdvancedPhotoEffects.dll`	AdvancedPhotoEffects contains many programmatic filters designed to edit photographic images. Similar to the filters in any modern photo manipulation application, the library contains filters such as GaussianBlurCommand, SharpenCommand, and commands to perform color correction.
Atalasoft DotImage Class Library	`Atalasoft.dotImage.dll`	Start with this library in your applications that use images to save developer time and effort.

Assembly	DLL	Notes
Atalasoft DotImage ISIS	`Atalasoft.dotImage.Isis.dll`	The ISIS library adds controls and support for interacting with ISIS device drivers allowing for automated and custom scanning applications.
Atalasoft dotImage Office decoder	`Atalasoft.dotImage.Office.dll`	Atalasoft DotImage Office files decoder.
Atalasoft dotImage Pdf Document Model	`Atalasoft.PdfDoc.dll`	Atalasoft DotImage Pdf Document Model library.
Atalasoft dotImage PdfDoc Bridge	`Atalasoft.dotImage.PdfDoc.Bridge.dll`	Atalasoft DotImage and DotPdf integration helper functions.
Atalasoft dotImage Tesseract 3 OCR support.	`Atalasoft.dotImage.Ocr.Tesseract3.dll`	Atalasoft DotImage libraries for Google's Tesseract V3 engine.
Barcode Reader	`Atalasoft.dotImage.Barcoding.Reading.dll`	Atalasoft DotImage's BarcodeEngine is contained within this package. It provides functions and settings to interact with and extract bar code data from scanned images.
Barcode Writer	`Atalasoft.dotImage.Barcoding.Writing.dll`	BarcodeWriter is a helper class that can overlay scanable bar codes onto digital documents.
DICOM Codec	`Atalasoft.dotImage.Dicom.dll`	Use this library to add the DicomDecoder to Atalasoft DotImage's RegisteredDecoders collection to add support for the Dicom medical device format.
DWG Codec	`Atalasoft.dotImage.Dwg.dll`	Use this library to add the DwgDecoder to Atalasoft DotImage's RegisteredDecoders collection to add support for AutoCAD's DWG CAD drawing files.
DotTWAIN	`Atalasoft.DotTwain.dll`	Use this library to enable interaction with TWAIN drivers for scanners and cameras to capture images directly into custom applications.
Glyph Reader Engine	`Atalasoft.dotImage.Ocr.GlyphReader.dll`	Use this library with the Atalasoft DotImage OCR library to add the GlyphReader engine to usable engines.

Assembly	DLL	Notes
HEIF Codec	`Atalasoft.DotImage.Heif.dll`	Use this library to add the HeifDecoder to Atalasoft DotImage's RegisteredDecoders collection to add support for HEIF files.
Internal	`Atalasoft.dotImage.Lib.dll`	Lib contains a majority of the additional functionality for the Atalasoft DotImage toolkit.
JBIG2 Codec	`Atalasoft.dotImage.Jbig2.dll`	Use this library to add the JB2Decoder to Atalasoft DotImage's RegisteredDecoders collection to add support for Jbig2 compressed files.
JPEG2000 Codec	`Atalasoft.dotImage.Jpeg2000.dll`	Use this library to add the Jpeg2000Decoder to Atalasoft DotImage's RegisteredDecoders collection to add support for Jpeg2000 compressed files.
OCR	`Atalasoft.dotImage.Ocr.dll`	Use this library to add Optical Character Recognition (OCR) to convert scanned images into their contained text.
PDF Image Encoder	`Atalasoft.dotImage.Pdf.dll`	Atalasoft DotImage's DotPdf library helps create programmatic PDF and allows editing of existing PDF files.
PDF Reader	`Atalasoft.dotImage.PdfReader.dll`	Use this library to add the PdfDecoder to Atalasoft DotImage's RegisteredDecoders collection to add support for Adobe's PDF files.
RAW Image Codec	`Atalasoft.dotImage.Raw.dll`	Use this library to add the RawDecoder to Atalasoft DotImage's RegisteredDecoders collection to add support for RAW camera files.
WPF Controls	`Atalasoft.dotImage.Wpf.dll`	Atalasoft DotImage viewers for Windows Presentation Foundation apps.

Assembly	DLL	Notes
Windows Forms Controls	`Atalasoft.dotImage.WinControls.dll`	Atalasoft DotImage viewers for WinForms. AnnotateViewer, DocumentViewer, and ImageViewer allow for displaying all of our image formats in a windows application.

Create images

Atalasoft DotImage has one class, AtalaImage, which is used for representing images. An AtalaImage is an in-memory representation of a raster (or pixel-based) image. An image is defined by several characteristics as listed in the table below.

Attribute	Description
Width	The width of the image is in pixels.
Height	The height of the image is in pixels.
Resolution	The number of pixels per unit of measure, e.g. pixels per inch.
Pixel Format (or Depth)	The way the pixels represent color.

Create an image from scratch

You can create AtalaImages from the following:

- From scratch
- From other AtalaImages
- From streams

AtalaImages can represent black and white, gray, and color images in a number of different formats. They can be converted to and from any format.

Use the following code to create a new image from scratch:

```
AtalaImage image = new AtalaImage(width, height, PixelFormat.Pixel24bppBgr);
```

This code creates a new blank image of the given width and height in 24 bit per pixel color using the RGB color model. The image data is set to zeroes, which in this case creates a black image.

Copying an image

An image can be copied by using the Clone() method.

```
AtalaImage newImage = (AtalaImage)oldImage.Clone();
```

This code makes a new copy of the old image. Changes to newImage are not seen in oldImage.

Open images

There are several ways to open images in Atalasoft DotImage. For example, use a path to an image file as shown below.

```
AtalaImage image = new AtalaImage(pathToAnImage);
```

There is a lot that happens under the hood: first the path is opened into a Stream object, then Atalasoft DotImage checks to see if there is an ImageDecoder installed that supports this image file format. If there is, that ImageDecoder is chosen to turn the file contents into an AtalaImage. The path also can be an URL.

If there is no available decoder, this method fails and throws an ImageReadException. If you need to check to see if there is an installed decoder for a file you can do the following.

C#

```
public bool IsImageFormatSupported(string path)
{
            try {
                        ImageInfo info = RegisteredDecoders.GetImageInfo(path);
                        return true;
            }
            catch (ImageReadException) {
                        return false;
            }
}
```

RegisteredDecoders is initialized with a large suite of ImageDecoders, including ImagesDecoders for Windows Bitmap, JPEG, TIFF, and eleven other formats. You can add or remove items from this collection. If your installation of Atalasoft DotImage includes add-on modules with ImageDecoders (such as the PDF Reader module), you need to add the decoder to RegisteredDecoders yourself. See the Atalasoft DotImage Class Library.

C#

You can also work with an ImageDecoder directly. For example, if you were opening only TIFF files, you could use the TiffDecoder object yourself:

```
public AtalaImage OpenTiff(string path)
{
            using (FileStream stm = new FileStream(path, FileMode.Open,
 FileAccess.Read, FileShare.Read)) {
                        TiffDecoder decoder = new TiffDecoder();
                        if (!decoder.IsValidFormat(stm))
                                    throw new Exception("not a TIFF");
                        AtalaImage image = decoder.Read(stm, null);
                        return image;
            }
```

```
}
```

This opens a TIFF file and return an AtalaImage for the first page. ImageDecoder.Read() includes a second argument, ProgressEventHandler. If a non-null ProgressEventHandler is passed in, client code can receive information on progress in loading an image.

Several image formats include multiple pages in one file. In this case, the ImageDecoder includes methods that have a frame index argument. Passing in a zero based value returns an AtalaImage for that frame.

Save images

Saving images is not very different from reading images. Saving an image is done through an ImageEncoder object. Select an ImageEncoder for the file format that you would like, then use it to save the image. This can be done with the convenience method, AtalaImage.Save(), or through the ImageEncoder itself. For example, to save an image as a JPEG, you could do the following:

C#

```
AtalaImage image = GenerateImage(); // gets or creates an image
image.Save("mynewfile.jpg", new JpegEncoder(), null);
```

Use a ProgressEventHandler to monitor progress.

To use the ImageEncoder directly, you need to use a stream for the output first. The following method saves an image as a JPEG using the encoder directly:

C#

```
public void SaveAsJpeg(string path, AtalaImage image)
{
            JpegEncoder encoder = new JpegEncoder();
            AtalaImage saveImage = image;
            if (!encoder.IsPixelFormatSupported(image.PixelFormat)) {
                        saveImage =
 image.GetChangedPixelFormat(encoder.SupportedPixelFormats[0]); // arbitrary choice
            }
            FileStream stm = null;
            try {
                        stm = new FileStream(path, FileMode.Create);
                        encoder.Save(stm, image, null);
            }
            finally {
                        if (stm != null)
                                    stm.Close();
                        if (saveImage != image)
                                    saveImage.Dispose();
            }
}
```

In addition to saving the image, this method checks the ImageEncoder to determine if it supports the PixelFormat of the image. If not, it creates a new image in a supported PixelFormat.

In addition to saving single images, many ImageEncoders include options for setting compression, adding metadata, or multiple pages. See the object reference for each ImageEncoder to see the specific features available.

Process and clean up images

Atalasoft DotImage features the ability to perform operations on the contents of images to create new image. The basis for this is the ImageCommand object. To process an image, you create an object derived from ImageCommand, modify its properties, if needed, and then call the ImageCommand's Apply() method. For example, to invert an image, you can use the InvertCommand:

C#

```
  AtalaImage image = GenerateImage();
  InvertCommand command = new InvertCommand();
  ImageResults results = command.Apply(image);
```

ImageResults contains the results of performing the command. This includes a property named Image which is the resulting image. It also includes a boolean, IsImageSourceImage, which indicates whether the ImageResults.Image property is the same as the original source image. If IsImageSourceImage is true, it indicates that the Image property of the ImageResults object is the same as the source image. If IsImageSourceImage is false, it indicates that the Image property of the ImageResults object is a new image. You can check this ahead of time by checking ImageCommand.InPlaceProcessing. If InPlaceProcessing is true, the command operates on the source image itself.

Several ImageCommands provide more information than is in an ImageResults object. These commands return a subclass of ImageResults to expose the extra information.

Dispose objects

Because images take up a great deal of memory, it's a good idea to dispose AtalaImage objects after you are done with them. For example, if you creating a brand new image by processing and do not care about the old image anymore, you might want to follow this pattern:

C#

```
public AtalaImage Process(AtalaImage image, ImageCommand command)
{
            ImageResults results = command.Apply(image);
            if (!results.IsImageSourceImage)
                        image.Dispose();
            return results.Image;
}
```

Image commands may not operate on all image formats. You can call ImageCommand.IsPixelFormatSupported() to determine if a command operates natively on an particular PixelFormat. In addition, many commands can be induced to operate on any PixelFormat by setting the property ApplyToAnyPixelFormat to true. When ApplyToAnyPixelFormat is true, the source image is used to create a new image in an appropriate PixelFormat before performing the command. While all current Atalasoft DotImage ImageCommands allow this, it is possible for a command to reject this process by returning false in the property CanApplyToAnyPixelFormat.

You can get a list of all PixelFormats that an ImageCommand supports via the SupportedPixelFormats property.

Generally speaking, ImageCommands on their own are not thread-safe. That is, if you use the same ImageCommand object in two or more threads concurrently, the results are unpredictable. It is recommended instead that you use an ImageCommand for each thread. Generally speaking, it is

very cheap to make ImageCommand objects. The constructors typically do very light weight work and the commands themselves consume very little memory.

ImageCommand objects

Atalasoft DotImage contains more than one hundred ImageCommand objects. They are broken down into functional groups as listed in the table that follows.

Object	Description
Channels	Commands that operate on images with multiple components, like color images
Document	Commands that are geared for scanned documents
Effects	Perform visual effects on images like mosaic or beveling
Filters	Perform mathematical filtering like high or low pass filtering
Threading	Commands that can make other commands operate in a multithreaded environment to improve performance
Transforms	Perform coordinate transforms or depth transforms like rotate or ripple

Image controls for Windows Forms applications

Atalasoft DotImage includes controls and components to add imaging GUI functionality to Windows Forms applications in the Atalasoft.dotImage.WinControls assembly. The following components are included.

Image viewing

The following images are all displayed in a viewer.

Component	Use
WorkspaceViewer	Displaying, scrolling, and interacting with AtalaImage objects and applying commands
ImageViewer	Displaying and scrolling AtalaImage objects
BitmapViewer	Displaying .NET Bitmaps and Metafiles
Magnifier	Displaying a magnified, zoomed view of a small section of the image below the mouse pointer

Thumbnail viewing

Component	Use
ThumbnailView	Display a list of thumbnail images in a single control.
FolderThumbnailView	Display a list of thumbnail images from a folder on the file system in a single control.

RubberBanding

Component	Use
RectangleRubberBand	Drawing a rectangle rubber band (used for defining rectangular regions and drawing rectangles on an image)
EllipseRubberBand	Drawing an elliptical rubber band (used for defining an elliptical region and drawing ellipses on an image)
LineRubberBand	Drawing lines on an image
RectangleSelection	Implementing features such as resizing and animation (derives from RectangleRubberBand)

Printing

Component	Use
ImagePrintDocument	Printing single or multipage images directly to a printer
ImageCompositePrintDocument	Printing multiple images as a photo composite at standard sizes and layouts.

Dialogs

Component	Use
OpenImageFileDialog	Loading an image from the file system, optionally showing a preview thumbnail

Image controls for web applications

Atalasoft DotImage provides AJAX-enabled imaging controls for ASP.NET that you can use in web applications. The controls can be used for image viewing and editing, thumbnails, and annotation. You can customize these controls to meet the requirements of your applications. Add the controls by dragging and dropping them on your WebForm.

For all versions of .NET, the Web Document Viewer (WDV) is recommended. The WDV is a JavaScript-based image viewing control that communicates directly with a WebDocumentRequestHandler on the server side. You do not need to create a traditional WebServerControl back end.

The following legacy controls are available for .NET Framework. These are not supported by .NET 5, and you should use the WDV instead.

WebImageViewer

The AJAX enabled WebImageViewer is used to view and edit images in a browser. This control works by loading tiled sections of an image "On-Demand" when the relevant section of an image is panned or zoomed into view, converting each tile to a browser friendly format on the fly. It is especially optimized to view tiled and stripped TIFF's as it selectively decodes tiles from the source image as it streams them to the client. Perfect for viewing large TIFF images, such as mechanical CAD drawings, or documents scanned at a high resolution as the bandwidth requirements for the

client are minimal. Also perfect for a universal document and image viewer when combined with the other supported codecs such as PDF, JPEG2000, and the other built-in formats.

This control is also useful for processing images interactively by tying a button to a server method. The result is that an image updates without a page postback. The rubberband selection feature can be used for cropping an image, or selecting an area of an image.

WebImageViewer provides the following features:

- Server-Side image viewing and processing with no client dependencies or installs
- Scale images with optional high-speed Scale-to-Gray display only available when used with Atalasoft DotImage Document Imaging
- On-demand loading of tiles as image is scrolled or panned
- A variety of mouse tools that zoom, pan, zoom to area, select, or center the image without posting back the entire page
- A rubberband for selecting an area of the image for navigation, cropping, or basic annotating
- Access to the full power of Atalasoft DotImage's Image Processing, Document Imaging, Metadata, Color Management, and other imaging functionality
- Choice of browser output format as JPEG, PNG, or GIF
- The ability to open images from HTTP, HTTPS, FTP, or from the local server
- The ability to print images by streaming as a PDF, or by using the JavaScript Print() method
- Support for viewing TIFF FAX images with different X and Y resolutions
- Very low bandwidth required to view very large images
- Access nearly all functionality through JavaScript with remote invoke methods, requiring no postback
- Very efficiently viewing of TIFF's that are saved as tiles or in strips
- Viewing of PDF's with the PDF Reader Module
- A "web" version of the Windows Form WorkspaceViewer Control
- Supports dragging and dropping controls onto an ASP.NET WebForm for design time

WebThumbnailViewer

The AJAX enabled WebThumbnailViewer is used to view multiple thumbnails from images stored in a multipage document such as a TIFF or PDF, or can be used to view thumbnail images from a folder. This control works by loading thumbnails "On-Demand" reducing bandwidth requirements and enabling the control to view thousands of thumbnails that are loaded only when scrolling in view. This control can be tied with the WebImageViewer for an advanced document image viewer.

WebThumbnail viewer provides the following features:

- On-demand loading of thumbnails as control is scrolled or panned
- Automatic synchronization of a WebImageViewer to show the selected thumb without any code
- The ability to place custom captions on each thumbnail with automatic placement of the page number and filename
- Select one or multiple Thumbs
- Requires very little bandwidth to view multipage images
- Access nearly all functionality through JavaScript and remote invoke methods, requiring no postback
- Supports the "web" version of the Windows Form's ThumbnailViewer Control

- Supports dragging and dropping controls onto an ASP.NET WebForm for design time support

WebAnnotationViewer

The AJAX enabled WebAnnotationViewer is used to annotate images in a browser. As WebAnnotationViewer is based on WebImageViewer, you get all of WebImageViewerfeatures (panning, zooming, tiling, and so forth), plus the ability to create, delete, view, and modify annotations without a postback.

WebAnnotationViewer uses the same rendering and data engine as Atalasoft DotImage Annotations. Therefore annotations can be created on the and viewed in a WinForms application and vice-versa. This control is useful for marking up images on the either in a collaborative environment or in an online image workflow application.

WebAnnotationViewer provides the following features:

- Access to the full power of Atalasoft DotImage's Annotations component for creating, loading, and saving annotations
- Users can create annotations with the mouse
- Users can move or resize annotations with the mouse
- User can edit Text Annotations in the browser
- Saved annotations are interoperable with the WinForms version of Atalasoft DotImage Annotations
- Access nearly all functionality through JavaScript and remote invoke methods, requiring no postback
- The "web" version of the Windows Form's AnnotationViewer Control
- Supports dragging and dropping controls onto an ASP.NET WebForm for design time support

Use WebAnnotationViewer to create:

- A thin client document viewer to view mechanical drawings from a database
- A based FAX image viewing and markup application
- A based QA application for manual review comparing original and new scanned images side-by-side
- A browser based annotations application for marking up documents without affecting the underlying image data
- An online image editor for an digital photo gallery

TWAIN scanning

DotTwain is a Managed .NET Windows Form component with fast capturing of digital images from scanners, cameras, and other devices supporting TWAIN.

DotTwain:

- Utilizes the improvements of the TWAIN 2.4 specifications for ultra fast scanning.
- Works with a .NET Bitmap object and has no dependencies other then the .NET Framework.

Supported Features

DotTwain:

- Provides full context-sensitive help and documentation.

- Support object-oriented design as it is a TWAIN extension to the .NET Framework.
- Supports the creation of a custom interface to scan images, or the use of the default TWAIN interface.
- Supports automatic document feeders, with full control of the feeder operations.
- Supports duplex scanning.
- Provides file system support for navigating the internal device memory.
- Supports direct acquisition of images to file utilizing device-supported compression algorithms.
- Provides frame support, allowing you to select a region of the scanning bed to be acquired.
- Supports uploading a scanned image to a server with the built in HTTP Post class.
- Provides simplified automatic capture allowing a device to quickly capture multiple images for later acquisition.
- Provides Imprinter/Endorser methods allowing a text string ito be stamped onto the acquired image during the acquisition.
- Provides Barcode/Patchcode classes to access bar codeand patchcode recognition abilities of the device
- Provides a low-level class allowing very fine control over the scanning session for advanced users
- Is extensible allowing for custom driver capabilities access
- Supports image datasets enabling acquisition of specific images from a camera in a single operation
- Quickly determines the capabilities, compression modes, frame sizes, resolutions and more, that are supported by a device
- Provides more than 80 properties and 50 methods to give you total control over the image acquisition

DotTwain has been tested on many industry leading document scanners including those from Kodak, Fujitsu, Panasonic, Canon, Visioneer, Xerox, and Bell & Howell.

Imaging

Dithering

The DitherCommand gives even finer control over the dithering process. The command includes the Floyd Steinberg and Ordered Dithering methods and allows for the customization of each.

When converting an image from an RGB or RGBA format, with many millions of possible colors, to a colormapped format, such as 8-bit or 1-bit format where there are only 256 or 2 colors available, there is a loss of color information. As a result, there is likely to be a difference (or “error”) for every pixel color in the original image and the corresponding pixel color in the new image.

There are two ways to deal with this error:

- Ignore it
- Push a portion of the error onto each of the surrounding pixels

This latter method is called “dithering” or “error diffusion”.

By default, when converting to a colormapped format, Atalasoft DotImage uses the Floyd Steinberg dithering algorithm to spread out color errors as defined in the AtalaPixelFormatChanger. The result is preferable to ignoring the error totally.

Because of the way we perceive colors, as an error spreads over an area of pixels, that area tends to resemble the same area in the original image even though there may be significant color errors on each individual pixel. This effect works much like an ink jet printer--many dots of just a few colors when placed close to one another look like a completely different color to the eye.

ℹ Dithering works best for photographic images.

You can control how much dithering takes place with the DitherErrorMax property. Values range from 0 to 127.

A value of 0 is equivalent to ignoring the color error completely. Images generated this way tend to have large areas of solid color and the transitions between these colors are harsh. The results can be dramatic, but it is not a good idea to make this your default.

As you increase the DitherErrorMax value, the amount of error that can be spread to other pixels increases and the resulting images pick up subtle shading and smoother color transitions.

Compression

In a raw state, images can occupy a rather large amount of memory both in RAM and in storage. Image compression reduces the storage space required by an image and the bandwidth needed when streaming that image across a network.

Types

- Lossless: When lossless data is decompressed, the resulting image is identical to the original
- Lossy: Lossy compression algorithms result in loss of data. The decompressed image is not the same as the original.

Methods

JPEG compression

JPEG compression is the format of choice for photographic color and grayscale images. JPEG uses a non-linear lossy compression that can achieve high compression ratios. It takes advantage of the fact that human vision is more sensitive to changes in brightness than in color.

JPEG compression performs well on smooth gradients with gradual changes in color. In images that contain sharp lines and text, JPEG compression performs less well and "artifacts" can appear that are visible to the human eye.

Saving a JPEG image always results in some loss in quality and/or image data. Repeatedly decompressing and recompressing an image with JPEG compression continually degrades the image.

ℹ TIFF images can be saved with JPEG compression.

Deflate/PNG compression

Deflate compression is a lossless scheme used in standard ZIP files. In imaging, PNG compression is most commonly used in the PNG format (pronounced "Ping"). Deflate compression works well with blocks of consistent data, and not as well with gradients. As a

result, PNG's are good choice whenever data preservation is critical. For example, it is a good choice when compressing data for archival purposes and when compressing logos, graphics, and images within blocks of text.

Atalasoft DotImage also supports the Deflate compression in TIFF images.

CCIT Group 4 / Group 3 compression

CCIT compression is lossless. It is most commonly used for compressing binary images as it was originally designed for FAX images. CCIT is good at compressing binary data in black and white images, and is the compression algorithm of choice for document images.

CCIT Group 3 is an older standard and very similar to Group 4.

CCIT compression is available in the TIFF image format.

LZW compression

LZW compression is similar to Deflate but was patented by Terry Welch and his patent was enforced by Unisys. Unisys's US patent on the LZW algorithm expired in 2003 and for some other countries in 2004. LZW compression has gained wide use in the GIF format developed by CompuServe. LZW is also common in TIFF images and like Deflate is always lossless. LZW is slightly faster than Deflate but generally does not produce the same degree of compression.

Huffman compression

Huffman is lossless compression that relies on identifying repetitive patterns in the data of a file. CCIT Huffman compression is 1-bit TIFF's support Huffman compression.

RLE compression

Run Length Encoding (RLE) is a simple lossless compression algorithm that works by representing a sequence of identical values with a counter and a value. RLE works well when horizontal lines in an image are consistent, but for some images RLE can increase the file size.

The 4-bit or 8-bit BMP, and CCIT RLE for 1-bit TIFF formats all support RLE compression

Wavelet / JPEG2000 compression

Wavelet compression is a newer compression technology that is most commonly used with the JPEG2000 format. Wavelet compression can yield compression ratios that are greater than JPEG at the expense of speed.

Wavelet compression is not as widely supported as JPEG and does not always yield a significantly better compression ratio then standard JPEG.

Wavelet compression is slower then JPEG compression.

Asynchronous image processing

Asynchronous image processing can be advantageous because of the large amount of processing power required to process, load, and save images. Programs that do not use asynchronous processing may be rendered unresponsive due to the processing power requirements of image processing.

For this reason, Atalasoft DotImage provides strong support for asynchronous image processing. Atalasoft DotImage can process images asynchronously using the Workspace object with no additional coding required.

By setting the Asynchronous property in the Workspace or WorkspaceViewer object to true, all image operations including opening, saving, and applying commands are performed in a separate thread.

Process queue

When an image is processed asynchronously, there are special considerations.

For example, a single AtalaImage object cannot perform two operations at once. Performing multiple operations at the same time yields unpredictable results. When opening, processing, or saving an image, the operation is entered into the ProcessQueue. This specialized collection ensures that the operations are performed in sequence. Each call to Open, ApplyCommand, or Save, adds the operation to the queue, and the operation does not start until all of the items ahead of the newly queued item are complete.

The queue can be accessed at any time to determine the current state of an operation, or to add or remove an item from the queue.

When working with a Workspace in Asynchronous mode, any queries to the current AtalaImage object yields the current state of the object. Suppose you need to check the number of colors using the AtalaImage.CountColors() method after loading an image. To do so, you must run synchronously, or else handle the ProcessComplete event and wait until that event is fired before obtaining information about the processed image.

ThumbnailView control

The ThumbnailView control utilizes asynchronous processing by default. It loads thumbnail images in a background thread pool. The number of concurrent worker threads can be controlled by the NumWorkerThreads property.

ℹ This property defaults to 3.

Resampling

In signal processing, the act of resizing a stream of data is known as resampling. This means that you take measurements of the data source at specific intervals in such a way that you can reproduce the original signal with a different amount of data, within the constraints of your system.

In image processing, one tries to resize a two dimensional image in such a way that the resized image resembles the original as much as possible, given a limited amount of time and memory. More specifically,

- When reducing images, the goal is to preserve the character of the source data, but with fewer points.
- When enlarging images, the goal is to invent data to fill in the holes where there is no source data.

In the examples that follow, assume there are Ns samples in the source and there are Nd samples in the destination, in one dimension. Two dimensional resampling is a natural extension of the one dimensional process.

ShowNearest neighbor

The easiest way to resample anything is to pick every Floor(Ns/Nd)th sample out of the source. Ex. : if Ns = 10 and Nd = 5 (you want your output data to be 1/2 the size of the source), you will pick every

2nd data point from the source. The math is simple and so, this is a very quick operation. As with many processes that are both quick and simple, the results are not ideal. This method ignores too much of the source data to accurately reproduce it.

Even worse, when enlarging, you end up picking the same data points again and again. If Ns = 5 and Nd = 20, you end up using a new source pixel every four destination pixels. This leads to the dreaded "fat pixel" effect.

Strictly speaking, when enlarging with this method, you are in fact preserving the data in the original image exactly: every source point is used in the destination image and, you are not introducing any new data into the new image. However, the results are not visually pleasing.

More sophisticated techniques can get rid of the fat pixel effect by making educated guesses as to what the data points between source pixel N and N+1 would be.

ShowBi-Linear resampling

For every destination pixel, find the location of the ideal source pixel by using the Ns/Nd ratio, as above. But this time, don't use the Floor function, preserve the fractional information. Ex. if Ns = 5 and Nd = 15 and we're trying to find the 2nd point in the destination, the ideal source point is at 0.666 (2 * (5 / 15) = 0.666). Because we can't address data at fractional locations, we'll do a weighted average of the two data points closest to our ideal location: 0 and 1. Our destination pixel is then 0.666 of the data at point 0 and (1.0 - 0.666) of the data at point 1.

This technique is called linear interpolation. If you were to graph the two data points used in the calculation above, with a line between them, the ideal data value will be somewhere on that line. Bi-linear means that you do it twice - once horizontally and once vertically. The math is the same in either direction.

Using this technique on image data gives results that are far better than the nearest neighbor technique. The images lose the fat pixel effect and you can almost believe that the resizing code has somehow recovered data that was missing from the source.

ShowBi-Cubic resampling

Cubic interpolation is similar to linear interpolation in that you use your existing data to come up with an equation to model that data so that you can make an educated guess at what other points in that data set will be. Linear interpolation uses two data points to generate a simple line, and you pick your destination data from that line; in cubic interpolation, you use four data points to generate a 3rd degree equation (of the form ax^3 + bx^2 + cx + d) and pick your destination data from the curve. This makes the math much more complicated.

In linear interpolation, you use two source data points to find one destination point. In cubic interpolation, you use four source points to find one destination point. So, each destination point is created from twice as many source points, and a cubic equation can model mathematical relationships in data much more accurately than a simple line can.

The images that result from bi-cubic interpolation (cubic interpolation performed in both X and Y directions) are often sharper than bi-linear interpolation because of the higher accuracy possible with 3rd degree equations and the higher number of source pixels used. The downside is that the math required to do this is complex and time consuming. You need to be sure the time spent generating the image is worth the slight visual improvement.

ShowArea averaging

This technique is much different from the resampling techniques described above. In those techniques, the number of source points per destination point was fixed by the requirements of the math: 2 for linear resampling, 4 for cubic resampling. The intent of this algorithm is also different. The resampling techniques are designed to reproduce or mimic the source data as closely as possible; area averaging is designed to find the average data value in a given range of data.

The way it works is simple: divide the source data into Nd regions. Each destination point is the average data value from the corresponding source region. This is a very intuitive way to reduce the size of a data stream. Every data point in the source contributes equally to the output: and a destination pixel is the average of all source pixels that it represents.

Unfortunately, this technique can only be used when reducing images. If Ns < Nd, the source regions end up being less than one pixel each and the technique degenerates into a nearest-neighbor equivalent. Also, this technique isn't good when you are only slightly reducing an image: the regions become too small to do much more than echo single source data points. But, for large reductions, area-averaging can give results that are equal to or better than any of the resampling techniques described above.

ShowFilter resampling

The Filter resizing methods are handled by a single resizing "engine". For each pixel, the engine combines a number of neighboring pixels in a weighted average to form the output pixel. Each method uses a different number of neighbor pixels and the weighting for these pixel is determined by a filtering function; each method uses a different filtering function, ranging in complexity from trivial to elaborate.

The results from these methods vary dramatically. Some of the methods are suboptimal for most purposes, while some of the others give results that far exceed any other resizing method. But, as with all things, the better the results, the longer the calculation takes.

For most purposes, the simple resampling methods (NearestNeighbor, BiLinear, AreaAverage) give respectable results. If you can afford to wait, the filter resizing methods can give outstanding results, in particular the LanczosFilter.

Morphology

Morphology is the study of form and structure. In image processing, morphology encompasses a set of techniques that merge traditional image processing with mathematical set theory. In general, morphological operations are performed on binary (two-color, 1-bit) images, although we provide extensions that support grayscale images as well.

All of the morphological operations discussed here are implemented in the Atalasoft DotImage SDK.

Set theory

Morphology applies these set theory rules to images. Additional rules may also apply.

- The union of two sets, A and B, is the set of all the elements that belong to A or to B, or to both A and B. When processing two binary images, implement union by merging the images using logical OR.
- The intersection of two sets is the set of elements that belong to both A and B. None of the elements that belong to A only or B only are included. When processing two binary images, implement intersection by merging the images with logical AND.
- The difference of A and B (A – B) is the set of elements in A that are not in B. “A – B” is equivalent to the intersection of A with the complement of B.
- The complement of the set A is the set of elements that are not in A (written here as ~A). When processing binary images, implement complement by using logical NOT.
- Two sets, A and B, are disjoint if they have no common elements.

Binary processing

Binary processing supports the following operations:

- Dilation
- Erosion
- Opening and Closing
- Hit or Miss
- Boundary Extraction
- Thinning

Dilation

Dilation is one of the two fundamental morphological operations. It is defined as the set of all points where the intersection of the structuring element and the image are non-empty. A mask

acting as the structuring element, passes over the image. For each source pixel, if any of the pixels in the mask are “1” and line up with a source pixel which is also “1”, the output pixel is “1”.

Example 1

The mask is 3x3 with its center at (1,1):

0 1 0

1 1 1

0 1 0

The source image, A, is 4x4 and looks like:

0 0 0 0

0 1 1 0

0 1 1 0

0 0 0 0

At the first pixel (0,0), with the mask centered on (0,0) none of the mask pixels overlay any source pixels where both are 1.

In the Atalasoft DotImage implementation, when the mask is off the edge, the nearest edge pixels stand-in for the pixels off the edge.

With the mask centered at source pixel (1,0), there is a match (mask pixel (1, 2) matches image pixel (1,1)). Therefore the intersection of A and the mask when the mask is at (1,0) is non-empty. So, the output pixel at (1,0) is set to “1”.

After processing the entire image, the output is as shown:

0 1 1 0

1 1 1 1

1 1 1 1

0 1 1 0

The transformation has dilated or expanded the image. It is no accident that the output image in some ways resembles the mask.

Example 2

If the mask is changed to that shown below:

0 0 0

1 1 1

0 0 0

If the same operations performed in Example 1 are repeated, the output is:

0 0 0 0

1 1 1 1

1 1 1 1

0 0 0 0

In general, the shape of the pixels in the mask has a huge effect on the output image. With an image this small, the effect is exaggerated as a 3x3 mask only affects pixels on the edges of the input image. These examples illustrate the fact that dilation increases an object’s geometric area.

Erosion

Erosion is the second of the two basic morphological operations. It is defined as the set of all points z such that the mask, translated by z, is contained in the image, A. In other words, erosion outputs a zero if any of the input pixels under the "1" pixels in the mask are zero. Formally, erosion can also be defined as the complement of the dilation of the complement of A with mask, or : Erosion(A, mask) = ~(Dilation(~A, mask)).

In terms of image processing, erosion reduces an object's geometric area.

Dilation, erosion, and logical and set theory operations make up the basis for morphological image processing.

Opening and closing

The opening operation is the dilation of the erosion of the image. It tends to smooth outward bumps, breaks narrow sections and eliminates thin protrusions. In other words, as the mask is passed over the image, the output pixel is 1 only if all non-zero elements in the mask match non-zero elements in A: it is the set of all locations where the mask fits into the input image.

The closing operation is the erosion of the dilation of the image. It tends to eliminate small holes and remove inward bumps.

Imagine a square; then imagine a smaller circle inside the square. A common analogy for opening and closing is to imagine a ball rolling around a geometric shape or, in our case, the circle rolling around inside the square. The opening of the square is all the area the circle can cover inside the square. This is most of the square except for areas in the corners where the circle does not fit. It is a square with rounded corners.

Imagine a square inside a square. The circle is inside the inner square and performs the same filling that it did in the opening example. But, after the circle has filled the square, take the complement of the inner square and intersect that with the outer square. What you get is a square with rounded corners inside a larger square. This is the closing.

Hit or miss

The Hit or Miss translation is a basic tool for shape detection. Unlike the previous four operations, Hit or Miss requires two structure elements. One element defines the foreground features to detect while the other defines the background features to detect. Strictly speaking, one could be derived from the other (assuming foreground is always the opposite of the background), but Atalasoft DotImage allows you to specify both masks explicitly.

Hit or Miss is defined as the intersection of the erosion of A by the first structure element (SE) and the erosion of the compliment of A by the second structure element: HitOrMiss = Intersect(Erosion(A, SE1), Erosion(~A, SE2)). So, the output of a Hit or Miss transformation is the set of points which match the erosion of A by SE1 (the foreground) and those which match the erosion of the compliment of A with SE2 (the background). The result is the matching of certain edge features (exactly which features depends on the shapes within the two structure elements).

Because the SEs are looking at complimentary features, they should be compliments of each other: if you have a “1” in SE1, you need to have a “0” in the corresponding location in SE2.

Boundary Extraction

Boundary Extraction is defined as the difference between the image A and the erosion of A with SE: A – Erosion(A, SE). The SE is usually a solid square of 1’s. The size of the SE determines the thickness of the border.

Thinning

The morphological thinning operation is defined as the intersection of the image A and the compliment of the Hit or Miss of A with SE1 and SE2: Thinning = Intersect(A, ~(HitOrMiss(A, SE1, SE2)). In practice, thinning is applied using a sequence of SE's (and their complements), where the definition above is applied iteratively, using each pair of SEs in the sequence.

In practice, the iteration over the sequence of SEs is generally iteratively applied itself (a loop within a loop) until one of two conditions are reached.

- The output has not changed from one iteration of the outer loop to the next, indicating that image will not change on the next iteration.
- The maximum number of iterations allowed have taken place.

Since this can be a lengthy operation, setting a maximum iteration count is often a good idea. The function in Atalasoft DotImage handles both inner and outer loops for you but, if you want to handle the outer loop yourself (to show the thinning in progress) you can call it repeatedly with a maximum iteration count of 1.

Grayscale processing

The set theory definitions above do not translate well to the grayscale world. But, as there are a number of basic operations that can be defined in slightly different ways we can do similar, though not identical, operations on grayscale images.

The grayscale processing operations are:

- Dilation
- Erosion
- Opening and Closing
- Smoothing
- Gradient

Dilation

In grayscale, dilation for a grayscale pixel is calculated for each input pixel by adding the value of a pixel in the SE to the corresponding image pixel beneath and then finding the maximum value of all the additions.

If all elements in the SE are equal, dilation becomes a simple maximum filter (output is the maximum pixel value under the filter), plus an offset.

Erosion

In grayscale, erosion for a grayscale pixel is calculated for each input pixel by subtracting the value of a pixel in the SE from the corresponding image pixel beneath; then finding the minimum value of all the subtractions.

If all elements in the SE are equal, dilation becomes a simple minimum filter (output is the minimum pixel value under the filter), minus an offset.

Opening and closing

As in the binary case, opening and closing are defined in terms of erosion and dilation. Opening tends to darken bright details while closing tends to brighten dark details.

Smoothing

A smoothing operation can be defined by performing an opening followed by a closing. Since opening reduces bright details and closing brightens dark details, the combination of the two yields an image where the extremes have been smoothed.

Gradient

The gradient is a way to highlight sharp gray level transitions (ie. edges) within an image. It is defined as the arithmetic difference between the dilation of the image and the erosion of the image. Because erosion reduces the geometric area and dilation increases the area, the difference between the two tend to accentuate the edges (where the shrinking and enlarging occurred).

Best practices

Memory management

Garbage collection

In managed code in the .NET framework, memory is allocated from the managed heap. This heap is controlled by the **Garbage Collector**.

Microsoft chose to use a **Garbage Collector** rather than more traditional deterministic finalization in order to eliminate a layer of memory management responsibility from the programmer.

The **Garbage Collector** has many benefits, but also some disadvantages. When dealing with normal classes, objects, and types, you do not need to worry about memory as it is entirely handled by the **Garbage Collector**. However, when dealing with un-managed memory such as memory pointers and handles using GDI or GDI+, because that memory does not reside on the managed heap it needs to be explicitly disposed. For that reason, many objects in .NET have a dispose() method.

The reference to the object is controlled by the managed heap. When it is time for the **Garbage Collector** to collect the reference, it runs the finalization code. As a developer, you have no idea when this finalization might happen.

When working with large chunks of memory it is a good practice to free that memory as soon as it is no longer needed by invoking the dispose() method.

Memory allocation

Images can occupy just a few bytes of data or many megabytes. For performance reasons, Atalasoft DotImage does not use the managed heap to allocate image memory. Instead it by default uses GlobalAlloc, or System.Runtime.InteropServices.Marshal.AllocHGlobal in the .NET Framework.

Similarly, Atalasoft DotImage uses GlobalFree or System.Runtime.InteropServices.Marshal.FreeHGlobal to free memory when the object is disposed. This method of allocating memory chunks is very fast and efficient, as long as there is available memory. As soon as the system memory is about to run out, or the memory chunk is extremely large, the system begins to page to file and cause delays.

Load very large image files

Atalasoft DotImage can display very large TIFF images gigabytes in size using the IScaledDecoder interface which is applied to the TiffDecoder and JpegDecoder. This interface will load the image at a scaled size without requiring all the image data to be in memory at once. The IRegionReadable interface can be used to read a rectangular section of the image without reading the entire image into memory and is applied to the TiffDecoder.

TIFF Images that are saved as a single strip do not benefit from IScaledDecoder or IRegionReadable because the entire image must be loaded into memory. We recommend using the default strip size or saving TIFF images as tiles.

Pixel memory

Memory layout

The Atalasoft DotImage memory layout is described below.

- An image is broken into a series of scanlines.
- A scanline contains a number of bytes equal to the width of the image times the number of bits per pixel divided by eight. This number is rounded up to the next higher multiple of four. For example, an image that is 1 bit per pixel and 33 pixels wide needs 33 / 1 / 8 = 4.125 bytes - this then gets rounded up to 8 bytes.

Image type	Memory layout
1 bit	Highest order bit in a byte is the leftmost in display. The lowest order bit is the rightmost.
4 bit	High order nibble (or group of 4 bits) is the leftmost in display. The low order nibble is the rightmost.
24 bit	Consecutive bytes are blue, green, and red respectively
32 bit rgb	Consecutive bytes are blue, green, red, and alpha. If the pixel format doesn't have an alpha channel, the 4th byte is a placeholder.
32 bit cmyk	Consecutive bytes are cyan, magenta, yellow, and black.
16 bit gray with alpha	Consecutive bytes are gray and alpha.
16 bit gray	Consecutive words are gray
48 bit	Consecutive words are blue, green, and red.
64 bit	Consecutive words are blue, green, red, and alpha. If the pixel format doesn't have an alpha channel, the 4th word is a placeholder.

To obtain the width of a row of an AtalaImage in bytes, you can use image.RowStride. To obtain the width of a row in bytes give a width in pixels and a PixelFormat, use the method PixelFormatUtilities.RowStride().

The number of bytes needed by an entire image is typically the height times the width in bytes. This varies with the image representation.

Pixel access

In a PixelMemory object, you can use a PixelAccessor to locate the individual pixels

A PixelAccessor can only be created by a PixelMemory object. Once created, a PixelAccessor is always tied to that PixelMemory object and therefore to the image that owns the PixelMemory.

A PixelAccessor provides you with the ability to read from and write to individual scanlines within an image without needing to know about the actual underlying memory implementation.

With a PixelAccessor, you acquire a scanline to work on and when you are done, you release it. PixelAccessor provides a number of shortcuts to make working with image memory easy and to

make it difficult to inadvertently forget to release a scanline or forget to release the PixelAccessor itself.

Acquire a Scanline

To acquire a scanline, call AcquireScanline, AcquireNextScanline, or AcquirePrevious scanline. Each will return an array or bytes that represents the scanline data. Calling ReleaseScanline releases the last scanline acquired, if any.

As a service, if you call a flavor of AcquireScanline a second time without calling ReleaseScanline, PixelAccessor does an implicit ReleaseScanline for you. Therefore it is incorrect to do something like this:

```
byte[] top, bottom;
top = accesor.AcquireScanline(0); // get the top row
bottom = accessor.AcquireScanline(memory.Height-1); // get the bottom row
// at this point top will have been released and bottom is the active scanline
```

You should consider PixelAccessors to be cheap resources. Use multiple accessors instead of one when you need to access multiple scanlines at the same time.

```
byte[] top, bottom;
top = topAccessor.AcquireScanline(0); // get the top row
bottom = bottomAccessor.AcquireScanline(memory.Height-1); // get the bottom row
```

Be aware that when you acquire a scanline, you have a non-exclusive lock on that memory. This means that other PixelAccessors can access the unchanged scanline. When a PixelAccessor releases a scanline, either implicitly or explicitly, the data is written back into the image. It is up to the client to ensure that there are no conflicts, especially in a multithreaded environment.

AcquireNextScanline always acquires the scanline after the currently acquired scanline. If there has been no previous acquisition or the previously acquired scanline was released, AcquireNextScanline is equivalent to AcquireScanline(0). If there are no more scanlines to acquire, AcquireNextScanline returns null.

AcquirePreviousScanline always acquires the scanline before the currently acquired scanline. If there has been no previous acquisition or the previously acquired scanline has been released or there are no more scanlines to acquire, AcquirePreviousScanline returns null.

It is possible for the PixelMemory object which owns an accessor to be disposed, leaving accessor that can longer correctly access memory. When a PixelMemory object is disposed, it notifies all its PixelAccessors that it is going away. At this point all its PixelAccessors are marked as invalid and no longer function.

For example, doing the following throws an exception:

C#

```
PixelAccessor pa = pixelMemory.AcquirePixelAccessor();
pixelMemory.Dispose();
// throws - the owning PixelMemory is gone.
pa.AcquireScanline(0);
```

This is considered to be a serious problem and if it occurs, it usually indicates an architectural problem in client code. It is possible to check the validity of a PixelAccessor using the Valid

property, but you are encouraged to design your code so that you need never check. That is, you should always know when a PixelAccessor is good.

Since PixelAccessor objects can contain resources, they implement IDisposable. When you are done with a PixelAccessor, you should Dispose it. If you have not released your last scanline, PixelAccessor releases it for you when it is disposed.

In addition, PixelAccessor contains a method called Release(). Release() marks the PixelAccessor as invalid and releases it back to its owning PixelMemory without releasing the last acquired scanline. This is necessary since Release() may be called by the PixelMemory object itself at a point when it would be invalid to write back into the image.

In general, it is most convenient to use a PixelAccessor within a using block as shown in the example below.

Example

Using a PixelAccessor with a Using Block

C#

```
public void ClearImage(AtalaImage image)
{
    using (PixelAccessor pa = image.PixelMemory.AcquirePixelAccessor())
    {
        byte[] row;
        while ((row = pa.AcquireNextScanline()) != null)
        {
             for (int i=0; i < row.Length; i++) row[i] = 0;
        }
    }
}
```

Example

The PixelAccessor is disposed when control leaves the using block.

Direct memory access

Sometimes it is necessary to access the memory of an image as directly as possible and without the overhead of copying it into a buffer. The definition of PixelMemory and PixelAccessor are such that this is not always possible. It is very straight-forward to design a version of PixelMemory in which the memory is laid out in many strips or tiles instead of one single block. In most cases, however, the memory is in one contiguous block. If that is the case, then the PixelMemory object also implements an interface IDirectScanlineAccess, which provides the means to make this determination and to get at scanlines through pointers.

IDirectScanlineAccess includes the method GetScanlinePtr(int i), which returns an IntPtr which points to the first byte in the ith scanline. This gives the direct memory address of any scanline, regardless of the memory layout. In addition, there is the method IsContiguous, which returns **true** if the image's memory is in one block and consecutive scanlines follow each other consecutively in memory.

If IsContiguous is **true**, then it is possible to treat GetScanlinePtr(0) as a pointer to the beginning of the image's memory, which every scanline after falling in intervals of the image's RowStride.

Currently all Atalasoft DotImage built-in commands and codecs require that PixelMemory implement IDirectScanlineAccess and that IsContiguous returns **true**. This may not be the case in the future.

In order to test if it is possible to work directly with memory easily, the PixelMemory class provides several static utility methods listed in the table below.

Method	Description
PixelMemory.PixelDataFromPixelMemory	Returns a pointer to the first scanline in an image. If the PixelMemory is not contiguous, this method returns null.
PixelMemory.ThrowOnNonContiguous	Throws an exception if the supplied AtalaImage or PixelMemory is not a contiguous block of memory.

C#

```
protected override AtalaImage PerformActualCommand(AtalaImage source, AtalaImage dest,
 Rectangle imageArea, ref ImageResults results)
{
    PixelMemory.ThrowOnNonContiguous(source.PixelMemory);
    if (dest != null)
    {
        PixelMemory.ThrowOnNonContiguous(dest.PixelMemory);
    }
    SpecializedCode.PeformUnmanagedWork(PixelMemory.PixelDataFromPixelMemory(source),
 imageArea,
        source.RowStride, PixelMemory.PixelDataFromPixelMemory(dest), dest.RowStride),
    return null;
}
```

PixelMemory types

Atalasoft DotImage provides two main PixelMemory types for public use:

- GlobalAllocPixelMemory
- BitmapPixelMemory

In GlobalAllocPixelMemory, memory is supplied by the Win32 routine GlobalAlloc and is subsequently disposed with GlobalFree. GlobalAllocPixelMemory allocates the memory itself, but is also configurable to work with any block of memory allocated by GlobalAlloc.

As most allocation in Windows code goes through GlobalAlloc, this class works under almost all circumstances. In most cases you only need to supply the total number of bytes for the memory, the height of the image and the number of bytes per scanline, rounded up to the next multiple of four bytes.

GlobalAllocPixelMemory can also work under circumstances where the first scanline does not start at the first byte of the block of allocated memory. Several versions of the constructor include a parameter called offsetToFirstScanline, which indicates how far from the first byte the first scanline should start. This offset should be a multiple of 4.

BitmapPixelMemory is a variant of PixelMemory that is constructed from a .NET Bitmap object. Once constructed, the BitmapPixelMemory object owns the Bitmap and will Dispose it when the PixelMemory is disposed.

Client code is not likely to construct PixelMemory objects directly. Instead, PixelMemory objects are typically constructed by ImageDecoder objects or by AtalaImage.

Access pixel data

You can access to the image data in an AtalaImage object directly. This data can be accessed in the following ways:

- GetPixel and SetPixel to modify the pixel values individually
- PixelMemory and PixelAccessors to manipulate the image data

Use GetPixel and SetPixel

Image pixels can be accessed with the GetPixelColor and SetPixelColor, or GetPixelIndex and SetPixelIndex: for colormapped images, to easily retreive or set color information in an image. However, these methods are slow when manipulating many pixels at once (as demonstrated in the example code below). When processing a 600 x 400 pixel 24-bit image these methods were 56 times slower then the methods in the next two sections.

Example

Set new pixel color

C#

```
Color clr;
//loop through the entire image
for (int y = 0; y < image.Height; y++)
{
    for (int x = 0; x < image.Width; x++)
    {
        //darken each channel in an image by a ratio
        clr = image.GetPixelColor(x, y);
        image.SetPixelColor(x, y, Color.FromArgb(clr.A / 2, clr.R / 2, clr.G / 2,
 clr.B / 2));
    }
}
```

Atalasoft DotImage includes classes for addressing memory without resorting to unsafe code.

Every AtalaImage includes a property called PixelMemory which is an object that represents the memory for an image. The easiest way to manipulate the image data directly is to use a PixelAccessor object taken from the image's PixelMemory.

PixelMemory and PixelAccessors use an acquire/release model for accessing an image's data. When you want a PixelAccessor object, you must acquire it from a PixelMemory object. When you are all done with a PixelAccessor, you must call its Release() method.

To operate on a scanline of image data, you must acquire the scanline from a pixel accessor. When you are done with the scanline, you must release it.

C#

```
PixelMemory pm = image.PixelMemory;
using (PixelAccessor pa = pm.AcquirePixelAccessor())
{
    for (int y=0; y < image.Height; y++)
    {
```

```
        byte[] row = pa.AcquireScanline(y);
        for (int i=0; i < row.Length; i++)
        {
            row[i] = (byte)row[i] / 2; // dim each byte value by 50%        }
        pa.ReleaseScanline();
    }
}
```

Prior to Atalasoft DotImage 8.0, it was possible although discouraged to access image memory directly by using the ImageData property of AtalaImage. In addition to the PixelMemory abstraction, Atalasoft DotImage added the ability to get the address of a scan line direction. ImageData is no longer supported and direct scanline access needs to be modified. Atalasoft DotImage 8.0 introduces the notion of locking memory before accessing it and unlocking it afterwards. This abstraction allows the notion of memory that is either movable or purgeable from physical memory when not needed. If you access image memory via PixelAccessor objects then you do not need to change your code at all. Locking and unlocking happens automatically as needed. If you used the method PixelMemory.PixelDataFromPixelMemory or used the IDirectScanlineAccess interface, you need to make some minor changes to your code.

Before calling PixelMemory.PixelDataFromPixelMemory you must call the Lock method of PixelMemory. After you are done with the memory, you must call the Unlock method of PixelMemory. This can be done manually by using a try/finally block to ensure that memory is locked and unlocked properly or your can embed a PixelMemoryLocker object in a using statement. We strongly recommend the latter approach. Both approaches are illustrated here:

C#

```
try
{
    image.PixelMemory.Lock();
    PixelMemory.ThrowOnNonContiguous(image);
    IntPtr p = PixelMemory.PixelDataFromPixelMemory(image);
    PerformDirectAccess(p);
}
finally
{
    image.PixelMemory.Unlock();
}
using (PixelMemoryLocker locker = new PixelMemoryLocker(image))
{
    PixelMemory.ThrowOnNonContiguous(image);
    IntPtr p = PixelMemory.PixelDataFromPixelMemory(image);
    PerformDirectAccess(p);
}
```

The PixelMemoryLocker object implements IDisposable such that on Dispose it unlocks the PixelMemory. Note that PixelMemoryLocker should never be used outside of a using block. Also note the check if the memory is contiguous. In the future, it may be the case the image memory will not consist of a single contiguous block. It is not safe to assume that an offset of AtalaImage.RowStride from the first scan line will be a pointer to the second scanline unless the PixelMemory implements IDirectScanlineAccess and IDirectScanlineAccess.IsContiguous return true.

Calling PixelMemory.PixelDataFromPixelMemory or IDirectScanlineAccess.GetScanLinePtr on PixelMemory that is not locked will throw an exception if that operation is undefined. ManagedPixelMemory will throw a PixelMemoryException if GetScanLinePtr is called on unlocked memory. ManagedPixelMemory used memory in the managed heap. When the

PixelMemory is unlocked, the garbage collector is free to move that memory without warning, hence the restriction.

Create custom PixelMemory

In rare cases, it is necessary to create a custom PixelMemory implementation. There are two approaches to creating your own implementation of PixelMemory.

- The first is to subclass PixelMemory
- The second is to subclass ResidentPixelMemory

Subclass PixelMemory

1. Implement DisposeManagedResource, the method responsible for disposing any objects contained within your implementation of IDisposable.
2. Implement DisposeUnmanagedResources. The method responsible for disposing any objects from unmanaged code (such as blocks of memory allocated outside the GC heap).
3. Implement LLGetHeight. The method that returns the height of the image that this memory object represents.
4. Implement LLGetRowStride. The method that returns the width of the row in bytes, rounded up to the next multiple of 4.
5. Implement LLGetPixelAccessor. The method that returns a new PixelAccessor object suitable for accessing image data in your PixelMemory.
6. Implement LLClone. The method that performs a deep copy of the PixelMemory
7. Implement a suitable PixelAccessor for your PixelMemory.

Create a custom PixelAccessor

1. Implement LLAcquireScanline. The method that acquires a single scanline and returns an IntPtr to the first byte.
2. Implement LLReleaseScanline. The method that releases a previously acquired scanline.
3. Implement DisposeManagedResources. The method responsible for disposing any objects contained within your implementation that implement IDisposable.
4. Implement DisposeUnmanagedResources. The method is responsible for disposing any objects from unmanaged code (such as blocks of memory allocated outside the GC heap).

When your subclass of PixelMemory either allocates or frees memory, it should call appropriate methods in PixelMemoryTracker.Memory to report changes in allocated memory.

Subclass ResidentPixelMemory

When subclassing ResidentPixelMemory, implement the following methods:

1. Implement LLAllocateMemory. The method allocates a block of memory, optionally clearing it.
2. Implement LLFreeMemory. The method frees a block of memory previously allocated by LLAllocateMemory.

3. Implement LLReallocateMemory - the method resizes a block of memory, either shrinking it or expanding it as needed.

   There is no need to implement a PixelAccessor or track memory within a subclass of ResidentPixelMemory. That is managed for you.

Image Source

The ImageSource class makes it possible to work effectively with an arbitrary number of images without worrying about the details of where those images come from and how they are managed.

The ImageSource object allows you to work with a potentially unbounded number of AtalaImage objects without requiring that all those images be in memory simultaneously.

You can think of ImageSource as half of a source/sink pair. An ImageSource is a place from which images come. The sink is an application or an image consumer. Images are managed through an acquire/release model. The ImageSource object performs the following services:

- Acquire images in order
- Release images tin any order
- Track available image memory
- Automatically free released images using either lazy or aggressive mechanisms
- Allows limited reacquisition of released images
- Allows reloading mechanism to enable images to be cached

ImageSource does not dictate where or how images are loaded into memory. Instead, it manages the conditions under which images come and go. The ImageSource subclass FileSystemImageSource, is one of the subclasses responsible for the actual loading of images.

In this model, an image can be thought of as a resource. Rather than simply being read and used, an image is acquired from the ImageSource and released when processing is complete. Any number of consumers can acquire any given image, and it is released only when each Acquire has been balanced with a Release.

You can configure an ImageSource to aggressively unload images or you can configure the source to observe memory restrictions. Some ImageSource variants can reload previously unloaded images or cache images.

While it appears that an ImageSource could implement IEnumerator or IEnumerable, those interfaces have facilities which are not guaranteed to function in ImageSource.

Inside the ImageSource class

An image can be thought of as a resource. Rather than simply being read and used, an image is acquired from the ImageSource and released when finished. Any number of consumers can acquire any given image and it is released only when each **Acquire** has been balanced with a **Release**.

In this way, an ImageSource can be used as follows:

C#

```
public void ProcessImages(ImageSource source)
{
```

```
      while (source.HasMoreImages()) {
            AtalaImage image = source.AcquireNext();
            ProcessImage(image);
            source.Release(image);
      }
}
```

An image that has been acquired and is not yet released can be acquired any number of times. In the example above, all the images within the ImageSource are processed serially. You can make this a parallel process by creating worker threads to perform the processing and allow them to acquire and release the images as well. Structuring the code as follows makes that possible:

C#

```
public void ProcessImages(ImageSource source)
{
      while (source.HasMoreImages()) {
AtalaImage image = source.AcquireNext();
            CreateImageWorker(source, image, ProcessImage);
            source.Release(image);
      }
}

private void ProcessImage(ImageSource source, AtalaImage image)
{
      // do processing here
      source.Release(image);
}

public delegate void ProcessImageProc(ImageSource source,
AtalaImage image);

public void CreateImageWorker(ImageSource source,
                              AtalaImage image, ProcessImageProc proc)
{
      source.Acquire(image); // Acquire here
      Thread t = CreateImageWorkerThread(source, image, proc);
      t.Start();
}

private Thread CreateImageWorkerThread(ImageSource source,
                              AtalaImage image, ProcessImageProc proc)
{
      // threading details left out
}
```

In the above code, the main loop acquires each image, passes it to CreateImageWorker, then releases it. CreateImageWorker calls Acquire for a second time, then creates a worker thread to do the processing, starts it and returns. The worker thread calls ProcessImage which does the work before calling Release. In this way, the images are processed in parallel.

ImageSource categorizes images into three groups, Acquired, Released, and Culled. An image that is Acquired is in memory and available for use. An image that is Released is in memory, but should not be used until it has been reacquired. An image that is Culled is no longer in memory, but may have the facility to be reloaded.

Example:

C#

```
TryOne(ImageSource source)
{
   source.Reset();
   AtalaImage image = source.AcquireNext();
   AtalaImage image1 = source.Acquire(0); // reacquire the 0th image
}
```

If image is non-null, image1 is always non-null and identical to image.

This code works in most cases:

C#

```
TryTwo(ImageSource source)
{
   source.Reset();
   AtalaImage image = source.AcquireNext();
   source.Release(image);
   AtalaImage image1 = source.Acquire(0); // reacquire the 0th image
}
```

ImageSource marks image as Released and unless there are severe memory restrictions, the image can be reacquired. The resulting image, however, should be checked for null.

This code only reliably works if the particular ImageSource implements reloadable images:

C#

```
TryThree(ImageSource source)
{
   source.Reset();
   while (source.HasMoreImages()) {
AtalaImage image = source.AcquireNext();
      source.Release(image);
   }
   AtalaImage image1 = source.Acquire(0); // reacquire the 0th image
}
```

The ability to reload an image is not defined within ImageSource, but is instead left to a class that descends from ImageSource.

On its own, ImageSource is geared perfectly for situations where an image can be accessed once and only once, such as a video source or a scanner with an automatic feeder.

Since not every ImageSource falls into this category, there is an abstract descendant of ImageSource called RandomAccessImageSource. For a RandomAccessImageSource, any image can be reliably acquired at any time and in any order. Again, images may be Acquired, Released, and Culled, but in this case Acquire should always succeed.

RandomAccessImageSource adds the array operator to the object and the Count property. In this way, it is possible to access the image source as shown below.

C#

```
public void ProcessImages(RandomAccessImageSource source)
{
      for (int i=0; i < source.Count; i++) {
```

```
            AtalaImage image = source[i]; // this does the acquire
            ProcessImage(image);
            source.Release(image);
      }
}
```

From here it is a short step to get to the main concrete ImageSource class, FileSystemImageSource. FileSystemImageSource allows a client to iterate over a set of image files as well as multiple frames within image files that support that. Since it is clearly a variety of ImageSource that can trivially reload images, it descends from RandomAccessImageSource. As designed, FileSystemImageSource can iterate over all image files within a folder, all files matching a pattern within a folder or through a list of files. Optionally, FileSystemImageSource also iterates across all frames.

For better or for worse, pattern matching is limited to that provided by .NET for files. This is not full regular expression matching. On one hand, it is consistent with the general Windows User Interface, but on the other hand it is somewhat limited.

To avoid that inherent limitation, yet maintain compatibility, FileSystemImageSource includes a file filter hook to allow a client to perform all filtration of image files. By setting the FileFilterDelegate property to a method of the form:

C#

```
bool MyFilter(string path, int frameIndex, int frameCount)
{
}
```

A client is able to allow or disallow any file based on its own criteria. By returning true from the FileFilterDelegate, a file or frame within a file is included. Return false and the file or frame is ignored.

To implement a custom ImageSource, create a class that inherits from either ImageSource or RandomAccessImageSource. A class that inherits from ImageSource asserts that it can provide a sequence of images in order. To do so, a class must implement the following abstract methods:

```
protected abstract ImageSourceNode LowLevelAcquireNextImage();
```

LowLevelAcquireNextImage gets the next available image in the sequence and returns it packaged in an ImageSourceNode. An ImageSourceNode is used to manage an image while it is in memory.

The main constructor for ImageSourceNode takes an AtalaImage as an argument and an object that implements the IImageReloader interface. An IImageReloader is a class that makes it possible to reload an image into memory. For a typical class inheriting from ImageSource, the LowLevelAcquireNextImage() returns a new ImageSourceNode with a valid image, but a null IImageReloader. This indicates that the image cannot be reloaded once it has been culled from memory. If it is not possible to acquire the next image, LowLevelAcquireNextImage returns null.

```
protected abstract bool LowLevelHasMoreImages();
```

LowLevelHasMoreImages returns a boolean indicating whether or not there are more images to be loaded.

```
protected abstract void LowLevelReset();
```

LowLevelReset returns an ImageSource to its starting state, if possible. For some ImageSources, this is not always possible. If it is not possible to Reset, this method does nothing.

```
protected abstract void LowLevelSkipNextImage();
```

LowLevelSkipNextImage is called when an image that had previously been loaded is still available. For example, if ImageSource needs to load an image, it calls LowLevelAcquireNext, but if it determines that it is not necessary to load an image, it does not call LowLevelAcquireNext. In this case it is necessary to allow your class to maintain its bookkeeping.

```
protected abstract void LowLevelDispose();
```

LowLevelDispose is called to allow a class to dispose of any non-reclaimable resources when then class is garbage collected. This might include closing files, releasing devices, closing network connections, and so forth.

```
protected abstract bool LowLevelFlushOnReset();
```

LowLevelFlushOnReset indicates whether or not ImageSource should dump all cached images upon Reset. For ImageSource varieties that will not return the same sequence of images every single time, this method should return true. Typically, most classes will return false to take full advantage of the cache.

```
protected abstract bool LowLevelTotalImagesKnown();
```

LowLevelTotalImagesKnown returns true if this ImageSource can know a priori how many images are available, false otherwise.

```
protected abstract int LowLevelTotalImages();
```

LowLevelTotalImages returns the total number of available images. If LowLevelTotalImagesKnown returns false, this will never be called.

A RandomAccessImageSource adds one new method to implement:

```
protected abstract ImageSourceNode LowLevelAcquire(int index);
```

LowLevelAcquire acts just like LowLevelAcquireNext except that it passes in an index. With this method, it's convenient to implement LowLevelAcquireNext in terms of LowLevelAcquire.

ℹ A class that inherits from RandomAccessImageSource must provide an IImageReloader when it is asked to load an image. Without this, it is impossible to guarantee robust operation of the ImageSource.

In addition RandomAccessImageSource implements LowLevelTotalImagesKnown, returning true.

Use an ImageSource

Typically, an image source loops over each element of the ImageSource, working with each image in turn. The following example illustrates how this might be done.

Loop to process images

You can loop process images using the following example.

C#

```
public void ProcessImages(ImageSource source)
{
    while (source.HasMoreImages()) {
        AtalaImage image = source.AcquireNext();
        DoSomeWork(image);
```

```
        source.Release(image);
}
```

Use an image source multiple times

You can use an ImageSource multiple times by employing the Reset method as shown below.

C#

```
public void ProcessingLoop(ImageSource source)
{
    while (NotDone()) {
        ProcessImages(source);
        source.Reset();
    }
}
```

The reset() method can produce unexpected results. Although reset() implies that you are starting over, some ImageSource objects lack that capability. For example, a live video feed or a document feeder cannot restart at the beginning since there is no clear beginning. Similarly, In the case of a live video feed, there is no clear end.

You must call Release for every call to **AcquireNext** or **Acquire**.

When an image is acquired, it is held in memory as long as the ImageSource is in memory.

Create a custom ImageSource

Making a custom ImageSource is a straight-forward process. Management of image coming and going is taken care of in the base class. This leaves you with the task of bringing the image in and providing some simple management code.

To create a custom ImageSource, create a class which inherits from ImageSource and which overrides the following methods:

- protected abstract ImageSourceNode LowLevelAcquireNextImage();
- protected abstract bool LowLevelHasMoreImages();
- protected abstract void LowLevelReset();
- protected abstract bool LowLevelFlushOnReset();
- protected abstract void LowLevelSkipNextImage();
- protected abstract void LowLevelDispose();

LowLevelAcquireNextImage gets the next image in the current sequence. The result is returned as an ImageSourceNode, a simple class for holding images and tracking acquisition information.

If you are writing an image source that is reading from a peripheral device, you can implement LowLevelAcquireNextImage as shown below.

Examples

Acquire an image from a peripheral device

C#

```
protected override ImageSourceNode LowLevelAcquireNextImage()
```

```
{
    AtalaImage image = MyPeripheralGetNthImage(MyImageCount);
    if (image != null) {
        MyImageCount++;
        return new ImageSourceNode(image, null);
    }
    return null;
}
```

Read additional images

LowLevelHasMoreImages is responsible for reporting whether or not there are additional images to be read. Returns true if there are, false otherwise:

C#

```
protected override bool LowLevelHasMoreImages()
{
     return MyImageCount < MyPeripheralTotalImageCount();
}
```

Reset image processing state

LowLevelReset provides a way to reset state in your object so that it is ready for reading from the beginning again. It is not strictly necessary for this method to do anything, as in some cases, it may not be possible to reset in a meaningful way.

C#

```
protected override void LowLevelReset()
{
    MyImageCount = 0;
    MyPeripheralReset();
}
```

If an ImageSource has been Reset, it is possible to keep old images in the source. In some cases--file systems, for example-- keeping around old images that have not been unloaded after a reset results in a significant performance gain. In other cases, keeping old images around can produce unexpected results. Therefore an image source needs to indicate whether or not old images should be flushed on reset via the method LowLevelFlushOnReset. Return true, if old images should be flushed, false otherwise.

Indicating Whether Old Images Should Be Flushed

C#

```
protected override bool LowLevelFlushOnReset()
{
    return false;
}
```

Track skipped images

If you are returning true in LowLevelFlushOnReset, LowLevelAcquireNextImage is called each time AcquireNextImage is called. If you are returning false in LowLevelFlushOnReset, then LowLevelAcquireNextImage may not be called for every call to AcquireNextImage. In this case, you need to create code in the method LowLevelSkipNextImage() that maintains your own internal bookkeeping:

C#

```
protected override bool LowLevelSkipNextImage()
{
    MyImageCount++;
```

```
}
```

Call LowLevelDispose for a device

LowLevelDispose() allows your object an opportunity to release resources, such as open file handles, that may not be handled by normal garbage collection.

Call LowLevelDispose() only one time.

C#

```
protected override bool LowLevelDispose()
{
    MyPeripheralDeviceClose();
}
```

Unload an image

When all acquisitions of an image have been released, the ImageSource determines whether to unload the image from memory based on the setting of the property ImmediateUnload. By default, the property value is set to true to minimize memory use.

When unloading, ImageSource calls the image's Dispose() method and then drop its reference to the image. ImageSource does not specifically induce garbage collection. Rather, the decision as to when garbage collection is run is left to the application and to the CLR.

When ImmediateUnload is set to false, images are kept around. In this case, an image can be confidently reacquired using the two flavors of Acquire.

Setting ImmediateUnload to false without any other adjustments to the ImageSource is tantamount to using an ImageCollection which keeps all images in memory at all times. Alternatively, you can set memory usage limits through the MemoryLimit property. Setting this property to a value greater than 0 forces ImageSource to unload fully released images if memory usage is greater than the MemoryLimit. MemoryLimit is not a hard limit. Rather it is a suggestion to ImageSource to attempt to limit memory use to the specified amount. If client code is careful about releasing images it is done processing, the limit is more likely to be obeyed. If the client code never releases images, then ImageSource is not able to comply with the request to unload images.

Example code demonstrating the use of the MemoryLimit property is provided below.

Example

Set a memory limit

C#

```
private ArrayList imageStore;

// process a set of images, keeping 5 around
public void ProcessImages(ImageSource source)
{
    source.ImmediateUnload = false;

    // 8 megabyte limit
    source.MemoryLimit = 1024 * 1024 * 8;
    while (source.HasMoreImages()) {
        AtalaImage image = source.AcquireNext();
        StoreAndProcessImage(source, image);
```

```
    }
    ReleaseAll(source);
}

private void StoreAndProcess(ImageSource source, AtalaImage image)
{
    // only keep 5 images
 if (imageStore.Count > 5) {
        AtalaImage oldImage = (AtalaImage)imageStore[0];
        imageStore.RemoveAt(0);
        source.Release(oldImage);
    }
    imageStore.Add(image);
    Process(image);
}

private void ReleaseAll(ImageSource source)
{
    for (i=0; i < imageStore.Count; i++) {
        source.Release((AtalaImage)(imageStore[i]));
    }
    imageStore.Clear();
}
```

It is possible to reaccess a previously unloaded image. Doing so requires that the ImageSource supply the image with an object that can reload the image as needed. If such an object is supplied, you can reload a previously released and unloaded image as needed.

Write an ImageSource cache

In many cases, an ImageSource loads images from a source that lacks the ability to retain old images. In such cases, you may want to cache these images so that they can be loaded and unloaded at will. The ImageSourceFileCache class makes that task easy.

ImageSourceFileCache is not an ImageSource. Rather, it is an object that can cache an image in a file and return an IImageReloader that can bring that image back into memory at a later date. ImageSourceFileCache also manages temporary files and their removal.

Use the following code to create an object of type ImageSourceFileCache in your custom ImageSource object.

Create an image source cache

This example creates an ImageSourceFileCache.

C#

```
public class MyCustomImageSource : ImageSource
{
    private ImageSourceFileCache _cache;

    public MyCustomImageSource()
        : base()
    {
        _cache = new ImageSourceFileCache(null);
    }
}
```

Acquire the next cached image

The cache is also needed to implement of LowLevelAcquireNext as shown below.

C#

```
public AtalaImage LowLevelAcquireNext()
{
    AtalaImage image = MyPeripheralGetNthImage(MyImageCount);
    if (image != null) {
        MyImageCount++;
        IImageReloader reloader = _cache.Cache(image);
        return new ImageSourceNode(image, reloader);
    }
    return null;
}
```

When _cache.Cache() is called, the image is written out to a temporary file on disk. By default, this image is written out as a TIFF file, a choice flexible enough to handle most image file formats. If this is not sufficient, by providing the ImageSourceFileCache with an Encoder object, you can specify a different file format.

By default, the ImageSourceFileCache stores temporary files in the windows temporary folder (typically C:\WINDOWS\Temp). You can make an alternate choice by passing a path to an existing writable folder to the constructor of ImageSourceFileCache.

Loop over images in a file system

The FileSystemImageSource object provides a flexible mechanism for looping over image files contained in a file system. A FileSystemImageSource can be constructed with a path to a directory, a path to a directory with a search pattern, or an array of paths to files. In addition, if a file contains multiple images, FileSystemImageSource can optionally look at all frames in a file.

For example, to create a FileSystemImageSource that loops over every file in the root folder of your C drive, construct it like this:

Loop over every file in the root folder

C#

```
FileSystemImageSource source = new FileSystemImageSource(@"C:\", true);

//To create a FileSystemImageSource that can loop over all images files presented
 by a digital camera,
//do something along these lines:

FileSystemImageSource source = new FileSystemImageSource(@"E:\DCIM\", "DSC*.*",
 true);

//To loop over these files, use code like this:
for (int i=0; i < source.Count; i++)
{
    AtalaImage image = source[i];
    // do something interesting with image
    source.Release(image);
}

//The above code is exactly equivalent to the code that follows
 while (source.HasMoreImages())
{
    AtalaImage image = source.AcquireNext();
```

```
    // do something interesting with image
    source.Release(image);
}
```

Filtering

All files in a directory that match the given search pattern or all paths passed in are considered to be candidates. Before these candidates are used, there are two levels of filtering that happen. First the candidate is tested to see if it is an image type recognized by Atalasoft DotImage. If that's the case, then the path is passed to a FileFilterDelegate. If the FileFilterDelegate accepts the path, then the image will be available for acquisition.

In addition, if the ImageSource was constructed with doAllFrames set to true then each frame within an image file will also be considered a candidate and handed to the FileFilterDelegate.

Since the pattern that can be passed in to the FileSystemConstructor is limited to the * and ? wildcard characters, it is convenient to be able to apply a more complicated pattern or to filter on things other than the filename.

For example, the following example shows how a simple filter will turn a regular FileSystemImageSource into a gray scale image finder:

C#

Use a simple filter to turn a FileSystemImage source into a gray scale image finder:

```
private bool IsGrayscale(string fileName, int frameIndex, int frameCount)
{
    ImageInfo info = RegisteredDecoders.GetImageInfo(fileName, frameIndex);
    if (info != null)
    {
        return info.PixelFormat == PixelFormat.Pixel1bppIndexed ||
            info.PixelFormat == Pixel8bppGrayscale ||
            info.PixelFormat == Pixel16bppGrayscale;
    }
    else return false;
}

public void MakeGrayscaleImageSource(FileSystemImageSource source)
{
    source.FileFilter = new FileSystemImageSource.FileFilterDelegate(IsGrayscale);
}
```

Filtering happens immediately before the very first image the FileSystemImageSource is used to acquire an image. The candidate files will also be filtered again after calling the FileSystemImageSource's Reset() method. Note that this exact behavior is described to help conceptualize the operation of the filtering. It may be subject to change in the future.

Events

In addition to the filter delegate, the FileSystemImageSource provides an event called ImageAcquired. This event is fired every time the ImageSource constructs an image in response to an Acquire. This sounds complicated, but it's fairly simple. The base class, ImageSource, contains all the logic for caching and holding onto images. FileSystemImageSource knows nothing about the caching, so it can only fire and event when it has been requested to retrieve an image. This is actually convenient in that this event is fired only once for every image. This makes it easy to add a pre-processing step into an image source or to track images as they go by.

Create an ImageSource to accesses Windows AVI files

The real power in ImageSource is the ability to create new sources that can be used generically. What follows is a complete example of an ImageSource that can access Windows AVI files.

In this class we want to be able to load every frame of an AVI file. Since AVI files can be read at any point, this is a good candidate for a RandomAccessImageSource as the base class, although a plain ImageSource would work.

This class contains a number of PInvoke definitions that link directly to the Win32 AVI calls. Discussion of the operation of these methods is beyond the scope of this document.

Most of the work is in opening the AVI file and loading a frame. All the rest of the abstract members of RandomAccessImageSource end up being one line methods. This is a very good thing as it leads to highly robust software.

C#

```
usingSystem;
usingSystem.Runtime.InteropServices;
usingAtalasoft.Imaging;

namespaceAviSource
{
       public class AviImageSource : RandomAccessImageSource
       {
              string _fileName;
              IntPtr _aviFileHandle = IntPtr.Zero;
              int _currentFrame = 0;
              int _firstFramePosition;
              int _totalFrames = 0;
              IntPtr _aviStream = IntPtr.Zero;
              AVISTREAMINFO _streamInfo = new AVISTREAMINFO();

              static AviImageSource()
              {
                     AVIFileInit();
              }

              public AviImageSource(string fileName)
              {
                     _fileName = fileName;
                     // LowLevelReset will force the file to be loaded
                     // and will fetch all the relevant information
                     LowLevelReset();
              }

              protected override void LowLevelReset()
              {
                     // attempt to load the file if we haven't
 if (_aviFileHandle == IntPtr.Zero)
                     {
                            OpenAvi();
                            LoadAviInfo();
                     }
                     // reset the frame counter
                     _currentFrame = 0;
              }
```

```
            private void CloseAvi()
            {
                  // clear everything out
                  _currentFrame = 0;
                  _totalFrames = 0;

                  // if the file handle is non-null, there may be a stream to close
if (_aviFileHandle != IntPtr.Zero)
                  {
                        // if the stream handle is non-null, close it
if (_aviStream != IntPtr.Zero)
                        {
                              AVIStreamRelease(_aviStream);
                              _aviStream = IntPtr.Zero;
                        }
                        AVIFileRelease(_aviFileHandle);
                        _aviFileHandle = IntPtr.Zero;
                  }
            }

            private void OpenAvi()
            {
                  // open the file and get a stream interface
int result = AVIFileOpen(out _aviFileHandle, _fileName,
32 /*OF_SHARE_DENY_WRITE*/, 0);
                  if (result != 0)
                        throw new Exception("Unable to open avi file " + _fileName +
" (" + result + ")");

                  result = AVIFileGetStream(_aviFileHandle, out _aviStream,
0x73646976 /* 'vids' -> four char code */, 0);
                  if (result != 0)
                        throw new Exception("Unable to get video stream (" + result
+ ")");
            }

            private void LoadAviInfo()
            {
                  if (_aviStream == IntPtr.Zero)
                        throw new Exception("LoadAviInfo(): Bad stream handle.");

                  // get first frame
                  _firstFramePosition = AVIStreamStart(_aviStream);
                  if (_firstFramePosition < 0)
                        throw new Exception("LoadAviInfo(): Unable to get stream
start position.");

                  // get total frame count
                  _totalFrames = AVIStreamLength(_aviStream);
                  if (_totalFrames < 0)
                        throw new Exception("LoadAviInfo(): Unable to get stream
length.");

                  // pull in general information
int result = AVIStreamInfo(_aviStream, ref _streamInfo,
                                                      Marshal.SizeOf(_streamInfo));
                  if (result != 0)
                        throw new Exception("LoadAviInfo(): unable to get stream
info (" + result + ")");
            }

            // this method retrieves a frame from the file.
            // the class is internal because it will be used by
            // the AviImageReloader class.
```

```
internal AtalaImage GetAviFrame(int frame)
            {
                  // set up a bitmap info header to make a frame request
                  BITMAPINFOHEADER bih = new BITMAPINFOHEADER();
                  bih.biBitCount = 24;
                  bih.biCompression = 0; //BI_RGB;
                  bih.biHeight = _streamInfo.frameBottom;
                  bih.biWidth = _streamInfo.frameRight;
                  bih.biPlanes = 1;
                  bih.biSize = (uint)Marshal.SizeOf(bih);

                  // the getFrameObject is an accessor for retrieving a frame
                  // from an AVI file.  We could make exactly one when the stream
                  // is opened, but this works just fine.

                  IntPtr frameAccessor = AVIStreamGetFrameOpen(_aviStream, ref bih);
                  if (frameAccessor == IntPtr.Zero)
                        throw new Exception("Unable to get frame decompressor.");

                  IntPtr theFrame = AVIStreamGetFrame(frameAccessor, frame +
_firstFramePosition);

                  if (theFrame == IntPtr.Zero)
                  {
                        AVIStreamGetFrameClose(frameAccessor);
                        throw new Exception("Unable to get frame #" + frame);
                  }

                  // make a copy of this image
                  AtalaImage image = AtalaImage.FromDib(theFrame, true);
                  // closing the frame accessor drops the memory used by the frame
as well
                  AVIStreamGetFrameClose(frameAccessor);

                  return image;
            }

            protected override ImageSourceNode LowLevelAcquireNextImage()
            {
                  if (_currentFrame >= _totalFrames)
                        return null;
                  AtalaImage image = GetAviFrame(_currentFrame);
                  if (image != null)
                  {
                        ImageSourceNode node = new ImageSourceNode(image, null);
                        _currentFrame++;
                        return node;
                  }
                  return null;
            }

            protected override ImageSourceNode LowLevelAcquire(int index)
            {
                  if (index < 0 || index >= _totalFrames)
                        return null;
                  AtalaImage image = GetAviFrame(index);
                  if (image != null)
                  {
                        ImageSourceNode node = new ImageSourceNode(image, new
AviImageReloader(this, index));
                        _currentFrame++;
                        return node;
                  }
                  return null;
```

```
        }

        protected override bool LowLevelTotalImagesKnown()
        {
               return true;
        }

        protected override int LowLevelTotalImages()
        {
               return _totalFrames;
        }

        protected override bool LowLevelHasMoreImages()
        {
               return _currentFrame < _totalFrames;
        }

        protected override void LowLevelSkipNextImage()
        {
               _currentFrame++;
        }

        protected override bool LowLevelFlushOnReset()
        {
               return true;
        }

        protected override void LowLevelDispose()
        {
               CloseAvi();
        }


        #regionAviHooks
        [DllImport("avifil32.dll")]
        private static extern void AVIFileInit();

        [DllImport("avifil32.dll", PreserveSig=true)]
        private static extern int AVIFileOpen(
               out IntPtr ppfile,
               String szFile,
               int uMode,
               int pclsidHandler);

        [DllImport("avifil32.dll")]
        private static extern int AVIFileGetStream(
               IntPtr pfile,
               out IntPtr ppavi,
               int fccType,
               int lParam);

        [DllImport("avifil32.dll")]
        private static extern int AVIStreamRelease(IntPtr aviStream);

        [DllImport("avifil32.dll")]
        private static extern int AVIFileRelease(IntPtr pfile);

        [DllImport("avifil32.dll")]
        private static extern void AVIFileExit();

        [DllImport("avifil32.dll", PreserveSig=true)]
        private static extern int AVIStreamStart(IntPtr pAVIStream);

        [DllImport("avifil32.dll", PreserveSig=true)]
```

```
            private static extern int AVIStreamLength(IntPtr pAVIStream);

            [DllImport("avifil32.dll")]
            private static extern int AVIStreamInfo(
                   IntPtr pAVIStream,
                   ref AVISTREAMINFO psi,
                   int lSize);

            [DllImport("avifil32.dll")]
            private static extern IntPtr AVIStreamGetFrameOpen(
                   IntPtr pAVIStream,
                   ref BITMAPINFOHEADER bih);

            [DllImport("avifil32.dll")]
            private static extern IntPtr AVIStreamGetFrame(
                   IntPtr pGetFrameObj,
                   int lPos);

            [DllImport("avifil32.dll")]
            private static extern int AVIStreamGetFrameClose(IntPtr pGetFrameObj);

            #endregion
      }
}
```

In addition to this class, it is necessary to have a class that implements IImageReloader. For this, we provide an AviReloader class which encapsulates enough information to reload a frame from a file. In this case, it is the frame index and the AviImageSource from which it came. AviImageSource has one internal method which extracts a frame and converts it to an AtalaImage. Rather than keep any more information than is needed, we can just use this method. This assumes that the AVI file and the associated stream will still be open when the image is reloaded, but since this is kept across the life of the AviImageSource object, this is a safe assumption to make.

C#

```
usingSystem;
usingAtalasoft.Imaging;

namespaceAviSource
{
      public class AviImageReloader : IImageReloader
      {
            private int _frame;
            private AviImageSource _source;
            public AviImageReloader(AviImageSource source, int frame)
            {
                  _source = source;
                  _frame = frame;
            }

            #regionIImageReloader Members

            public AtalaImage Reload()
            {
                  return _source.GetAviFrame(_frame);
            }

            #endregion

            #regionIDisposable Members

            public void Dispose()
            {
```

```
            }

            #endregion
        }
}
```

Access images

Read and write images to a database

Images can be stored in a database with Atalasoft DotImage by using a MemoryStream, or the convenient ToByteArray() and FromByteArray() methods of the AtalaImage object.

When using a SQL database, the image should be stored in a binary Image field. In MS Access this would be an OLE field. The following code samples show how to read and write images with either SQL or Access databases using ADO.NET.

Example

Writing an Image Into a SQL Database

This example demonstrates writing an AtalaImage into a SQL database as a JPEG encoded image where image is an AtalaImage object. Be sure to declare the System.Data.SqlClient namespace directive.

C#

```
SqlConnection myConnection = null;
try
{
   //save image to byte array and allocate enough memory for the image
 byte[] imagedata = image.ToByteArray(new Atalasoft.Imaging.Codec.JpegEncoder(75));

   //create the SQL statement to add the image data
   myConnection = new SqlConnection(CONNECTION_STRING);
   SqlCommand myCommand = new SqlCommand("INSERT INTO Atalasoft_Image_Database
        (Caption, ImageData) VALUES ('" + txtCaption.Text + "', @Image)",
 myConnection);
   SqlParameter myParameter = new SqlParameter("@Image", SqlDbType.Image,
 imagedata.Length);
   myParameter.Value = imagedata;
   myCommand.Parameters.Add(myParameter);

   //open the connection and execture the statement
   myConnection.Open();
   myCommand.ExecuteNonQuery();
}
finally
{
   myConnection.Close();
}
```

Example

Reading an Image From an Access/OLE Database

Similarly an image can be retrieved from an access database. Be sure to declare the System.Data.OleDb namespace directive.

C#

```
OleDbConnection myConnection = null;
try
{
   //establish connection and SELECT statement
   myConnection = new OleDbConnection(CONNECTION_STRING);
   OleDbCommand myCommand = new OleDbCommand("SELECT ImageData FROM
 [Atalasoft_Image_Database]
        WHERE Caption = '" + txtCaption.Text + "'", myConnection);
   myConnection.Open();

   //get the image from the database
 byte[] imagedata = (byte[])myCommand.ExecuteScalar();
   if (imagedata != null)
   {
      return AtalaImage.FromByteArray(imagedata);
   }
   else
   {
      MessageBox.Show("Image does not exist in database.");
      return null;
   }
}
finally
{
   myConnection.Close();
}
```

Access multipage images

For more information on working with multipage documents, see the following:

- Save an image to a multipage TIFF file
- Remove pages from a multipage TIFF
- Work with multipage TIFFs

Add support for RAW images

As some RAW images are recognized as TIFF images by the TIFF Decoder, the RAW Decoder must be inserted before the TIFF Decoder as shown in the example below.

1. Open the **RegisteredDecoders** collection.

2. Add the RawDecoder to the list.

   ❗ Because some RAW images are recognized as TIFF images by the TIFF Decoder, the RAW Decoder must be inserted before the TIFF Decoder

   Support for RAW images is now enabled.

   as shown in the example below.

   Example

   This example code shows how to add support for RAW images in your application by inserting the RAW Decoder as the first item in the Decoders collection.

   C#

```
Atalasoft.Imaging.Codec.RegisteredDecoders.Decoders.Insert(0,
    new Atalasoft.Imaging.Codec.RawDecoder());
```

Get image information

At a minimum, all decoders can retrieve an image's width, height, bitdepth, and PixelFormat using the GetImageInfo() method of the RegisteredDecoders class without loading the image data into memory. The information is returned as an ImageInfo object.

In some cases, an image format has extended information, and returns this information in an object derived from ImageInfo.

1. To retrieve the extended information from a PNG image, get the image information.
2. Confirm that the image type is PNG.
3. Cast to the specialized ImageInfo class.

   Example

   The following example demonstrates how to work with image format information.

   C#

```
ImageInfo info = RegisteredDecoders.GetImageInfo("c:\\test.png");
if (info.ImageType == ImageType.Png)
{
    PngImageInfo pngInfo = (PngImageInfo)info;
    Console.WriteLine("Png Interlaced = " + pngInfo.PngInterlaced);
}
```

View images

ASP.NET WebForm controls

Server-side image viewing with ASP.NET

DotImage Document Imaging can be used to display and edit images in ASP.NET WebForm applications.

The primary control is the WebImageViewer which is the version of the WinForm's WorkspaceViewer. It is an AJAX server-side image viewer with methods to open, save, and process images.

The WebImageViewer control is a server control that can display any image supported by DotImage. When an image is viewed, the control sections the original image into tiles and streams each tile to the browser as needed based on the scroll position of the control. The control is efficient at loading TIFF images as generally TIFF's are stored in strips or tiles. It only loads into memory the strips or tiles it needs to create the tile at the requested size.

Improving server performance

The WebImageViewer loads any image that's listed in the RegisteredDecoders.Decoders collection when invoking the Open() or OpenUrl() methods. However an image format that stores data in strips or tiles is more efficient than one that stores an image as a single block of data. That is the case with the TIFF format which we recommend for the best server performance.

Ideally the TIFF should be tiled as opposed to stripped with the tile size of the TIFF the same as the control, which defaults to 512 x 512 pixels.

The PreCacheTiles property caches tiles of the entire image prior to loading into the control. This causes a slight initial delay when loading the image and after making edits to the image, but overall performance is improved when scrolling the image. When loading large JPEG images, PreCacheTiles is considerably more efficient as at this point DotImage loads the entire JPEG image in server memory for each tile unless this property is true. TIFF's are much more efficient as DotImage has the ability to only load the surrounding image data required to save the tile.

Antialias display

The AntialiasDisplay property when set to the default of none uses Javascript resizing. For higher quality scaling, set AntialiasDisplay to ReductionOnly. When set to a value other than None, JavaScript initially resizes the tiles, but then requests a higher quality resized image from the server. The server can efficiently generate a scaled high quality version of the tile which uses Scale to Gray scaling when the source image is 1-bit.

Browser format

The friendly image format which is streamed to the browser can be set with the BrowserFormat property of the WebImageViewer control. It can be set to JPEG, PNG, GIF, or Auto (default). JPEG is best for photographic color and grayscale images. PNG or GIF are best for documents. Auto automatically selects JPEG for 24-bit and higher images, and PNG for 1-bit and grayscale images.

Scrolling behavior

WebImageViewer progressively loads tiles from the source image and streams them to the client as a PNG, JPEG, or GIF. When using ImageDisplayOrder.OnDemand, only the tiles that are needed for the current view of the image are loaded. Once the tiles are loaded, they remain in the control so scrolling back to a tile previously loaded does not cause the tile to be reloaded. This behavior results in a very efficient and presentable image viewer.

Previous versions of DotImage WebImageViewer had multiple ScrollBarStyles. The ScrollBarStyle property no longer exists and the only scrolling method is the On-Demand tiled loading.

MouseTools

The WebImageViewer provides tools that interact with the image for Zooming In, Zooming Out, Zoom to Area, Centering, Panning, and rubberband selections for the left and right mouse buttons. See the MouseTool property in the WebImageViewer.

AutoZoom

The image zoom can be set to automatically zoom the image based on the image size, or to fit the control to the image size. Set the AutoZoom property of the WebImageViewer to None, BestFit, BestFitShrinkOnly, FitToWidth, FitToHeight, or FitToImage.

Rubberbanding in ASP.NET

To draw with the Rubberband using post back:

1. Set the AutoPostBack property of the WebImageViewer's Selection, to true.
2. Create a new event handler for the WebImageViewer's SelectionChanged event, similar to the method shown in the example below.

The following example demonstrates how to draw a rectangle using the RubberBand Selection MouseTool.

C#

```
Canvas myCanvas = new Canvas(this.WebImageViewer1.Image);
Rectangle mySelection = this.WebImageViewer1.Selection.Rectangle;

// Draws a black rectangle, with a semi-transparent orange fill
myCanvas.DrawRectangle(mySelection, new AtalaPen(Color.Black),
new SolidFill(Color.FromArgb(128, Color.DarkOrange)));
// Resets the Selection so it's no longer there
this.WebImageViewer1.Selection.Reset();
// Notifies the control that the Image was modified
this.WebImageViewer1.Update();
```

Image cache

The WebImageViewer creates temporary cache images in the folder specified in the `Web.Config` configuration setting, AtalasoftWebControls_Cache.

This folder must be set with Modify and Write permissions for the control to work properly.

How the cache works

The WebImageViewer saves the current Image to the cache if any of the following are true:

1. The image was created from a stream or assigned directly to the Image object.
2. The image was opened from a mapped path (ie, c:\myImage.jpg).
3. The image has been modified by using ApplyCommand.
4. The Update() method has been called.

When scrolling the image, the Image is streamed directly to the browser.

Due to the nature of session state and components, it is difficult to notify ASP.NET when the browser closes, or when the session expires within a component.

To delete the old files, the control checks the cache for old files the first time it is loaded in a new session. If the control is never run again, the files are never deleted. Additionally, they are not deleted within a current SessionID. However if the browser is open for longer than the

CacheLifeTime, and another session is created while the browser is not active, the cached images are deleted.

The control checks the .stamp (date time stamp) file's time created stamp, and if the present time is past the the file's time stamp + CacheLifeTime, it deletes all files that begin with that new SessionID. You may take advantage of this cache for custom purposes by prepending an image file with SessionID and saving it to the cache.

Server performance

Often a deployment requires multiple servers for a high-traffic applications. In this case, dedicate one server for the image cache. Configure each server with a virtual directory that points to this one cache server.

`Web.config` parameters

Setting	Description	Note
AtalasoftWebControls_Cache	Location of the cache relative to the root of the application	
AtalasoftWebControls_CacheFilesOnly	Only set to **true** if files should be put in the cache. Set to **false** if it is OK to create directories. Default is **true**.	Setting this value to false while using .NET can cause the application to recompile while files are deleted from the cache. Use the default setting for .NET.
AtalasoftWebControls_CacheLifeTime	Number of minutes a file can live in the cache without being used. Default is **20**.	Set value to the length of time that a session is allowed to stay alive before it times out.
AtalasoftWebControls_DisableCache	Set to true to disable the image cache. The default is "false".	Use this option only if you are never going to need to modify images, load them from a direct file path, or you are saving them yourself. This turns off warnings for the Cache, and may render the control unusable. Use at your own risk.
AtalasoftWebControls_ErrorLogging	Set to **true** to enable the error log. The default is **false**.	Log files will be placed in the Cache alongside images and time stamps for the same session. These files are automatically cleaned up when the session is cleaned up.
AtalasoftWebControls_ResourcePath	Set the location of Javascript and Image resources required by the control.	Before version 4.0, this was a property of the control. You must now set in the `web.config`. If it is not set, resources are requested from the control assembly, which is slower.

Setting	Description	Note
AtalasoftWebControls_ShowClientErrors	Set to **true** to show JavaScript errors in the browser. Otherwise, set to **false**. The default is **true**.	

Example

```
<appSettings>
 <add key="AtalasoftWebControls_Cache" value="ImageCache/" />
 <add key="AtalasoftWebControls_CacheLifeTime" value="60" />
 <add key="AtalasoftWebControls_ResourcePath" value="Resources/" />
</appSettings>
```

Annotations in ASP.NET

DotImage allows you to view and edit annotations on the web. The control, WebAnnotationViewer, is the ASP.NET equivalent of the AnnotateViewer

Using the new WebAnnotationViewer and the classes in DotImage Annotations, you can:

- Read annotations in WANG or XMP format from a file and show them in a browser
- Allow the user to edit those annotations
- Allow the user to add and delete annotations
- Allow the user to edit text in a text annotation use a double-click

Add the control to a page

1. Open an ASP.NET page in Design view.
2. Drag a WebAnnotationViewer from the Toolbox to the page.

Set up annotation defaults

This procedure sets up the defaults that should be used when the built-in annotations are created by the user.

1. Create a method called InitializeDefaultAnnotations().
2. Create AnnotationData objects that you want to use as defaults.
3. Set the name property on each default AnnotationData object, and give them each a unique name such as RedEllipse or BlackRect.
4. Add each AnnotationData object created above to the DefaultAnnotations property.
5. In the Page's Onlnit event handler, and before the base.OnInit call, add a call to InitializeDefaultAnnotations.

Load annotations from a file

Do one of the following:

1. On the server, call WebAnnotationViewer.LoadAnnotations(String).

   or
2. On the client, call the name of the file containing the annotations.

Allow the user to put a new annotation on the viewer

1. From JavaScript call WebImageViewer.CreateAnnotation(String) with the name of the AnnotationData class you want to allow the user to create, such as "TextData".
2. Use WebImageViewer.setInteractMode() to set the interaction mode to Author (3).
3. If you want a chance to alter the data before it is used, handle the WebAnnotationViewer.AnnotationCreated event on the server. The event args passed in will have a property called AnnotationData that can be altered.

   The user can draw this annotation until the interaction mode is set to 0 (None) or 2 (Modify).

Example
JavaScript

```
// This client side code is used with C# examples
WebAnnotationViewer1.setInteractMode(atalaAnnotationInteractionMode.Author);
WebAnnotationViewer1.CreateAnnotation('RubberStampData', 'myStampRed');
```

C#

```
protected Atalasoft.Imaging.WebControls.Annotations.WebAnnotationViewer
 WebAnnotationViewer1;

   override protected void OnInit(EventArgs e)
   {
      //
      // CODEGEN: This call is required by the ASP.NET Web Form Designer.
      //
      InitializeComponent();
      InitializeDefaultAnnotations();
      base.OnInit(e);
   }

   private void InitializeDefaultAnnotations()
   {
      RubberStampData myRubberStamp = new RubberStampData("Red Stamp!");
      myRubberStamp.Name = "myStampRed";
      myRubberStamp.FontBrush.Color = Color.Red;
      WebAnnotationViewer1.Annotations.DefaultAnnotations.Add(myRubberStamp);
   }
```

Example 2
JavaScript

```
// This client side code is used with C# examples
WebAnnotationViewer1.setInteractMode(atalaAnnotationInteractionMode.Author);
WebAnnotationViewer1.CreateAnnotation('EllipseData');
```

C#

```
protected Atalasoft.Imaging.WebControls.Annotations.WebAnnotationViewer
 WebAnnotationViewer1;

   private void WebAnnotationViewer1_AnnotationCreated(object sender,
 AnnotationCreatedEventArgs e)
   {
      AnnotationData newData = e.AnnotationData;
      EllipseData ellipse = newData as EllipseData;
      if(ellipse != null)
      {
         ellipse.Outline = new AnnotationPen(Color.Black, 3);
         ellipse.Fill = new AnnotationBrush(Color.Red);
```

```
        }
    }
```

For more information see Server-side image viewing with ASP.NET, Client-side scripting in ASP.NET, and How to draw shapes.

Thumbnails in ASP.NET

For efficient viewing of thumbnail images, use DotImage on the server side, or the WebThumbnailViewer control. The WebImageViewer control is not intended to be used for viewing thumbnails.

Create DotImage Server-side thumbnails

1. Create an ASP.NET WebForm that creates the thumbnail from a full size image or a file cached thumbnail. (You can use the Image Cache for this). Include an Image Control tag () in your WebForm page with a src attribute pointing to the thumbnail ASPX page specifying arguments in the query string as needed.

   Typically, these are file name, width, and height.

2. Create a hyperlink tag () around this Image Control tag, that points to the page that has the WebImageViewer control on it, and passes the file path in the query string to load the corresponding image that the thumbnail was created from.

   The thumbnail can also be linked to the WebImageViewer control when clicked. Create an ASP.NET WebForm that takes the given file path from the query string and loads it into the WebImageViewer control in the Page_Load event.

Client-side thumbnails using WebThumbnailViewer and WebImageViewer

Introduced in DotImage 4.0, the WebThumbnailViewer control creates thumbnails on demand from either a multipage image, or a directory file path. It uses a rich AJAX JavaScript user interface that updates dynamically as the image or WebForm changes.

1. Drag and drop a WebThumbnailViewer control and a WebImageViewer control from the toolbox onto your WebForm. (If you do not have this control in the toolbox, see Document Imaging in ASP.NET).
2. Size the controls either by dragging the grips in DesignTime, or by setting the Width and Height properties directly.
3. Decide which Property Grid Layout options you need for the WebThumbnailViewer. Your choices are Horizontal or Vertical with Vertical being the default).
4. Open the CodeBehind and use either the Open() or OpenUrl() methods to open your content.

   The following example creates thumbnails of all JPEG and TIFF files in the given directory, and uses a WebImageViewer to display them.

   Example

   Create and display thumbnails

   C#

```
protected Atalasoft.Imaging.WebControls.WebThumbnailViewer WebThumbnailViewer1;
protected Atalasoft.Imaging.WebControls.WebImageViewer WebImageViewer1;

   private void Page_Load(object sender, System.EventArgs e)
```

```
   {
      if (!Page.IsPostBack)
      {
         string[] seachPatterns = new string[]{"*.tif", "*.tiff", "*.jpg",
"*.jpeg"}

         this.WebThumbnailViewer1.Layout = Layout.Vertical;
         this.WebThumbnailViewer1.ViewerID = this.WebImageViewer1.ClientID;
         this.WebThumbnailViewer1.OpenUrl("/TestImages/", searchPatterns);
      }
   }
```

ASP.NET

Work with remote events

The WebImageViewer control provides the ability for an ASP.NET Page object to receive an event when a client side script requests a remote invocation. When JavaScript performs a RemoteInvoke, an http POST is performed to send parameters back to the server side. To get similar capabilities without the complexity of events, see the section about remotely invoking Page() methods.

Remote invoke event arguments

A handler for a Remote Invoke Event receives an object of type RemoteInvokeEventArgs. This object contains three properties: Page, Parameters, and ReturnValue.

Property	Description
Page	Object of type System.Web.UI.Page that contains the WebImageViewer that received the event
Parameters	Object of type System.Collection.Specialized.NameValueCollection which contains all parameters provided by the POST.
ReturnValue	ArrayList which is used by event handlers to pass information back. Typically the return value is a one element array list containing a string that represents the return value of the method which has been remotely invoked

Parameters

In addition to other keys provided to the WebImageViewer, there is a key with the name atala_rm. This key is associated with the name of the method requested to be invoked. To retrieve the method name from the Parameters property, do the following:

```
In addition to other keys provided to the WebImageViewer, there is a key with the name
 atala_rm. This key is associated with the name of the method requested to be invoked.
 To retrieve the method name from the Parameters property, do the following:
```

For each parameter passed in there is a key with a name that follows this pattern:

```
For each parameter passed in there is a key with a name that follows this pattern:
```

<type> is s (string), b (bool), or n (number)

<parameter number> is an integer starting from 0 that corresponds to the position of the parameter in the array passed into the JavaScript RemoteInvoke().

Parameters can be retrieved with code like this:

C#

```
int i = 0;
ArrayList params = new ArrayList();
```

```
ArrayList types = new ArrayList();
while (true)
{
   string val;
   val = eventArgs.Parameters.Get("atala_ras" + i);
   if (val != null) {
      types.Add(typeof(string));
      params.Add(val);
      i++;
      continue;
   }
   val = eventArgs.Parameters.Get("atala_ran" + i);
   if (val != null) {
      types.Add(typeof(double));
      params.Add(Convert.ChangeType(val, typeof(double)));
      i++;
      continue;
   }
   val = eventArgs.Parameters.Get("atala_rab" + i);
   if (val != null) {
      types.Add(typeof(bool));
      params.Add(Convert.ChangeType(val, typeof(bool)));
      i++;
      continue;
   }
   break;
}
```

Writing an event handler

To write a RemoteInvoke event handler, first create the method which will receive the event. This method must take an object and a RemoteInvokeEventArgs and have no return type. Such an event handler might look like the example shown below.

Example

RemoteInvoke event handler

C#

```
private void HandleRemoteInvoke(object sender, RemoteInvokeEventArgs args)
{
   // your event handling code goes here
}
```

To install the event handler, tell the WebImageViewer to add your event handler into its chain as shown in the example below.

C#

```
webImageViewer1.RemoteInvoke += new RemoteInvokeHandler(this.HandleRemoteInvoke);
```

Invoke ASP.NET page methods remotely

The WebImageViewer control provides the ability to call methods in the owning ASP.NET Page object via client side JavaScript. In addition to being able to send typed information to the Page object, the return value for the remote method is sent back to the calling JavaScript code.

Terminology

Term	Definition
Server side	Code or objects that are invoked on an ASP server
Client side	Code or objects that are invoked in a user's browser
Parameter	Value that is passed from one function or method to another
Signature	Combination of parameter types and the return type of a function or method

Preparing a server side method for remote invocation

To invoke a method remotely, it must meet the following criteria:

- The method must be a member of a Page object that contains a WebImageViewer
- The method must be public
- The method must be marked with the RemoteInvokable attribute
- Parameters of the method must be one of the following types:
  - int
  - double
  - bool
  - string
- The method must return a type that can be converted to a string. Null or no return value are also acceptable.

Example

C#

```
[Atalasoft.Imaging.WebControls.RemoteInvokable]

public bool WaterMark(int x, int y, string message) { ... }
```

Calling a method from JavaScript

To invoke a method within a server side Page object from JavaScript, the client side code must call the RemoteInvoke() method of the JavaScript object atalaWebImageViewer. The first argument is a string representing the name of the method to invoke. The second argument is an array of values that is passed to the remote method.

Example

JavaScript

```
WebImageViewer1.RemoteInvoke("WaterMark", new Array(100, 100, "Preview Only"));
```

Getting the return value from a RemoteInvoke

RemoteInvokable() methods can have a return value, as long as they return a type that can be converted to a string. Because the return value is populated asynchronously, the JavaScript WebImageViewer.RemoteInvoked event needs to be handled. An example is shown below.

JavaScript

```
WebImageViewer1.RemoteInvoked = OnRemoteInvoked;
function OnRemoteInvoked(){
   var success = WebImageViewer1.getReturnValue();
   if (success == true){
```

```
      alert('WaterMark Succeeded.');
   }
   else {
      alert('WaterMark Failed.')
   }
}
```

Parameter type conversion

JavaScript has a limited number of built-in data types that can be readily identified within a client side script. These are number, bool, and string. The JavaScript method RemoteInvoke() bundles up each parameter with information about its data type so that it can be correctly used on the server side. Server side code makes further effort to automatically distinguish between the JavaScript notion of a generic number and the .NET notion of an integer or a floating point number. If a JavaScript number arrives on the server which contains a decimal or an exponent, it will automatically be promoted to a floating-point number. Otherwise, the number is assumed to be an integer.

No attempt is made to interpret the contents of a string.

Method identification

.NET languages can define functions or methods with the same name but different signatures. These are called overloaded methods. Server side code attempts to find the version of a method that best matches the parameters passed from JavaScript. The match happens in two stages. Server side code first tries to find an exact match where each client-passed parameter type matches the server side parameter type exactly. If there are no matches, server side code then tries to find a method for which numeric parameters can be converted without loss of information.

Example

If the client side includes this remote invocation:

JavaScript

```
WebImageViewer1.RemoteInvoke("Overload", new Array(1, 2));
```

The server side has the following methods defined:

C#

```
[Atalasoft.Imaging.WebControls.RemoteInvokable]
public string Overload(int a, int b) { ... }
[Atalasoft.Imaging.WebControls.RemoteInvokable]
public string Overload(double a, double b) { ... }
```

The RemovtInvoke no longer takes two integers as parameters and matches the first method.

JavaScript

```
WebImageViewer1.RemoteInvoke("Overload", new Array(1.0, 2));
```

In this example, the RemoteInvoke matches the second method, although it is not a perfect match.

Open images from browser

This example code shows how to load a file from the browser, save it in the file cache, and load it into the control.

Example

Load file, save to cache, and load into control

C#

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
namespace DotImageWebControlsDemo
 {
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected Atalasoft.Imaging.WebControls.WebImageViewer WebImageViewer1;
        protected System.Web.UI.WebControls.Button Button1;
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.HtmlControls.HtmlInputFile File1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }
        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // codeGEN: This call is required by the ASP.NET Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.Button1.Click += new System.EventHandler(this.Button1_Click);
            this.Load += new System.EventHandler(this.Page_Load);
        }
        #endregion
        private void Button1_Click(object sender, System.EventArgs e)
        {
            string cachePath = System.Configuration.ConfigurationSettings.
                AppSettings["AtalasoftWebControls_Cache"];
            string fullPath = Page.MapPath(cachePath);
            string fileName;
            if (File1.PostedFile.FileName.Length != 0)
            {
                fileName = System.IO.Path.GetFileName(File1.PostedFile.FileName);
                try
                {
```

```
                    // Save uploaded file to server
                    File1.PostedFile.SaveAs(fullPath + fileName);
                }
                catch (Exception exc)
                {
                    Label1.Text = "Error saving file <b>" + cachePath + fileName +
                         "</b><br>" + exc.Message;
                }
                try
                {
                    // Set main viewer to the users image
                    WebImageViewer1.OpenUrl(cachePath + fileName);
                }
                catch (Exception exc)
                {
                    Label1.Text = "Error opening file <b>" + cachePath + fileName +
                    "</b><br>" + exc.Message;
                }
            }
            else
                Label1.Text = "Error: You must specify a file name.";
        }
    }
 }


{
```

HTML to run the above code

HTML

```
<form id="WebForm1" method="post" runat="server" encType="multipart/form-data">

<table id=Table1 cellspacing=0 cellpadding=0 width=300 align=center border=0>
  <tr>
    <td>
      <p align=center>
      <cc1:WebImageViewer id=WebImageViewer1 runat="server"></
cc1:WebImageViewer><br>
      <input id=File1 type=file name=File1 runat="server"><br>
      <asp:Button id=Button1 runat="server" Text="Submit"></asp:Button></p>
      <p align=center> </p>
      <p style="FONT-WEIGHT: bold; COLOR: red" align=center><asp:Label id=Label1
         runat="server"></asp:Label></p>
    </td>
  </tr>
 </table>
 </form>
```

Importing namespaces

DotImage separates functionality into logical namespaces. To avoid typing the entire namespace when referencing a DotImage class, you may use the following Imports (or using in C#) statements.

C#

```
using Atalasoft.Imaging;
using Atalasoft.Imaging.codec;
using Atalasoft.Imaging.ColorManagement;
using Atalasoft.Imaging.Drawing;
using Atalasoft.Imaging.ImageProcessing;
using Atalasoft.Imaging.ImageProcessing.Channels;
using Atalasoft.Imaging.ImageProcessing.Document;
using Atalasoft.Imaging.ImageProcessing.Effects;
```

```
using Atalasoft.Imaging.ImageProcessing.Fft;
using Atalasoft.Imaging.ImageProcessing.Filters;
using Atalasoft.Imaging.ImageProcessing.Transforms;
using Atalasoft.Imaging.Metadata;
using Atalasoft.Imaging.WebControls;
```

Display image in the current output stream

The following sections explain how to use DotImage to display an image in the current output stream.

Use the class library

Example

C#

```
workspace.Open(Server.MapPath("myimage.tiff"));
 Response.ContentType = "image/jpeg";
 workspace.Save(Response.OutputStream, new JpegEncoder(90));
```

Use WebControls

When using DotImage to display images in an ASP.NET Web application, We recommend that you use the WebImageViewer control. The namespace Atalasoft.Imaging.WebControls contains the user interface classes and controls that can be used in a WebForm application.

Add the DotImage WebControls to the toolbox

1. In the Tools menu, select Customize Toolbox on the Visual Studio .NET.
2. Make sure the .NET Framework Components tab is selected, and find the Atalasoft.DotImage.WebControls and Atalasoft.DotImage.WebControls.Annotations namespaces.
3. Select the checkboxes that correspond to the WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls.
4. Click **OK**.

WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer are included in your toolbox and you can drag and drop these controls onto your form.

File cache

The WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls require a writable folder for caching images and other data. This folder needs to be mappable from your application directory.

1. Create a directory for the image cache within your wwwroot directory ex: "/ImageCache/".

   When using ASP.NET, you can only put the cache folder inside the application folder if the AtalasoftWebControls_CacheFilesOnly `Web.config` key is set to true. This is the default behavior in DotImage 4.0 and up. If the AtalasoftWebControls_CacheFilesOnly `Web.config` key is set to false, ASP.NET will recompile and end all sessions for the application every time images are deleted from the cache.

2. Make sure this directory is writable by applications, you can do this using the IIS control panel. You may also need to grant the MACHINE\ASPNET user account Modify permissions over this folder in Windows Explorer.

   When using impersonation, you will need to make sure that every user account that is being impersonated, has Modify permissions over this folder. Using the ASPNET user account is not enough, and can cause intermittent problems.

3. Modify your `Web.config` file by adding these lines inside the `<appSettings>` tags:

   ```
   <add key="AtalasoftWebControls_Cache" value="/ImageCache/" />
    <add key="AtalasoftWebControls_CacheLifeTime" value="60" />
   ```

   These two lines control where the cache files are saved, and how many minutes they stay in there. The control will not work at all if it cannot write the images to this location.

Control the WebImageViewer behavior

After you drop the WebImageViewer control onto your form, you can modify the behavior of the WebImageViewer by changing the properties in the development environment (IDE). Many of the properties will look familiar to you because the WebImageViewer control inherits from Control. The following properties are specific to the imaging aspects of this control and can be modified in the IDE:

- Image
- AntialiasDisplay
- Centered
- Zoom
- AutoZoom

Use JavaScript with DotImage WebControls

All DotImageWebControls offer a JavaScript API so that a rich client interface can be created in the browser. Many of the properties and events available in code-behind are available in the JavaScript API under the same name.

Use the WebAnnotationViewer

The WebAnnotationViewer control inherits from the WebImageViewer control, and can be used to overlay interactive annotations over an image.

Images and resources

There are several files used in the control, that are stored within the compiled WebControls dll.

You can put these files can in a location accessible from your application to speed up the loading of the control. By default, they are installed into the `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\WebResources` directory.

To link to these files, do the following:

1. Create a directory for the resources within your wwwroot directory ex: /files/.
2. Copy all files that are in the `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin\WebResources` directory to the new one you made.
3. Set the PathToResources `web.config` AtalasoftWebControls_ResourcePath appSetting to the relative or virtual path to your /files/ directory. If you have set up the path correctly, the JavaScript and images used load from the Resources directory instead of through the assembly and startup is faster.

   For example, you can modify your `Web.config` file by adding this lines inside the `<appSettings>` tags:

   <add key="AtalasoftWebControls_ResourcePath" value="/files/" />

ℹ Previous versions of DotImageWebImageViewer had images for drawing the scrollbar. Scrollbars are now drawn by the browser and you cannot override the look and feel with images in the resource directory.

Print images in ASP.NET

Because the WebImageViewer control is server side, the entire image must be streamed to the client to print. There are two suggested methods to print an image from the server using DotImage. A third method that could be used, which is not covered here, is to use a WinForms application deployed on an ASP.NET WebForm in the browser.

JavaScript printing

The Print() method in the WebImageViewer control requires JavaScript and uses the browser's Print functionality.

This method launches a new browser window including just the image to print, then calls client side JavaScript code to invoke the self.Print() method. This causes the browser to open the print dialog. When the print dialog closes, the newly created browser window attempts to close.

Security settings and pop-up blocking software may cause the browser to ask the user if they want to allow the window to be created or closed.

PDF printing

A PDF image can be created with DotImage using the PdfEncoder and streamed to the browser to print single or multiple images from within Adobe Acrobat. This will then download the entire image to the client into the free Adobe Acrobat reader which then can control the printing process.

The following example demonstrates how to use the response stream on a postback to export the image in PDF format.

Example

Use response stream to export PDF image

C#

```
Response.Clear();
Response.ContentType = "application/pdf";

// Create a new PdfImageCollection and add your images.
PdfImageCollection col = new PdfImageCollection();

// Add all pages from a multipage TIFF.
col.Add(new PdfImage(this.WebImageViewer.Image, PdfCompressionType.Auto));

// Create the PDF.
PdfEncoder pdf = new PdfEncoder();

// Set any properties.
pdf.JpegQuality = 85;
pdf.Metadata = new PdfMetadata("Test PDF", "Atalasoft", "Testing PdfEncoder", "",
 "",
"DotImage", DateTime.Now, DateTime.Now);

// Make each image fit into an 8.5 x 11 inch page (612 x 792 @ 72 DPI).
pdf.SizeMode = PdfPageSizeMode.FitToPage;
pdf.PageSize = new Size(612, 792);
Stream pdfStream = new MemoryStream();
pdf.Save(pdfStream, col, null);
pdfStream.Seek(0, SeekOrigin.Begin); byte[] pdfBytes = new byte[pdfStream.Length];
pdfStream.Read(pdfBytes, 0 , (int)(pdfStream.Length));
Response.BinaryWrite(pdfBytes);
Response.Flush();
Response.End();
```

Stream directly to a browser

Most image processing Web applications are server side. That is the reading and manipulating of images is done using the server processor and memory. To display an image, it is streamed to the browser and is displayed in a standard HTML <IMG> tag.

When you use DotImage server side, your Web application is platform and browser independent. The client does not have to download any dependencies to use the application.

ℹ It is possible to deploy a WinForm application in an ASP.NET Web page, but that is beyond the scope of this section.

The primary object used in ASP.NET applications is the Workspace, as you can easily manipulate the image and access extended functionality. To start using DotImage in an ASP.NET application, add a reference to Atalasoft.DotImage

ℹ You do not need to add Atalasoft.DotImage.Wincontrols because we are not using WinForms

Client-side scripting in ASP.NET

The DotImage WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer ServerControls support client side scripting for manipulation without post backs. You can access nearly all properties and methods of the control via client side JavaScript.

Basic syntax

To make the transition from a server oriented control to a client oriented control easier, the client side versions of the WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls use almost identical syntax to the server side code that you would need to accomplish similar tasks on the server side.

Public functions

All public constructors and global variables that are included within WebImageViewer.js, WebThumbnailViewer.js, WebAnnotationViewer.js, ClientTools.js, and Enums.js are prefixed with the word atala. This makes it unlikely that any another components used on the same page will have a similar function names.

Note: This is provided as general information, and does not mean that every function in the API reference requires the atala prefix, the syntax provided in the API reference should be used exactly as shown.

OnPageLoad and using AtalaInitClientScript

The WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer use the window.onload event to initialize the client side components of the control. This is necessary because as the page loads, elements can move depending on how long images or objects take to load, and whether or not the size attributes of those objects are defined.

You can use atalaInitClientScript to run code in the OnLoad event. This function queues up each string sent to it and tries to execute it as JavaScript code. This allows you to execute multiple scripts OnLoad. This function is located in ClientTools.js, and is required by all of the DotImage WebControls.

Example

Here is a general example that pops up an alert when the page is done loading.

```
<!-- You will need a reference to ClientTools.js before this snippet.
  -- This reference is automatically added to the page inline with the WebControls,
  -- so placing this snippet below one of these controls will be sufficient. -->
 <script language="javascript" type="text/javascript">
atalaInitClientScript("OnPageLoad()");
function OnPageLoad()
{
  alert("Page is finished loading.");
}
</script>
```

Objects

Every WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer control on the page is accessed by the ClientID generated by ASP.NET. Therefore WebImageViewer1 on your WebForm can be accessed in JavaScript using the same name.

Placing the WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls inside a container control such as a ContentPlaceHolder or a UserControl, changes the ClientID. You may need to take this into account when accessing methods on the client side.

The example below shows how you can get the ClientID inline on an aspx page.

Example

```
<asp:content ID="Content1" contentplaceholderid="_mainContent" runat="server">
    <cc1:WebImageViewer ID="WebImageViewer1" runat="server" Width="500px"
 Height="460px" />
    <script type="text/javascript">
        var myViewer;
        atalaInitClientScript("OnPageLoad()");
        function OnPageLoad() {
            myViewer = <%=WebImageViewer1.ClientID %>;
            myViewer.RemoteInvoked = Invalidate;
        }
        // This function forces the WebImageViewer to update all visible tiles, as
        // it may not know that the image has changed on the server side.
        function Invalidate() {
            myViewer.Update();
        }
        // This function calls a RemoteInvokable function on the server side, and
        // passes in an integer indicating which page to open.
        function GoToPage(n) {
            var vals = new Array();
            vals.push(n);
            myViewer.RemoteInvoke('GoToPage', vals);
        }
    </script>
</asp:content>
```

Properties

All properties for a given object are named the same as they are in the server code (unless otherwise noted on the table below). They are prefixed with the words get and set, and require parentheses to get the return values.

The following example sets mySize to an atalaSize object representing the size of WebImageViewer1's selection and sets the width of WebImageViewer2.

Example

```
var mySize = WebImageViewer1.getSelection().getSize();
WebImageViewer2.setWidth('400px');
```

Events

All the event handlers for the WebControls can be used by creating your own function and setting the event to that function, or by using atalaEventAdd. These examples pop up an alert box every time the ScrollPosition is changed. All of the methods shown below are valid.

ℹ The ZoomChanged event handler on the WebAnnotationViewer cannot be set to without breaking the automatic annotation zooming. Method 3 is the only method that works in this situation.

Method 1

```
// First Method
atalaInitClientScript("OnPageLoad()");
function OnPageLoad()
{
  WebImageViewer1.ScrollPositionChanged = myPositionChanged;
}
function myPositionChanged()
{
  alert("ScrollPosition Changed!");
}
```

Method 2

```
// Second Method
atalaInitClientScript("OnPageLoad()");
function OnPageLoad()
{
  WebImageViewer1.ScrollPositionChanged = function(){ alert("ScrollPosition
 Changed!"); };
}
```

Method 3

```
// Third Method
// This method allows you to add multiple handlers to the same event
atalaInitClientScript("OnPageLoad()");
function OnPageLoad()
{
  atalaEventAdd(this, WebImageViewer1, 'ScrollPositionChanged', myPositionChanged);
  atalaEventAdd(this, WebImageViewer1, 'ScrollPositionChanged', myUpdate);
}
function myPositionChanged()
{
  alert("ScrollPosition Changed!");
}
function myUpdate()
{
  alert("myUpdate!");
}
```

WebImageViewer

AtalaWebImageViewer

Although this function is public, it is not intended for users to create WebImageViewers on the client side directly. The server side generated HTML is needed for this function to initialize.

Properties

Server Name	JavaScript Syntax : Return Value	Description
ClientID	ID() : string	Gets the server control identifier generated by ASP.NET.
N/A	Form() : object	Gets the form object that the Web server control is on.
AntialiasDisplay	getAntialiasDisplay() : int setAntialiasDisplay(value : int)	Gets or sets the quality of the scaled image. Expected input: AntialiasDisplayMode or int from 0-3, 0:None 1:ScaleToGray 2:ReductionOnly 3:Full
AutoZoom	getAutoZoom() : int setAutoZoom(value : int)	Gets or sets a value indicating how the image should be zoomed as the control is resized or the image size changes. This will override setZoom. For a one time zoom, see setZoomMode. Expected input: AutoZoomMode or int from 0-5, 0:None 1:BestFit 2:BestFitShrinkOnly 3:FitToWidth 4:FitToHeight 5:FitToImage
BackColor	getBackColor() : string setBackColor(value : string)	Gets or sets the background color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
BorderColor	getBorderColor() : string setBorderColor(value : string)	Gets or sets the border color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
BorderStyle	getBorderStyle() : string setBorderStyle(value : string)	Gets or sets the border style used for this Web server control. Expected input: string representing Style: '1px solid #FF9900'
BorderWidth	getBorderWidth() : int setBorderWidth(value : int)	Gets or sets the border width of the Web server control in pixels.
BrowserFormat	getBrowserFormat() : int	Gets the format of the images created for browser output. Returns an int: 0:Jpeg 1:Png 2:Gif 3:Auto
CachePath	getCachePath() : string	Gets the file path used for the file cache as specified in the `Web.config` file.

Server Name	JavaScript Syntax : Return Value	Description
Caption	getCaption() : string setCaption(value : string)	Gets or sets a string used for creating a caption under the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.
Centered	getCentered() : bool setCentered(value : bool)	Gets or sets a value indicating if the image is centered when the image is smaller than the Web server control.
N/A	getDisplayImageUrl() : string	Gets a url that will return the current viewable portion of the image. Often used for printing the current area of the image.
Font	getFont() : string setFont(value : string)	Gets or sets the font name associated with the Web server control. Expected input: string representing font name: 'Verdana'
ForeColor	getForeColor() : string setForeColor(value : string)	Gets or sets the foreground color (typically the color of the text) of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
N/A	getFrameIndex() : int	Gets the index value used to open the current image. ℹ If the image has been modified with a server side ApplyCommand, or has been opened from a method other than OpenUrl or a linked WebThumbnailViewer, this will always return 0.
Height	getHeight() : string setHeight(value : string)	Gets or sets the CSS height style property of the server control. Expected input: string representing style height: '320px'
ImageDisplayOrder	getImageDisplayOrder() : int	Gets the order that tiles are loaded. Returns an int: 0:Sequential 1:VisibleFirst 2:OnDemand

Server Name	JavaScript Syntax : Return Value	Description
Image.Size	getImageSize() : atalaSize	Gets the size, in pixels, of the current image.
N/A	getImageUrl() : string	Gets the currently loaded image url.
MouseTool	getMouseToolLeft() : int getMouseToolRight() : int setMouseTool(left: int, right: int)	Gets or sets the behavior of the mouse when interacting with the viewable area. Expected input: MouseToolType or int from 0-7, 0:None 1:Center 2:Selection 3:ZoomIn 4:ZoomOut 5:ZoomArea 6:Pan 7:PassThrough
MouseTool.Cursor	getMouseToolCursor() : int setMouseToolCursor(value : int)	Gets or sets the cursor used with the mouse tool. Expected input: MouseToolCursor or int from 0-8, 0:Auto 1:Arrow 2:Crosshair 3:Grab 4:Hand 5:Move 6:ZoomIn 7:ZoomOut 8:Custom ℹ Currently, Grab is a CSS Extension supported by Mozilla based browsers only.
MouseTool.CustomCursor	getMouseToolCustomCursor() : string setMouseToolCustomCursor(value : string)	Gets or sets a value that represents the custom CSS style used for the MouseTool. Expected input: string representing Style: 'wait' or 'url(MyCursor.cur)' ℹ Url based CSS cursor styles may not work in all browsers.
PathToResources	getPathToResources() : string	Gets the virtual path to where the image, script, and cursor files are located.
N/A	getReturnValue() : var	Gets the return value populated by the last successful RemoteInvoke.
ImagePosition	getScrollPosition() : atalaPoint setScrollPosition(value : atalaPoint)	Gets or sets the upper left image position in relation to the upper left corner of the Web server control. Negative values are expected. Expected input: atalaPoint
ScrollBarVisibility	getScrollBarVisibility() : int	Gets the visibility of the scrollbars. Returns an int: 0:Dynamic 1:None 2:Always
Selection	getSelection() : Selection	Gets the Selection object of this Web server control.

Server Name	JavaScript Syntax : Return Value	Description
TileSize	getTileSize() : atalaSize	Gets the height and width of the tiles in pixels.
TitleBar	getTitleBar() : string setTitleBar(value : string)	Gets or sets a string used for a creating a title bar above the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.
ViewPortSize	getViewPortSize() : atalaSize	Gets the size of the viewable area, excluding scroll bars.
Visibility	getVisibility() : string setVisibility(value : string)	Gets or sets a value that indicates whether the Web server control is hidden or visible on the page. Expected input: string that is either 'hidden', 'visible', or 'inherit'. ℹ 'inherit' will return VisibilityStyle.Visible on the server side.
Width	getWidth() : string setWidth(value : string)	Gets or sets the CSS width style property of the server control. Expected input: string representing style width: '200px'
Zoom	getZoom() : float setZoom(value : float)	Gets or sets the zoom level of the image in this Web server control.
ZoomInOutPercentage	getZoomInOutPercentage() : int setZoomInOutPercentage(value : int)	Gets or sets the percentage used to increase or decrease the zoom level when the zoom MouseTools are used.

Events

Server Name	JavaScript Syntax	Description
N/A	AntialiasDisplayChanged = function()	Fires when the AntialiasDisplay property has changed.
N/A	AutoZoomChanged = function()	Fires when the AutoZoom property has changed.
N/A	BackColorChanged = function()	Fires when the BackColor property has changed.
N/A	BorderColorChanged = function()	Fires when the BorderColor property has changed.

Server Name	JavaScript Syntax	Description
N/A	BorderStyleChanged = function()	Fires when the BorderStyle property has changed.
N/A	BorderWidthChanged = function()	Fires when the BorderWidth property has changed.
N/A	CaptionChanged = function()	Fires when the Caption property has changed.
CenteredChanged	CenteredChanged = function()	Fires when the Centered property has changed.
N/A	FontChanged = function()	Fires when the Font property has changed.
N/A	ForeColorChanged = function()	Fires when the ForeColor property has changed.
ImageChanged	ImageChanged = function()	Fires when the Image has changed.
N/A	ImageSizeChanged = function()	Fires when the Image changes size.
N/A	MouseToolChanged = function()	Fires when the MouseTool property has changed.
RemoteInvoked	RemoteInvoked = function()	Fires when the RemoteInvoke function is called.
ImagePositionChanged	ScrollPositionChanged = function()	Fires when the ScrollPosition has changed
SelectionChanged	SelectionChanged = function()	Fires when the Selection has changed.
N/A	SizeChanged = function()	Fires when the Size property has changed.
N/A	TitleBarChanged = function()	Fires when the TitleBar property has changed.
N/A	VisibilityChanged = function()	Fires when the Visibility property has changed.
N/A	ZoomChanged = function()	Fires when the Zoom property has changed.
N/A	ZoomInOutPercentageChanged = function()	Fires when the ZoomInOutPercentage property has changed.
Mouse Events	ℹ All of these events pertain to the underlying image only, and attempt to pass a ImageMouseEvent object to the function.	
N/A	Clicked = function(e)	Fires when the image is clicked. Passes an ImageMouseEvent object.
N/A	DoubleClicked = function(e)	Fires when the image is double clicked. Passes an ImageMouseEvent object.

Server Name	JavaScript Syntax	Description
N/A	MouseDown = function(e)	Fires when a mouse button is pressed down on the image. Passes an ImageMouseEvent object.
N/A	MouseDownLeft = function(e)	Fires when the when the left mouse button is pressed down. Passes an ImageMouseEvent object.
N/A	MouseDownRight = function(e)	Fires when the when the right mouse button is pressed down. Passes an ImageMouseEvent object.
N/A	MouseMove = function(e)	Fires when the mouse cursor moves over the image. Passes an ImageMouseEvent object.
N/A	MouseOut = function(e)	Fires when the mouse cursor leaves the image area. Passes an ImageMouseEvent object.
N/A	MouseOver = function(e)	Fires when the mouse cursor enters the image area. Passes an ImageMouseEvent object.
N/A	MouseUp = function(e)	Fires when a mouse button is released on the image. Passes an ImageMouseEvent object.
N/A	RightClicked = function(e)	Fires when the image is right clicked. Passes an ImageMouseEvent object.
Key Events	ℹ The control must have focus for these events to fire. You can programmatically set focus to this control by calling Focus(). These events also attempt to pass a ImageKeyEvent object to the function.	
N/A	KeyDown = function(e)	Fires when a key is pressed down, if the control has focus. Passes an ImageKeyEvent object.
N/A	KeyUp = function(e)	Fires when a key is released, if the control has focus. Passes an ImageKeyEvent object.
N/A	KeyPress = function(e)	Fires when a key is pressed down, and then released, if the control has focus. Passes an ImageKeyEvent object.

Methods

Server Name	JavaScript Syntax	Description
N/A	ClearImage()	Clears the image from client side only, used to return to a blank state.
N/A	Focus()	Puts focus on the WebImageViewer, used to enable key events.
N/A	Invalidate()	Forces the WebImageViewer to re-position child objects.
OpenUrl	OpenUrl(url: string) OpenUrl(url: string, index: int)	Opens an image into the WebImageViewer control from a URL or virtual path, and frame index. ℹ This function is asynchronous. If any code needs to be executed after this call, it should be placed in the ImageChanged event handler.
N/A	PauseRefresh()	Pauses UI updates for this control until ResumeRefresh is called. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
Print	Print()	Prints the current image using javascript client code.
Redo	Redo()	Re-does an undo.
Invoke	RemoteInvoke(mthd : string) RemoteInvoke(mthd : string, args : Array)	Remotely invokes a server side method with the array of arguments. ℹ This function is asynchronous. If any code needs to be executed after this call, it should be placed in the RemoteInvoked event handler.
N/A	ResumeRefresh()	Resumes paused UI updates for this control. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
N/A	ScrollBy(dx : int, dy : int)	Scrolls the viewer by the given values.

Server Name	JavaScript Syntax	Description
N/A	ScrollTo(x: int, y: int)	Scrolls the viewer to the given coordinates.
N/A	setZoomMode(value : int)	Zoom the image once, according to the given AutoZoomMode. Expected input: AutoZoomMode or int from 0-5, 0:None 1:BestFit 2:BestFitShrinkOnly 3:FitToWidth 4:FitToHeight 5:FitToImage
Undo	Undo()	Reverts the image back to the previously cached image. ℹ This function is limited to Image changes only. The number of available undos relies on the server side CacheLevels property of the control.
Update	Update()	Forces the current image to be cached and updates the viewable area.

ImageMouseEvent

This object is based on the browser's mouse event object. The following properties are in addition to the properties that are specified by each individual browser.

Usage

JavaScript

```
/* You will need a WebImageViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below the WebImageViewer control will be sufficient.
 */
Atalasoft.Utils.InitClientScript(BindImageMouseEvents);
function BindImageMouseEvents(){
   WebImageViewer1.DoubleClicked = ImageMouseEventExample;
   WebImageViewer1.RightClicked = ImageMouseEventExample;
}

// This simple example alerts the user when the image is double or right clicked.
function ImageMouseEventExample(e){
   var mousePos = Atalasoft.Utils.getMousePosition(e);
   alert(e.name + ' event fired on the image at ' + mousePos.X + 'x' + mousePos.Y);
}
```

Properties

Server Name	JavaScript Syntax : Return value	Description
N/A	name : string	The name of the event that is being fired.

Server Name	JavaScript Syntax : Return value	Description
N/A	type : string	The browser's mouse event type that this event is based on. Ex: 'mouseover'.

ImageKeyEvent

This object is based on the browser's key event object. The following properties are in addition to the properties that are specified by each individual browser (unless otherwise specified).

Usage

JavaScript

```
/* You will need a WebImageViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below the WebImageViewer control will be sufficient.
 */
Atalasoft.Utils.InitClientScript(BindImageKeyEvents);
function BindImageKeyEvents(){
   WebImageViewer1.KeyDown = ImageKeyEventExample;
   WebImageViewer1.Focus();
}

// This simple example alerts the user when a key is pressed down while thw
 WebImageViewer has focus.
function ImageKeyEventExample(e){
   alert('KeyCode: ' + e.keyCode + ' pressed down.');
}
```

Properties

Server Name	JavaScript Syntax : Return value	Description
N/A	altKey : bool	Returns true if the alt key is pressed in combination with this key event, false otherwise.
N/A	ctrlKey : bool	Returns true if the ctrl key is pressed in combination with this key event, false otherwise.
N/A	keyCode : int	The browser's specific key code for keyup or keydown typed events, returns ASCII code on keypress typed events. ℹ This property is populated by the browser, and may not be the same value across all browsers.
N/A	shiftKey : bool	Returns true if the shift key is pressed in combination with this key event, false otherwise.
N/A	name : string	The name of the event that is being fired.

Server Name	JavaScript Syntax : Return value	Description
N/A	type : string	The browser's key event type that this event is based on. Ex: 'keyup'.

Selection

The Selection object is internal to the atalaWebImageViewer object.

Server Name	JavaScript Syntax : Return value	Description
Animated	getAnimated() : bool setAnimated(value : bool)	Gets or sets a value that indicates whether the selection will animate or not.
AutoPostBack	getAutoPostBack() : bool setAutoPostBack(value : bool)	Gets or sets a value that indicates whether the selection will automatically post back when it has changed on the client side.
BackColor	getBackColor() : string setBackColor(value : string)	Gets or sets a string that represents the color behind the dashed line.
ForeColor	getForeColor() : string setForeColor(value : string)	Gets or sets a string that represents the color of the dashed line.
Height	getHeight() : int setHeight(value : int)	Gets or sets the height of the selection.
Movable	getMovable() : bool setMovable(value : bool)	Gets or sets a value that indicates whether the selection can be moved after it is drawn.
MultiColor	getMultiColor() : bool setMultiColor(value : bool)	Gets or sets a value indicating if the selection is a solid or dashed line.
Position	getPosition() : atalaPoint setPosition(value : atalaPoint)	Gets or sets the position of the selection.
Resizable	getResizable() : bool setResizable(value : bool)	Gets or sets a value that indicating whether the selection can resize after it is drawn.
ShowGrips	getShowGrips() : bool setShowGrips(value : bool)	Gets or sets a value that indicating whether the grips of the selection are visible.
ShowTooltip	getShowTooltip() : bool setShowTooltip(value : bool)	Gets or sets a value that indicating whether the tooltip is drawn while the selection is drawn.
Size	getSize() : atalaSize setSize(value : atalaSize)	Gets or sets the size of the selection.
Visibility	getVisibility() : bool setVisibility(value : string)	Gets or sets a value indicating if the selection is hidden or visible.

Server Name	JavaScript Syntax : Return value	Description
Width	getWidth() : int setWidth(value : int)	Gets or sets the width of the selection.

Events

Server Name	JavaScript Syntax	Description
N/A	Changing = function	Fires while the selection is being modified through mouse movement.
RubberBandChanged	Changed = function	Fires when the selection has finished being modified.

Web ThumbnailViewer

atalaWeb ThumbnailViewer

Although this function is public, it is not intended for users to create WebThumbnailViewers on the client side directly. The server side generated HTML is needed for this function to initialize.

Properties

Server Name	JavaScript Syntax : Return value	Description
ClientID	ID() : string	Gets the server control identifier generated by ASP.NET.
N/A	Form() : object	Gets the form object that the Web server control is on.
AllowMultiSelect	getAllowMultiSelect() : bool	Gets a value that allows multiple thumbs to be selected using the shift or ctrl keys.
AllowDragDrop	getAllowDragDrop() : bool	Gets a value that allows thumbnails to be dragged and dropped inside the bounds of the control.
AutoDragDrop	getAutoDragDrop() : bool	Gets a value that determines whether thumbnail reordering will be automatically handled on the client-side.
BackColor	getBackColor() : string setBackColor(value: string)	Gets or sets the background color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
BorderColor	getBorderColor() : string setBorderColor(value: string)	Gets or sets the inner border color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'

Server Name	JavaScript Syntax : Return value	Description
BorderStyle	getBorderStyle() : string setBorderStyle(value: string)	Gets or sets the outer border style used for this Web server control. Expected input: string representing CSS Style: '1px solid #FF9900'
BorderWidth	getBorderWidth() : int setBorderWidth(value: int)	Gets or sets the inner border width of the Web server control. Expected input: int representing the number of pixels wide.
BrowserFormat	getBrowserFormat : int	Gets the format of the images created for browser output. Returns an int: 0:Jpeg 1:Png 2:Gif 3:Auto
CachePath	getCachePath() : string	Gets the file path used for the file cache as specified in the `Web.config` file.
Caption	getCaption() : string setCaption(value: string)	Gets or sets a string used for creating a caption under the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.
Centered	getCentered() : bool setCentered(value: bool)	Gets or sets a value indicating if the image is centered when the image is smaller than the Web server control.
Columns	getColumns() : int setColumns(value: int)	Gets or sets the number of columns (0 means auto)
Count	getCount() : int	Gets the number of thumbnails.
FlowDirection	getFlowDirection() : int	Gets the FlowDirection enumeration for how the thumbnails are laid out.
Font	getFont() : string setFont(value: string)	Gets or sets the font name associated with the Web server control. Expected input: string representing font name: 'verdana'

Server Name	JavaScript Syntax : Return value	Description
ForeColor	getForeColor() : string setForeColor(value: string)	Gets or sets the foreground color (typically the color of the text) of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
Height	getHeight() : int setHeight(value: int)	Gets or sets the height of the Web server control in pixels.
ImageDisplayOrder	getImageDisplayOrder() : int	Gets the order that thumbs are loaded.
Layout	getLayout() : int	Gets the Layout set on the server.
MouseTool	getMouseToolLeft() : int getMouseToolRight() : int setMouseTool(left: int, right: int)	Gets or sets the behavior of the mouse when interacting with the viewable area. Expected input: MouseToolType or int from 0-7, 0:None 1:Center 2:Selection 3:ZoomIn 4:ZoomOut 5:ZoomArea 6:Pan 7:PassThrough
MouseTool.Cursor	getMouseToolCursor() : int setMouseToolCursor(value : int)	Gets or sets the cursor used with the mouse tool. Expected input: MouseToolCursor or int from 0-8, 0:Auto 1:Arrow 2:Crosshair 3:Grab 4:Hand 5:Move 6:ZoomIn 7:ZoomOut 8:Custom ℹ Currently, Grab is a CSS Extension supported by Mozilla based browsers only.
MouseTool.CustomCursor	getMouseToolCustomCursor() : string setMouseToolCustomCursor(value : string)	Gets or sets a value that represents the custom CSS style used for the MouseTool. Expected input: string representing Style: 'wait' or 'url(MyCursor.cur)' ℹ Url based CSS cursor styles may not work in all browsers.
PathToResources	getPathToResources() : string	Gets the virtual path to where the image, script, and cursor files are located.
Rows	getRows() : int setRows(value: int)	Gets or sets the number of rows (0 means auto)

Server Name	JavaScript Syntax : Return value	Description
ScrollBarVisibility	getScrollBarVisibility() : int setScrollBarVisibility(value : int)	Gets or sets the visibility of the scrollbars. Expected input: ScrollBarVisibility or int: 0:Dynamic 1:None 2:Always
ScrollPosition	getScrollPosition() : atalaPoint setScrollPosition(value : atalaPoint)	Gets or sets the upper left image position in relation to the upper left corner of the Web server control. Expected input: atalaPoint
SearchPattern	getSearchPattern() : string setSearchPattern(value: string)	Gets or sets a semi-colon separated list of file search patterns for loading images from a directory.
SelectedIndex	getSelectedIndex() : int	Gets the index of the selected thumbnail. If AllowMultiSelect is true, it has the index of the most recently selected thumb.
SelectedIndexes	getSelectedIndexes(): Array	Gets an array of integers which are the indexes selected in the thumbnail control. If AllowMultiSelect is false, this array will have an length of 1, and contain the same index as SeletedIndex. If AllowMultiSelect is true, it will have a list of all of the selected thumbs in the order they were selected.
N/A	getThumbCaption(value: int) : string	Gets the caption for the thumbnail at the specified index.
ThumbCaptionFormat	getThumbCaptionFormat() : string	Gets the caption format specified on the server side.
N/A	getThumbOrder() : Array	Gets an array of integers which correspond to the thumbnails in the thumbnail control. As DragDrop events occur, this array will reorder itself to correspond to the positions of the thumbnails in the control. ℹ This function is only meant to be called when AutoDragDrop is enabled. If the value is set to false, this function will throw an exception.
ThumbPadding	getThumbPadding() : int	Gets the number of pixels around each thumbnail.

Server Name	JavaScript Syntax : Return value	Description
ThumbSize	getThumbSize() : atalaSize	Gets the size of each thumbnail.
ThumbSpacing	getThumbSpacing() : int	Gets the number of pixels between thumbs.
TitleBar	getTitleBar() : string setTitleBar(value: string)	Gets or sets a string used for a creating a title bar above the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.
Url	getUrl() : string	Gets the URL representing the current image.
ViewerID	getViewerID() : string	Gets the ID of the WebImageViewer associated with this control.
ViewPortSize	getViewPortSize() : atalaSize	Gets the size of the viewable area, excluding scroll bars.
Visibility	getVisibility() : string setVisibility(value: string)	Gets or sets a value that indicates whether the Web server control is hidden or visible on the page. Expected input: string that is either 'hidden', 'visible', or 'inherit'. ℹ 'inherit' will return VisibilityStyle.Visible on the server side.
Width	getWidth() : int setWidth(value: int)	Gets or sets the CSS width style property of the server control. Expected input: string representing style width: '200px'
Zoom	getZoom() : float setZoom(value : float)	Gets or sets the zoom level of the thumbnails in relation to their original size, for this Web server control.

Events

Server Name	JavaScript Syntax	Description
N/A	BackColorChanged = function()	Fired when the BackColor property changes.
N/A	BorderColorChanged = function()	Fired when the BorderColor property changes.

Server Name	JavaScript Syntax	Description
N/A	BorderStyleChanged = function()	Fired when the BorderStyle property changes.
N/A	BorderWidthChanged = function()	Fired when the BorderWidth property changes.
N/A	CaptionChanged = function()	Fired when the Centered property changes.
CenteredChanged	CenteredChanged = function()	Fired when the Caption property changes.
N/A	CountChanged = function()	Fired when the number of thumbnails changes.
N/A	FontChanged = function()	Fired when the Font property changes.
N/A	ForeColorChanged = function()	Fired when the ForeColor property changes.
N/A	MouseToolChanged = function()	Fired when the MouseTool property has changed.
N/A	ScrollPositionChanged = function()	Fired when the scroll position changes.
N/A	SelectedIndexChanged = function()	Fired when the selected thumbnail changes.
N/A	SizeChanged = function()	Fired when the size changes.
N/A	TitleBarChanged = function()	Fired when the title bar changes.
N/A	UrlChanged = function()	Fired when the URL changes.
N/A	VisibilityChanged = function()	Fired when the Visibility property changes.
N/A	ZoomChanged = function()	Fired when the Zoom property has changed.
Thumbnail Events	ℹ These events pertain to all thumbnails, and attempt to pass a ThumbnailEvent or ThumbnailDropEvent object to the function.	
N/A	ThumbnailClicked = function(e)	Fires when the thumb is clicked. Passes a ThumbnailEvent object.
N/A	ThumbnailDoubleClicked = function(e)	Fires when the thumb is double clicked. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseDown = function(e)	Fires when a mouse button is pressed down on the thumb. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseDownLeft = function(e)	Fires when the when the left mouse button is pressed down on the thumb. Passes a ThumbnailEvent object.

Server Name	JavaScript Syntax	Description
N/A	ThumbnailMouseDownRight = function(e)	Fires when the when the right mouse button is pressed down on the thumb. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseMove = function(e)	Fires when the mouse cursor moves over the thumb. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseOut = function(e)	Fires when the mouse cursor leaves the thumb bounding box. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseOver = function(e)	Fires when the mouse cursor enters the thumb bounding box. Passes a ThumbnailEvent object.
N/A	ThumbnailMouseUp = function(e)	Fires when a mouse button is released on the thumb. Passes a ThumbnailEvent object.
N/A	ThumbnailRightClicked = function(e)	Fires when the thumb is right clicked. Passes a ThumbnailEvent object.
N/A	ThumbnailLoaded = function(e)	Fires when the image part of the thumbnail has finished loading. Passes a ThumbnailEvent object.
N/A	ThumbnailDrop = function(e)	Fires when a thumbnail has been dragged and dropped to a new location. Passes a ThumbnailDropEvent object.
N/A	ThumbnailDropServer = function(e)	Fires after the server has has been notified of a drag-drop operation. Passes a ThumbnailDropEvent object.
Key Events	ℹ The control must have focus for these events to fire. You can programmatically set focus to this control by calling Focus(). These events also attempt to pass a ThumbnailKeyEvent object to the function.	
N/A	KeyDown = function(e)	Fires when a key is pressed down, if the control has focus. Passes a ThumbnailKeyEvent object.
N/A	KeyUp = function(e)	Fires when a key is released, if the control has focus. Passes a ThumbnailKeyEvent object.

Server Name	JavaScript Syntax	Description
N/A	KeyPress = function(e)	Fires when a key is pressed down, and then released, if the control has focus. Passes a ThumbnailKeyEvent object.

Methods

Server Name	JavaScript Syntax	Description
N/A	Focus()	Puts focus on the WebThumbnailViewer, used to enable key events.
N/A	Invalidate()	Forces the WebThumbnailViewer to re-position child objects.
OpenUrl	OpenUrl(url: string) OpenUrl(dir: string, searchPattern: string)	Opens a URL. If the URL is an image, searchPattern should not be passed. If the URL is a directory, pass a searchPattern to use to match files (Use "*.*" to match all files). searchPattern expected input: string with semi-colon delimited search patterns (ex: "*.*", "*.tif;*.jpg". "*.gif")
N/A	PauseRefresh()	Pauses UI updates for this control until ResumeRefresh is called. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
N/A	ResumeRefresh()	Resumes paused UI updates for this control. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
SelectedIndex	SelectThumb(index: int)	Selects the thumbnail at the corresponding index (starting at 0)
N/A	Update()	Forces the control to update thumbnails.
N/A	UpdateThumb(index: int)	Forces the control to update the thumbnail at the corresponding index (starting at 0)

ThumbnailEvent

This object is based on the browser's mouse event object, where applicable. The following properties are in addition to the properties that are specified by each individual browser.

Usage

JavaScript

```
/* You will need a WebThumbnailViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below the WebThumbnailViewer control will be sufficient.
 */
Atalasoft.Utils.InitClientScript(BindThumbnailMouseEvents);
function BindThumbnailMouseEvents(){
   WebThumbnailViewer1.ThumbnailDoubleClicked = ThumbnailMouseEventExample;
   WebThumbnailViewer1.ThumbnailRightClicked = ThumbnailMouseEventExample;
}

// This simple example alerts the user when a thumbnail is double or right clicked.
function ThumbnailMouseEventExample(e){
   alert(e.name + ' event fired on thumbnail at index ' + e.index);
}
```

Properties

Server Name	JavaScript Syntax : Return value	Description
N/A	index : int	The zero based index of the thumbnail that this event is firing on.
N/A	name : string	The name of the event that is being fired.
N/A	type : string	The browser's event type that this event is based on, if any. Ex: 'mouseover'.

ThumbnailDropEvent

This object is based on the browser's mouse event object. The following properties are in addition to the properties that are specified by each individual browser.

Usage

JavaScript

```
/* You will need a WebThumbnailViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below the WebThumbnailViewer control will be sufficient.
 */

Atalasoft.Utils.InitClientScript(BindDropEvent);
function BindDropEvent(){
   WebThumbnailViewer1.ThumbnailDrop = ThumbnailDropped;
}

// This simple example alerts the user when a thumbnail is dragged and dropped.
function ThumbnailDropped(e){
   alert('Thumbnail dragged from index: ' + e.dragIndex + ' was dropped to index: ' +
 e.dropIndex);
```

```
}
```

Server Name	JavaScript Syntax : Return value	Description
N/A	dragIndex : int	The zero based index of the thumbnail, in relation to where it was dragged from.
N/A	dropIndex : int	The zero based index of the thumbnail, in relation to where it was dropped.
N/A	name : string	The name of the event that is being fired.

ThumbnailKeyEvent

This object is based on the browser's key event object. The following properties are in addition to the properties that are specified by each individual browser (unless otherwise specified).

Usage

JavaScript

```
/* You will need a WebThumbnailViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below the WebThumbnailViewer control will be sufficient.
 */
Atalasoft.Utils.InitClientScript(BindThumbnailKeyEvents);
function BindThumbnailKeyEvents(){
   WebThumbnailViewer1.KeyDown = ThumbnailKeyEventExample;
   WebThumbnailViewer1.Focus();
}

// Binds the up and down arrow keys to select previous and next thumbnails
// NOTE: the keyCodes used in this example were tested in Internet Explorer, FireFox,
//       Safari, and Chrome. The keyCodes for other browsers may be different.
function ThumbnailKeyEventExample(e){
   // Get the keyCode for the key that was pressed
 var myKeyCode = e.keyCode;
   var n = WebThumbnailViewer1.getSelectedIndex();
   var c = WebThumbnailViewer1.getCount();

   if (myKeyCode == 38){
      // up arrow was pressed, subtract one from SelectedIndex, and check lower bounds
      n = (n - 1 >= 0) ? n - 1 : 0;

      // select the previous thumbnail
      WebThumbnailViewer1.SelectThumb(n);

      // stop the default KeyUp action of scrolling, by returning false
 return false;
   }
   else if (myKeyCode == 40){
      // down arrow was pressed, add one to SelectedIndex, and check upper bounds
      n = (n + 1 < c) ? n + 1 : c - 1;

      // select the next thumbnail
      WebThumbnailViewer1.SelectThumb(n);

      // stop the default KeyUp action of scrolling, by returning false
```

```
 return false;
   }
}
```

Properties

Server Name	JavaScript Syntax : Return value	Description
N/A	altKey : bool	Returns true if the alt key is pressed in combination with this key event, false otherwise.
N/A	ctrlKey : bool	Returns true if the ctrl key is pressed in combination with this key event, false otherwise.
N/A	keyCode : int	The browser's specific key code for keyup or keydown typed events, returns ASCII code on keypress typed events. ℹ This property is populated by the browser, and may not be the same value across all browsers.
N/A	shiftKey : bool	Returns true if the shift key is pressed in combination with this key event, false otherwise.
N/A	name : string	The name of the event that is being fired.
N/A	type : string	The browser's key event type that this event is based on. Ex: 'keyup'.

WebAnnotationViewer

atalaWebAnnotationViewer

Although this function is public, it is not intended for users to create WebAnnotationViewers on the client side directly. The server side generated HTML is needed for this function to initialize.

Properties

Server Name	JavaScript Syntax : Return value	Description
Annotations[index]	getAnnotation(index : int): atalaAnnotation getAnnotation(layerindex : int, index : int) : atalaAnnotation	Gets the atalaAnnotation object at the given layer and item indexes, respectively.
Annotations	getAnnotations() : Array	Returns an array of atalaAnnotation objects that are on every layer in this control.
N/A	getAnnotationDataUrl() : string	Returns the url used to store the cached data file.

Server Name	JavaScript Syntax : Return value	Description
AuthorMode	getAuthorMode() : int	Gets a value that indicates how many annotations can be created with the mouse. Returns an int: 0:Unlimited 1:Single
AutoLinkThumbnailViewer	getAutoLinkThumbnailViewer() : bool	Gets a value that indicates whether this control should automatically link events with the WebThumbnailViewer that is associated with this control
CurrentLayer	getCurrentLayer() : atalaLayer	Gets or sets the current active layer in the control. This is primarily used to indicate which layer mouse events are handled on.
CurrentUserName	setCurrentLayer(value: atalaLayer) setCurrentLayer(value: int) getCurrentUserName() : string	Returns a string that represents the current user name associated with this session. ℹ This is not automatically populated, and as such, the CurrentUserName property must be set on the server side.
InteractMode	getInteractMode(): int setInteractMode(value : int)	Expected input: int from 0-2, 0:None 1:View 2:Edit
Annotations.Layers	getLayers() : Array	Returns an array of atalaLayer objects contained in this control.
N/A	getSelectedAnnotations() : Array	Returns an array of atalaAnnotation objects that have the Selected property set to true.

Properties inherited from atalaWebImageViewer

Server Name	JavaScript Syntax : Return value	Description
ClientID	ID() : string	Gets the server control identifier generated by ASP.NET.
N/A	Form() : object	Gets the form object that the Web server control is on.
AntialiasDisplay	getAntialiasDisplay() : int setAntialiasDisplay(value : int)	Gets or sets the quality of the scaled image. Expected input: AntialiasDisplayMode or int from 0-3, 0:None 1:ScaleToGray 2:ReductionOnly 3:Full

Server Name	JavaScript Syntax : Return value	Description
AutoZoom	getAutoZoom() : int setAutoZoom(value : int)	Gets or sets a value indicating how the image should be zoomed as the control is resized or the image size changes. This will override setZoom. For a one time zoom, see setZoomMode. Expected input: AutoZoomMode or int from 0-5, 0:None 1:BestFit 2:BestFitShrinkOnly 3:FitToWidth 4:FitToHeight 5:FitToImage
BackColor	getBackColor() : string setBackColor(value : string)	Gets or sets the background color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
BorderColor	getBorderColor() : string setBorderColor(value : string)	Gets or sets the border color of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
BorderStyle	getBorderStyle() : string setBorderStyle(value : string)	Gets or sets the border style used for this Web server control. Expected input: string representing Style: '1px solid #FF9900'
BorderWidth	getBorderWidth() : int setBorderWidth(value : int)	Gets or sets the border width of the Web server control in pixels.
BrowserFormat	getBrowserFormat() : int	Gets the format of the images created for browser output. Returns an int: 0:Jpeg 1:Png 2:Gif 3:Auto
CachePath	getCachePath() : string	Gets the file path used for the file cache as specified in the `Web.config` file.
Caption	getCaption() : string setCaption(value : string)	Gets or sets a string used for creating a caption under the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.

Server Name	JavaScript Syntax : Return value	Description
Centered	getCentered() : bool setCentered(value : bool)	Gets or sets a value indicating if the image is centered when the image is smaller than the Web server control.
N/A	getDisplayImageUrl() : string	Gets a url that will return the current viewable portion of the image. Often used for printing the current area of the image.
Font	getFont() : string setFont(value : string)	Gets or sets the font name associated with the Web server control. Expected input: string representing font name: 'Verdana'
ForeColor	getForeColor() : string setForeColor(value : string)	Gets or sets the foreground color (typically the color of the text) of the Web server control. Expected input: string representing a hex value or exact color name: '#ACFF00'
N/A	getFrameIndex() : int	Gets the index value used to open the current image. If the image has been modified with a server side ApplyCommand, or has been opened from a method other than OpenUrl or a linked WebThumbnailViewer, this will always return 0.
Height	getHeight() : string setHeight(value : string)	Gets or sets the CSS height style property of the server control. Expected input: string representing style height: '320px'
ImageDisplayOrder	getImageDisplayOrder() : int	Gets the order that tiles are loaded. Returns an int: 0:Sequential 1:VisibleFirst 2:OnDemand
Image.Size	getImageSize() : atalaSize	Gets the size, in pixels, of the current image.
N/A	getImageUrl() : string	Gets the currently loaded image url.

Server Name	JavaScript Syntax : Return value	Description
MouseTool	getMouseToolLeft() : int getMouseToolRight() : int setMouseTool(left: int, right: int)	Gets or sets the behavior of the mouse when interacting with the viewable area. Expected input: MouseToolType or int from 0-7, 0:None 1:Center 2:Selection 3:ZoomIn 4:ZoomOut 5:ZoomArea 6:Pan 7:PassThrough
MouseTool.Cursor	getMouseToolCursor() : int setMouseToolCursor(value : int)	Gets or sets the cursor used with the mouse tool. Expected input: MouseToolCursor or int from 0-8, 0:Auto 1:Arrow 2:Crosshair 3:Grab 4:Hand 5:Move 6:ZoomIn 7:ZoomOut 8:Custom ℹ Currently, Grab is a CSS Extension supported by Mozilla based browsers only.
MouseTool.CustomCursor	getMouseToolCustomCursor() : string setMouseToolCustomCursor(value : string)	Gets or sets a value that represents the custom CSS style used for the MouseTool. Expected input: string representing Style: 'wait' or 'url(MyCursor.cur)' ℹ Url based CSS cursor styles may not work in all browsers.
PathToResources	getPathToResources() : string	Gets the virtual path to where the image, script, and cursor files are located.
N/A	getReturnValue() : var	Gets the return value populated by the last successful RemoteInvoke.
ImagePosition	getScrollPosition() : atalaPoint setScrollPosition(value : atalaPoint)	Gets or sets the upper left image position in relation to the upper left corner of the Web server control. Negative values are expected. Expected input: atalaPoint
ScrollBarVisibility	getScrollBarVisibility() : int	Gets the visibility of the scrollbars. Returns an int: 0:Dynamic 1:None 2:Always
Selection	getSelection() : Selection	Gets the Selection object of this Web server control.
TileSize	getTileSize() : atalaSize	Gets the height and width of the tiles in pixels.

Server Name	JavaScript Syntax : Return value	Description
TitleBar	getTitleBar() : string setTitleBar(value : string)	Gets or sets a string used for a creating a title bar above the Web server control. HTML syntax is allowed. ℹ This property is tied to a hidden input tag, so changing this value to HTML in JavaScript could cause problems if the page needs to PostBack.
ViewPortSize	getViewPortSize() : atalaSize	Gets the size of the viewable area, excluding scroll bars.
Visibility	getVisibility() : string setVisibility(value : string)	Gets or sets a value that indicates whether the Web server control is hidden or visible on the page. Expected input: string that is either 'hidden', 'visible', or 'inherit'. ℹ 'inherit' will return VisibilityStyle.Visible on the server side.
Width	getWidth() : string setWidth(value : string)	Gets or sets the CSS width style property of the server control. Expected input: string representing style width: '200px'
Zoom	getZoom() : float setZoom(value : float)	Gets or sets the zoom level of the image in this Web server control.
ZoomInOutPercentage	getViewPortSize() : atalaSize getZoomInOutPercentage() : int setZoomInOutPercentage(value : int)	Gets or sets the percentage used to increase or decrease the zoom level when the zoom MouseTools are used.

Events

Server Name	JavaScript	Syntax Description
Annotation Events	ℹ These events only pertain to the underlying atalaAnnotation object.	
N/A	AnnotationChanged = function(e)	Fires when an atalaAnnotation has changed. Passes an AnnotationEvent object.
N/A	AnnotationChanging = function(e)	Fires while an atalaAnnotation is changing. Passes an AnnotationEvent object.

Server Name	JavaScript	Syntax Description
N/A	AnnotationClicked = function(e)	Fires when an atalaAnnotation is clicked with the left mouse button. Passes an AnnotationEvent object.
N/A	AnnotationCreated = function(e)	Fires when an atalaAnnotation is created with the mouse. Passes an AnnotationEvent object.
N/A	AnnotationDoubleClicked = function(e)	Fires when an atalaAnnotation is double-clicked with the left mouse button. Passes an AnnotationEvent object.
N/A	AnnotationMouseDown = function(e)	Fires when a mouse button is pressed down on an atalaAnnotation. Passes an AnnotationEvent object.
N/A	AnnotationMouseDownLeft = function(e)	Fires when the left mouse button is pressed down on an atalaAnnotation. Passes an AnnotationEvent object.
N/A	AnnotationMouseDownRight = function(e)	Fires when the right mouse button is pressed down on an atalaAnnotation. Passes an AnnotationEvent object.
N/A	AnnotationMouseMove = function(e)	Fires when the mouse cursor moves over an atalaAnnotation. Passes an AnnotationEvent object.
N/A	AnnotationMouseOut = function(e)	Fires when the mouse cursor leaves an atalaAnnotation's area. Passes an AnnotationEvent object.
N/A	AnnotationMouseOver = function(e)	Fires when the mouse cursor enters an atalaAnnotation's area. Passes an AnnotationEvent object.
N/A	AnnotationMouseUp = function(e)	Fires when a mouse button is released on an atalaAnnotation. Passes an AnnotationEvent object.
N/A	AnnotationRightClicked = function(e)	Fires when an atalaAnnotation is right clicked. Passes an AnnotationEvent object.
N/A	AnnotationsChanged = function(e)	Fires when the z-order of atalaAnnotations has changed. Passes an AnnotationEvent object.
Layer Events	ℹ These events only pertain to the underlying atalaLayer objects.	

Server Name	JavaScript	Syntax Description
N/A	CurrentLayerChanged = function(e)	Fires when the current atalaLayer object has been changed to a different atalaLayer. Passes an AnnotationEvent object.
N/A	LayerChanged = function(e)	Fires when an atalaLayer has changed. Passes an AnnotationEvent object.
N/A	LayersChanged = function (e)	Fires when the z-order of atalaLayers has changed. Passes an AnnotationEvent object.

Events inherited from atalaWebImageViewer

Server Name	JavaScript Syntax	Description
N/A	AntialiasDisplayChanged = function()	Fires when the AntialiasDisplay property has changed.
N/A	AutoZoomChanged = function()	Fires when the AutoZoom property has changed.
N/A	BackColorChanged = function()	Fires when the BackColor property has changed.
N/A	BorderColorChanged = function()	Fires when the BorderColor property has changed.
N/A	BorderStyleChanged = function()	Fires when the BorderStyle property has changed.
N/A	BorderWidthChanged = function()	Fires when the BorderWidth property has changed.
N/A	CaptionChanged = function()	Fires when the Caption property has changed.
CenteredChanged	CenteredChanged = function()	Fires when the Centered property has changed.
N/A	FontChanged = function()	Fires when the Font property has changed.
N/A	ForeColorChanged = function()	Fires when the ForeColor property has changed.
ImageChanged	ImageChanged = function()	Fires when the Image has changed.
N/A	ImageSizeChanged = function()	Fires when the Image changes size.
N/A	MouseToolChanged = function()	Fires when the MouseTool property has changed.
RemoteInvoked	RemoteInvoked = function()	Fires when the RemoteInvoke function is called.
ImagePositionChanged	ScrollPositionChanged = function()	Fires when the ScrollPosition has changed

Server Name	JavaScript Syntax	Description
SelectionChanged	SelectionChanged = function()	Fires when the Selection has changed.
N/A	SizeChanged = function()	Fires when the Size property has changed.
N/A	TitleBarChanged = function()	Fires when the TitleBar property has changed.
N/A	VisibilityChanged = function()	Fires when the Visibility property has changed.
N/A	ZoomChanged = function()	Fires when the Zoom property has changed.
N/A	ZoomInOutPercentageChanged = function()	Fires when the ZoomInOutPercentage property has changed.
Mouse Events	ℹ All of these events pertain to the underlying image only, and attempt to pass an ImageMouseEvent object to the function.	
N/A	Clicked = function(e)	Fires when the image is clicked. Passes an ImageMouseEvent object.
N/A	DoubleClicked = function(e)	Fires when the image is double clicked. Passes an ImageMouseEvent object.
N/A	MouseDown = function(e)	Fires when a mouse button is pressed down on the image. Passes an ImageMouseEvent object.
N/A	MouseDownLeft = function(e)	Fires when the when the left mouse button is pressed down. Passes an ImageMouseEvent object.
N/A	MouseDownRight = function(e)	Fires when the when the right mouse button is pressed down. Passes an ImageMouseEvent object.
N/A	MouseMove = function(e)	Fires when the mouse cursor moves over the image. Passes an ImageMouseEvent object.
N/A	MouseOut = function(e)	Fires when the mouse cursor leaves the image area. Passes an ImageMouseEvent object.
N/A	MouseOver = function(e)	Fires when the mouse cursor enters the image area. Passes an ImageMouseEvent object.

Server Name	JavaScript Syntax	Description
N/A	MouseUp = function(e)	Fires when a mouse button is released on the image. Passes an ImageMouseEvent object.
N/A	RightClicked = function(e)	Fires when the image is right clicked. Passes an ImageMouseEvent object.
Key Events	ℹ The control must have focus for these events to fire. You can programmatically set focus to this control by calling Focus(). These events also attempt to pass a ImageKeyEvent object to the function.	
N/A	KeyDown = function(e)	Fires when a key is pressed down, if the control has focus. Passes an ImageKeyEvent object.
N/A	KeyUp = function(e)	Fires when a key is released, if the control has focus. Passes an ImageKeyEvent object.
N/A	KeyPress = function(e)	Fires when a key is pressed down, and then released, if the control has focus. Passes an ImageKeyEvent object.

Methods

Server Name	JavaScript Syntax : Return value	Description
Annotations.Layers.Add	AddLayer(layer: atalaLayer) : bool	Adds the given atalaLayer to this control. Returns true on success, false otherwise.
N/A	Clear()	Clears all annotations, all layers, and the image contained in this control.
Annotations.Layers.Clear	ClearAnnotations()	Clears all annotations and all layers contained in this control.
N/A	CountAnnotations() : int	Returns an int of the total count of all annotations on all layers in this control.
Annotations.Layers.Count	CountLayers() : int	Returns an int of the count of all layers in this control.

Server Name	JavaScript Syntax : Return value	Description
N/A	CreateAnnotation(type: string) : atalaAnnotation CreateAnnotation(type: string, name: string) : atalaAnnotation	Creates and returns an atalaAnnotation from the object type string specified. The string must represent an object type that inherits from Atalasoft.Annotate.AnnotationData on the server side. Expected input: string representing AnnotationData: 'EllipseData' or 'RectangleData', string representing the name of a DefaultAnnotation created on the server side. ℹ The atalaAnnotation returned must be added to an atalaLayer by using InsertAnnotation or it will be created by drawing it with the mouse (if the current AnnotationInteractMode allows)
N/A	CancelCreateAnnotation()	Aborts the creation of an annotation without leaving a partial annotation on the existing page.
N/A	CreateLayer() : atalaLayer	Creates and returns an atalaLayer object. ℹ The atalaLayer returned must be added using AddLayer or it (including child annotations) will not be visible.
N/A	DeleteAnnotation(zindex: int) : bool	Deletes the atalaAnnotation at the given z-index from the CurrentLayer. Returns true on success, false otherwise.
N/A	DeleteAnnotations(anns: Array) : bool	Deletes an Array of atalaAnnotations from their parent atalaLayers. Returns true on success, false otherwise.
N/A	DeselectAll()	Sets the Selected property of all atalaAnnotations on all atalaLayers to false, and hides their grips.

Server Name	JavaScript Syntax : Return value	Description
Annotations.CurrentLayer.Insert	InsertAnnotation(ann: atalaAnnotation, zindex: int) : bool	Inserts the given atalaAnnotation into the CurrentLayer, at the given z-index, and automatically removes it from it's previous parent object (if any). Returns true on success, false otherwise.
LoadAnnotationDataUrl	LoadAnnotations(url: string)	Loads annotation data from a URL or virtual path.
N/A	SelectAll()	Sets the Selected property of all atalaAnnotations on all atalaLayers to true, and shows their grips.

Methods inherited from atalaWebImageViewer

Server Name	JavaScript Syntax	Description
N/A	ClearImage()	Clears the image from client side only, used to return to a blank state.
N/A	Focus()	Puts focus on the WebImageViewer, used to enable key events.
N/A	Invalidate()	Forces the WebImageViewer to re-position child objects.
OpenUrl	OpenUrl(url: string) OpenUrl(url: string, index: int)	Opens an image into the WebImageViewer control from a URL or virtual path, and frame index. ℹ This function is asynchronous. If any code needs to be executed after this call, it should be placed in the ImageChanged event handler.
N/A	PauseRefresh()	Pauses UI updates for this control until ResumeRefresh is called. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
Print	Print()	Prints the current image using javascript client code.
Redo	Redo()	Re-does an undo.

Server Name	JavaScript Syntax	Description
Invoke	RemoteInvoke(mthd : string) RemoteInvoke(mthd : string, args : Array)	Remotely invokes a server side method with the array of arguments. ℹ This function is asynchronous. If any code needs to be executed after this call, it should be placed in the RemoteInvoked event handler.
N/A	ResumeRefresh()	Resumes paused UI updates for this control. Pauses and Refreshes are nestable. The update happens when an equal number of ResumeRefresh and PauseRefresh calls are made.
N/A	ScrollBy(dx : int, dy : int)	Scrolls the viewer by the given values.
N/A	ScrollTo(x: int, y: int)	Scrolls the viewer to the given coordinates.
N/A	setZoomMode(value : int)	Zoom the image once, according to the given AutoZoomMode. Expected input: AutoZoomMode or int from 0-5, 0:None 1:BestFit 2:BestFitShrinkOnly 3:FitToWidth 4:FitToHeight 5:FitToImage
Undo	Undo()	Reverts the image back to the previously cached image. ℹ This function is limited to Image changes only. The number of available undos relies on the server side CacheLevels property of the control.
Update	Update()	Forces the current image to be cached and updates the viewable area.

AnnotationEvent

This object is based on the browser's mouse event object, if it pertains to a mouse event. The following properties are in addition to the properties that are specified by each individual browser.

JavaScript

```
/* You will need a WebAnnotationViewer and a reference to ClientTools.js before this
 snippet.
 * This reference is automatically added to the page inline with the WebControls,
```

```
 * so placing this snippet below the WebAnnotationViewer control will be sufficient.
 */
Atalasoft.Utils.InitClientScript(BindAnnotationEvents);
function BindAnnotationEvents(){
   WebAnnotationViewer1.AnnotationDoubleClicked = AnnotationEventExample;
   WebAnnotationViewer1.AnnotationRightClicked = AnnotationEventExample;
}

// This simple example alerts the user when an annotation is double or right clicked.
function AnnotationEventExample(e){
   var pos = Atalasoft.Utils.getMousePosition(e);
   alert('Event fired on annotation at position: ' + pos.X + 'x' + pos.Y);
}
```

Properties

Server Name	JavaScript Syntax : Return value	Description
N/A	annotation : atalaAnnotation	The atalaAnnotation object that this event was fired from, if any.
N/A	layer : atalaLayer	Returns null if this event was fired from a layer.

AtalaLayer

This object represents the client side JavaScript version of an Atalasoft.Annotations.LayerAnnotation object.

Properties

Server Name	JavaScript Syntax : Return value	Description
LayerAnnotation.Items	getAnnotations() : Array	Returns an array of annotations that are contained in this layer.
N/A	getLayerIndex() : int	Returns an integer that corresponds with the array index of this layer.
LayerAnnotation.Visible	getVisibility() : string setVisibility(value : string)	Gets or sets a value that indicates whether the layer and child elements are hidden or visible in the control. Expected input: string that is either "hidden" or "visible"

Methods

Server Name	JavaScript Syntax : Return value	Description
LayerAnnotation.Items.Add	Add() : bool	Adds the given atalaAnnotation to the annotation array at the top most z-order. Returns true if the add succeeded, false otherwise.

Server Name	JavaScript Syntax : Return value	Description
N/A	Delete() : bool	Deletes the atalaAnnotation at the given index, and disposes of DOM objects. Returns true on success, false on failure.
LayerAnnotation.Items.Insert	Insert() : bool	Inserts the given atalaAnnotation into the annotation array at the given index. Returns true if the insert succeeded, false otherwise.

Events

Server Name	JavaScript Syntax	Description
N/A	Changed = function(e)	Fires when the layer has changed.
N/A	ItemsChanged = function(e)	Fires when child annotations have changed z-order.

AtalaAnnotation

This object represents the client side JavaScript version of an Atalasoft.Annotations.AnnotationUI object.

Properties

Server Name	JavaScript Syntax : Return value	Description
AnnotationUI.AspectRatio	getAspectRatio() : float setAspectRatio(value : float)	Gets or sets a value that indicates the aspect ratio of this annotation. A value of zero will not maintain any aspect ratio.
N/A	setEditorHtml(value : string)	Sets the innerHTML of the DOM object that is used to edit the annotation.
N/A	getEditorObject() : object	Gets the DOM object that is used to edit the annotation.
AnnotationUI.Height	getHeight() : int setHeight(value : int)	Gets or sets the height of the annotation.
AnnotationUI.Movable	getMovable() : bool setMovable(value : bool)	Gets or sets a value that indicates whether the annotation can be moved with the mouse.
	getParent() : atalaLayer	Gets the parent object of this annotation.
AnnotationUI.Position	getPosition() : atalaPoint setPosition(value : atalaPoint)	Gets or sets the position of the annotation.
	getRectangle() : atalaRectangle setRectangle(value : atalaRectangle)	Gets or sets the size and position of this annotation.

Server Name	JavaScript Syntax : Return value	Description
AnnotationUI.Resizable	getResizable() : bool setResizable(value : bool)	Gets or sets a value that indicates whether the annotation can be resized with the mouse.
	getSize() : atalaSize setSize(value : atalaSize)	Gets or sets the size of this annotation.
AnnotationUI.Selected	getSelected() : bool setSelected(value : bool)	Gets or sets a value that indicates whether the annotation is selected.
AnnotationUI.Tooltip	getToolTip() : string setToolTip(value : string)	Gets or sets the tooltip that is displayed when the mouse is moved over this annotation.
	getType() : string	Gets the AnnotationData type for this annotation.
AnnotationUI.Visible	getVisibility() : string setVisibility(value : string)	Gets or sets a value that indicates whether the annotation is hidden or visible. Expected input: string that is either "hidden" or "visible"
AnnotationUI.Width	getWidth() : int setWidth(value : int)	Gets or sets the width of the annotation.
N/A	getZIndex() : int	Gets the z-index of this annotation.

Methods

Server Name	JavaScript Syntax	Description
N/A	HideEditor()	Hides the editor for this annotation, if there is one.
N/A	ShowEditor()	Shows the editor for this annotation, if there is one.
N/A	Synchronize()	Synchronizes the annotation bounds with underlying data. This should only be used if the data object is being changed directly.
N/A	Update()	Synchronizes the client-side annotation data with the server-side annotation data, and requests a new image from the server.
N/A	Repaint()	Sends a request to the server for a new annotation image.

Events

Server Name	JavaScript Syntax	Description
N/A	Changed = function(e)	Fires when the annotation has finished being modified.

Server Name	JavaScript Syntax	Description
N/A	Changing = function(e)	Fires while the annotation is being modified through mouse movement.
N/A	Clicked = function(e)	Fires when the annotation is clicked.
N/A	DoubleClicked = function(e)	Fires when the annotation is double clicked.
N/A	MouseDown = function(e)	Fires when a mouse button is pressed down on the annotation.
N/A	MouseDownLeft = function(e)	Fires when the left mouse button is pressed down on the annotation.
N/A	MouseDownRight = function(e)	Fires when the right mouse button is pressed down on the annotation.
N/A	MouseMove = function(e)	Fires when the mouse cursor moves over the annotation.
N/A	MouseOut = function(e)	Fires when the mouse cursor leaves the annotation area.
N/A	MouseOver = function(e)	Fires when the mouse cursor enters the annotation area.
N/A	MouseUp = function(e)	Fires when a mouse button is released on the annotation.
N/A	RightClicked = function(e)	Fires when the annotation is right clicked.
N/A	Selected = function(e)	Fires when the annotation is selected.

ClientTools

ClientTools objects

The objects shown below are used to represent the System.Drawing structs that are used within the server side portion of the DotImage WebControls.

Usage

JavaScript

```
/* You will need a reference to ClientTools.js before this snippet.
* This reference is automatically added to the page, inline with the DotImage
 WebControls,
* so placing this snippet below one of these controls will be sufficient.
 /
function ClientToolsObjectsExample(){
   // Create a new atalaPoint
   var myPoint = new atalaPoint(20, 100);
   // Change the point's values
   myPoint.X = 40;
   myPoint.Y = 80;
   // Create a new atalaSize
```

```
   var mySize = new atalaSize(320, 200);
   // Change the size's values
   mySize.Width = 40;
   mySize.Height = 80;
   // Create a new atalaRectangle
   var myRect = new atalaRectangle(50, 50, 320, 200);
   // Change the rectangle's values
   myRect.X = 100;
   myRect.Y = 100;
   myRect.Width = 800;
   myRect.Height = 600;
}
```

atalaPoint

This object mimics the System.Drawing.Point in syntax, for use on the client side.

Constructor

Server Name	JavaScript Syntax	Description
N/A	atalaPoint(x : int, y : int)	Creates a new atalaPoint given the coordinates.

Properties

Server Name	JavaScript Syntax	Description
N/A	X = int	Gets or sets the X coordinate for this atalaPoint.
N/A	Y = int	Gets or sets the Y coordinate for this atalaPoint.

atalaSize

This object mimics the System.Drawing.Point in syntax, for use on the client side.

Constructor

Server Name	JavaScript Syntax	Description
N/A	atalaSize(width: int, height : int)	Creates a new atalaSize given the height and width.

Properties

Server Name	JavaScript Syntax	Description
N/A	Height = int	Gets or sets the height of this atalaSize.
N/A	Width = int	Gets or sets the width of this atalaSize.

atalaRectangle

This object mimics the System.Drawing.Point in syntax, for use on the client side.

Constructor

Server Name	JavaScript Syntax	Description
N/A	atalaRectangle(x : int, y : int, width : int, height : int)	Creates a new atalaRectangle given the height, width, and coordinates.

Properties

Server Name	JavaScript Syntax	Description
N/A	X = int	Gets or sets the X coordinate for this atalaRectangle.
N/A	Y = int	Gets or sets the Y coordinate for this atalaRectangle.
N/A	Height = int	Gets or sets the height of this atalaRectangle.
N/A	Width = int	Gets or sets the width of this atalaRectangle.

ClientTools methods

Usage

JavaScript

```
/* You will need a reference to ClientTools.js before this snippet.
* This reference is automatically added to the page inline with the WebControls,
 * so placing this snippet below one of these controls will be sufficient.
 * This example also requires a WebImageViewer to demonstrate the usage of
 atalaEventAdd.
 */ // execute this function on page load
atalaInitClientScript(ClientToolsMethodsExample);

function ClientToolsMethodsExample(){
   // Add some event handlers to ZoomChanged, although it's possible to call
 myOtherZoomEvent
 // from myZoomEvent to achieve the same outcome, this demonstrates how multiple
 functions
 // can be added to any event on WebControls.
 // Current context is usually the keyword 'this'
   Atalasoft.Event.Attach(this, WebImageViewer1, 'ZoomChanged', myZoomEvent);
   Atalasoft.Event.Attach(this, WebImageViewer1, 'ZoomChanged', myOtherZoomEvent);
   // Run offsets example
   ClientToolsOffsetExample();
   // Bind document's mouse click event to get mouse position on click
   document.onclick = myClickEvent;
}
function myZoomEvent(){
   // do something zoom related
   alert('myZoomEvent: Zoom changed to ' + WebImageViewer1.getZoom());
}
function myOtherZoomEvent(){
   // do another thing zoom related
   alert('myOtherZoomEvent: Zoom changed to ' + WebImageViewer1.getZoom());
}
function myClickEvent(e){
   // Most browsers pass in an event object (in this case 'e')
   // If the given object is null, then it's probably using 'event'
```

```
   if (!e){
      e = event;
   }
   // Gets the page based mouse position, taking scroll position into account
   var mp = Atalasoft.Utils.getMousePosition(e);
   alert('Mouse click detected at x:' + mp.X + ' y:' + mp.Y);
}
function ClientToolsOffsetExample(){
   // Even though an object of the name WebImageViewer1 already exists, it is not a DOM
   // element, it is an object of type atalaWebImageViewer. To get the actual container
   // DOM object, we use getElementById.
   var viewerDomObject = document.getElementById('WebImageViewer1');
   // Get the offset from the left side of the page.
   var x = Atalasoft.DOM.getOffsetLeft(viewerDomObject);
   // Get the offset from the top of the page.
   var y = Atalasoft.DOM.getOffsetTop(viewerDomObject);
   // Alert the current position of the WebImageViewer1 DOM element
   alert('The WebImageViewer1 DOM element is ' + x + ' pixels from the left, and ' + y
 + ' pixels from the top.');
}
```

Server Name	JavaScript Syntax : Return value	Description
N/A	Atalasoft.Event.Attach(context : object, target : object, name : string, event : function) atalaEventAdd(context : object, target : object, name : string, event : function)	Appends a given function to be executed on the target object when the event name fires within the current context. Please see above example for syntax.
N/A	Atalasoft.Utils.getMousePosition(event : object) : atalaPoint atalaGetMousePosition(event : object) : atalaPoint	Attempts to get the mouse position from the event object passed in. Actual mouse position is added to the current scroll position (if any), to get true page based position. CSS1 comparability mode is also supported. Expected input: browser created mouse event object. ℹ This method does not sanity check the input object for performance reasons.
N/A	Atalasoft.DOM.getOffsetLeft(domElement : object) : int atalaGetOffsetLeft(domElement : object) : int	Gets the number of pixels on the X-axis from the given object to the top of the DOM tree recursively.
N/A	Atalasoft.DOM.getOffsetTop(domElement : object) : int atalaGetOffsetTop(domElement : object) : int	Gets the number of pixels on the Y-axis from the given object to the top of the DOM tree recursively.
N/A	Atalasoft.Utils.InitClientScript(function : string) atalaInitClientScript(function : string)	Adds a function or script to be executed when the page has finished loading. Expected input: function or string of JavaScript.

Server Name	JavaScript Syntax : Return value	Description
N/A	Atalasoft.Utils.UrlDecode(value : string) : string atalaUrlDecode(value : string) : string	Returns a decoded string that was url encoded with atalaUrlEncode.
N/A	Atalasoft.Utils.UrlEncode(value : string) : string atalaUrlEncode(value : string) : string	Returns an encoded string, so that it can be used in a url.

Enums

Enumeration objects

The objects shown below are used to represent the enumerations that are used within the server side portion of the DotImage WebControls. These objects do not need to be instantiated.

Usage

Javascript

```
/* You will need a reference to Enums.js before this snippet.
 * This reference is automatically added to the page, inline with the WebControls,
 * so placing this snippet below one of these controls will be sufficient.
 */ function EnumsExample(){
  // Best fit the image to the size of the WebImageViewer
  WebImageViewer1.setAutoZoom(AutoZoomMode.BestFit);

  // Scale 1-bit images to 8-bit grayscale for zoom levels less than 1
  WebImageViewer1.setAntialiasDisplay(AntialiasDisplayMode.ScaleToGray);

  // Sets the mouse tool to zoom in on a left click, and zoom out on a right click
  WebImageViewer1.setMouseTool(MouseToolType.ZoomIn, MouseToolType.ZoomOut);

  // Sets the interact mode to modify for annotation editing
  WebAnnotationViewer1.setInteractMode(AnnotationInteractMode.Modify);

  // Sets the mouse cursor to a cross hair
  WebImageViewer1.setMouseToolCursor(MouseToolCursor.Crosshair);
}
```

AutoZoomMode

This object mimics the Atalasoft.Imaging.WebControls.AutoZoomMode enumeration, for use on the client side. Specifies the automatic zoom setting of the image displayed in the control.

Properties

Server Name	JavaScript Syntax	Value	Description
None	None	0	Does not change the zoom of the image based on image size.
BestFit	BestFit	1	Fits the image by sizing the width or height to best fit the control.
BestFitShrinkOnly	BestFitShrinkOnly	2	Fits the image by sizing the width or height to best fit the control.
FitToWidth	FitToWidth	3	Fits the image by sizing the width to fit the control.

Server Name	JavaScript Syntax	Value	Description
FitToHeight	FitToHeight	4	Fits the image by sizing the height to fit the control.
FitToImage	FitToImage	5	Fits the image by sizing the control to the size of the image.

AntialiasDisplayMode

This object mimics the Atalasoft.Imaging.WebControls.AntialiasDisplayMode enumeration, for use on the client side. Specifies the antialias display mode that can be set to the control.

Properties

Server Name	JavaScript Syntax	Value	Description
None	None	0	Do not antialias (fastest performance)
ScaleToGray	ScaleToGray	1	Only antialias 1-bit document images for zoom levels less then 1.
ReductionOnly	ReductionOnly	2	Antialias all pixel formats when zoom level is less then 1.
Full	Full	3	Antialias all images for all zoom levels.

MouseToolType

This object mimics the Atalasoft.Imaging.WebControls.MouseToolType enumeration, for use on the client side. Specifies the behavior of the mouse.

Properties

Server Name	JavaScript Syntax	Value	Description
None	None	0	Specify no special mouse behavior.
Center	Center	1	Clicking the mouse will center the area clicked.
Selection	Selection	2	Mouse will be set to crosshairs and can be used to drag and modify a selection rectangle on the image.
ZoomIn	ZoomIn	3	Clicking will zoom in.
ZoomOut	ZoomOut	4	Clicking will zoom out.
ZoomArea	ZoomArea	5	Zoom in on a specified area defined by dragging a rectangle onto the image.
Pan	Pan	6	Clicking and dragging the mouse will scroll the image.
PassThrough	PassThrough	7	Clicking on the image will postback the page, used for custom MouseTools.

ScrollBarVisibility

This object mimics the Atalasoft.Imaging.WebControls.ScrollBarVisibility enumeration, for use on the client side. Specifies the scroll bar visibility.

Properties

Server Name	JavaScript Syntax	Value	Description
Dynamic	Dynamic	0	Automatically show or hide the horizontal and vertical scrollbars as the control or image resizes.
None	None	1	Never show scroll bars.
Always	Always	2	Always show scroll bars.

AnnotationInteractMode

This object mimics the Atalasoft.Imaging.WebControls.AnnotationInteractMode enumeration, for use on the client side. Specifies the annotation viewer interaction mode (with the mouse)

Properties

Server Name	JavaScript Syntax	Value	Description
None	None	0	Do not allow interaction with an annotation -- pass through to viewer
Modify	Modify	1	Annotations can be selected, moved and resized using the mouse.
Author	Author	2	Annotations can be created, selected, moved and resized using the mouse.

Annotation tool

This object mimics the Atalasoft.Imaging.WebControls.AnnotationInteractMode enumeration, for use on the client side. Specifies the annotation viewer interaction mode (with the mouse)

Properties

Server Name	JavaScript Syntax	Value	Description
N/A	None	0	No annotations are created
N/A	Line	1	Line annotations are created by pressing the left mouse button down, dragging, and releasing the button
N/A	Lines	2	Multi-line annotations are created by left-clicking the mouse for each point, right clicking stops editing
N/A	Freehand	3	Freehand annotations are created by pressing the left mouse button down, dragging, and releasing the button
N/A	Polygon	4	Multi-side polygon annotations are created by left-clicking the mouse for each point, right clicking stops editing

Server Name	JavaScript Syntax	Value	Description
N/A	Ellipse	5	Ellipse annotations are created by pressing the left mouse button down, dragging, and releasing the button
N/A	Rectangle	6	Rectangle annotations are created by pressing the left mouse button down, dragging, and releasing the button

MouseToolCursor

Properties

Server Name	JavaScript Syntax	Value	Description
Auto	Auto	0	Mouse cursor is automatically set based on the MouseTool and AnnotationTool
Arrow	Arrow	1	Mouse cursor is an arrow
Crosshair	Crosshair	2	Mouse cursor is a crosshair
Grab	Grab	3	Mouse cursor is a grabbing hand
Hand	Hand	4	Mouse cursor is a hand with the index finger pointing up
Move	Move	5	Mouse cursor is a cross with arrows in all four directions
ZoomIn	ZoomIn	6	Mouse cursor is a magnifying glass with a plus
ZoomOut	ZoomOut	7	Mouse cursor is a magnifying glass with a minus
Custom	Custom	8	Mouse cursor is defined by the MouseToolCustomCursor value

Windows form control

Rubberbands and Selection

The Rubberband component in the Atalasoft.Imaging.WinControls namespace can be used to select a region of an image. The Rubberband class is an abstract class (Must Inherit in Visual Basic). DotImage contains several default implementations of this class.

These Rubberband components are available in the toolbox when you add a reference to the DotImage WinControls to your project.

Component	Description
LineRubberband	Draw lines on an image
RectangleRubberband	Draw rectangles and rounded rectangles

Component	Description
EllipseRubberband	Draw and select ellipses
RectangleSelection	Select a rectangular area or resize an existing selection (animated by default)

Use a Rubberband

Any object that is derived from Rubberband requires that you specify a Viewport as the Parent. Because the Rubberband is a component located in the toolbox, you can do this interactively on your WinForm by dropping a Rubberband onto a form that contains a Viewport (WorkspaceViewer, ImageViewer, or BitmapViewer). Then, choose the the appropriate Viewport as the Parent in the property browser.

By default, the Rubberband will not be active. An active Rubberband responds to mouse events and can be actively created and resized. An inactive Rubberband will remain displayed if visible, but cannot be removed, moved or created interactively. Only one Rubberband per Viewport can be active at one time. If two Rubberbands are set in a Viewport, activating one automatically deactivates the other.

Viewport Selection

You can set the Selection property of the Viewport control to any Rubberband. In the Form designer, you can set this property to any Rubberband that is on the form. When the MouseTool property is set to Selection, the selection Rubberband is used. If Selection is set to null (Nothing in Visual Basic), a default RectangleSelection is used. The RectangleSelection extends the Rubberband and includes cosmetic features such as animation, animation speed, and the ability to resize an existing selection.

Obtain a region

While the Rubberband has a Rectangle property that returns the rectangular coordinates of the selection, there is also a GetRegion() method that returns the region occupied by the Rubberband. This region may not always be rectangular. For example, the EllipseSelection always returns a region in the shape of an ellipse. You can then apply this region to any ImageRegionCommand.

Rubberband pen

Set the Pen property of the Rubberband to change color, width, or style.

Set the Inverted property to true to draw an XOR style Rubberband. An XOR Rubberband inverts the colors when it draws, creating an always-visible, and very efficient Rubberband.

A solid color Rubberband must invalidate the rectangular region while it is being drawn, which is not as efficient as XOR.

Aspect ratio

If the entity you are drawing or selecting needs to maintain a particular aspect ratio (the width / height of the image, for example), you can set the AspectRatio property to a value greater than 0. This forces the entity to be drawn at the desired aspect ratio closest to the actual mouse position.

Other uses of the Rubberband

You can use the Rubberband for more than just a selection. such as drawing primitives onto an image.

You can use the Rubberband to define the size and position of the primitive, and then draw the primitive onto the image.

ℹ In some cases, the DotImage Annotations SDK might be more appropriate for this task.

Print images

Atalasoft DotImage offers two printing components for advanced image printing.

- ImagePrintDocument

  prints a single or a collection of AtalaImage objects. It provides options to center the image or resize the image to fit the page, and handles all of the calculations for differences between the image and printer resolutions.
- ImageCompositePrintDocument

  is designed to print a photo composite of multiple images. It supports common photo layouts such as full page prints, 8 x 10 prints, 4 x 6 prints (3 per page), wallets, and contact sheets. Metric sizes are also available.

Use ImagePrintDocument

ImagePrintDocument is a component that can be added to the Visual Studio toolbox and dropped onto a form. There are two ways to interact with the ImagePrintDocument.

1. Set the Image, or Images property, and those image/images will be printed when calling the Print() method.
2. Handle the GetImage event (or override the OnGetImage() method) and set the image in the PrintImageEventArgs. This eliminates the requirement for all images to be in memory before printing.

   The following example demonstrates how to print a single image using the ImagePrintDocument by setting the Image Property.

   Example

   C#

   ```
   ImagePrintDocument myPrintDocument = new ImagePrintDocument();
   myPrintDocument.Image = myImage;
   myPrintDocument.Center = true;
   myPrintDocument.Print();
   ```

   Use the PrintDialog component to display a dialog allowing printer properties to be adjusted prior to printing. To do so, set the ImagePrintDocument object to the PrintDialog.Document property. Show the dialog prior to calling the PrintImageDocument.Print() method. This stores any print options set by the user into the ImagePrintDocument object. An example is shown below.

   Example

   C#

   ```
   PrintDialog myPrintDialog = new PrintDialog();
   myPrintDialog.Document = imagePrintDocument1;
   ```

```
if (myPrintDialog.ShowDialog(this) == DialogResult.OK)
{
     imagePrintDocument1.Print();
}
```

Print multiple images

To print several images at one time, any of the following options can be chosen:

- Set the Images property to an array of AtalaImage objects.
- Pass an ImageCollection containing the images to the constructor.
- Pass any number of AtataImage objects to the constructor.
- Do not set the Image or Images property. Onstead, handle the GetImage event, setting the HasMorePages property of the PrintImageEventArgs to true until all images are printed.

Calling the Print() method invokes the printing process. See the ImagePrintDocument object reference for more information and examples.

Use ImageCompositePrintDocument

ImageCompositePrintDocument differs from the ImagePrintDocument in that it is used to print image composites, or multiple images that laid out on a single or multiple pages. The Layout property controls the type of composite to print. The available sizes are listed in the table that follows.

English Sizes	Metric Sizes	Images Per Page
FullPage	FullPage	1
8" x 10"	20cm x 35cm	1
5" x 7"	13cm x 18cm	2
4" x 6"	10cm x 15cm	2
4" x 6" Best Fit	10cm x 15cm Best Fit	3
3.5" x 5"	9cm x 13cm	4
Wallet	Wallet	9
Contact Sheet	Contact Sheet	35

To use this component, the PrintImage event must be handled, and the Image property of the PrintCompositeEventArgs must be set to the AtalaImage to print. To indicate that there are more images, set the HasMorePages property to true. To give each image a caption, set the Caption property in the PrintImage event.

See the ImageCompositePrintDocument object reference for more information and examples.

Customize printing

The ImagePrintDocument can be extended in order to customize a print job. For example, to add a text watermark to every page, handle the AfterPrintPage event, then draw the appropriate text string onto the Graphics object. To control the properties of each page on a "page by page" basis, handle the PrintPage event and change the properties appropriately (for example, portrait to landscape).

Use DotImage in Winform applications

This section provide examples of how to use DotImage in WinForms applications developed with Microsoft .NET.

Although sample code is provided only for C#, DotImage works with any CLS compliant language.

When using DotImage to display images in a Windows Forms .NET application, %company% recommends using the WorkspaceViewer control. The namespace Atalasoft.Imaging.WinControls contains the user interface classes and controls that can be used in a WinForm application. For displaying a list of thumbnails, see the ThumbnailView or the FolderThumbnailView.

Add WorkspaceViewer control to the toolbox

1. On the Visual Studio .NET menu, select **Tools** > **Customize Toolbox**.
2. Make sure the .NET Framework Components tab is selected, and find the Atalasoft.Imaging.WinControls namespace.
3. Select the WorkspaceViewer control check box.
4. Click **OK**.

Control the WorkspaceViewer behavior

After dropping the WorkspaceViewer control onto a form, the behavior of the WorkspaceViewer is modified by changing the properties in the development environment (IDE). Many of the properties will look familiar as the WorkspaceViewer control inherits from System.Windows.Forms.Control. One especially useful property that is inherited from Control is Anchor. The top, right, bottom, and left sides of the control can be anchored to the form so that the WorkspaceViewer is resized with the form.

The following properties are specific to the imaging aspects of this control and can be modified in the IDE:

- Image
- Selection
- AntialiasDisplay
- Centered:Atalasoft.DotImage.WinControls
- Zoom
- AutoUpdate
- AutoZoom
- Asynchronous
- UndoLevels

These properties are documented in the object reference.

Open and save images

The following sample code opens an image and saves it as a JPEG with a quality of 90.

C#

```
using Atalasoft.Imaging.codec;
...
workspaceViewer1.Open("c:\\myimage.tiff");
workspaceViewer1.Save("c:\\myimage.jpg", new JpegEncoder(90));
```

Add image processing

To extend upon the previous example, this sample code blurs the image by passing a BlurGuassianCommand into the Workspace.ApplyCommand method.

C#

```
using Atalasoft.Imaging.codec;
...
workspaceViewer1.Open("c:\\myimage.tiff");
workspaceViewer1.ApplyCommand(new BlurGaussianCommand(20));
workspaceViewer1.Save("c:\\myimage.jpg", newJpegEncoder(90));
```

Import namespaces

DotImage separates functionality into logical namespaces. To avoid typing the entire namespace when referencing a DotImage class, you may use the following Imports (or using in C#) statements.

C#

```
[C#]
using Atalasoft.Imaging;
using Atalasoft.Imaging.Codec;
using Atalasoft.Imaging.ColorManagement;
using Atalasoft.Imaging.Drawing;
using Atalasoft.Imaging.ImageProcessing;
using Atalasoft.Imaging.ImageProcessing.Channels;
using Atalasoft.Imaging.ImageProcessing.Document;
using Atalasoft.Imaging.ImageProcessing.Effects;
using Atalasoft.Imaging.ImageProcessing.Fft;
using Atalasoft.Imaging.ImageProcessing.Filters;
using Atalasoft.Imaging.ImageProcessing.Transforms;
using Atalasoft.Imaging.Metadata;
using Atalasoft.Imaging.WinControls;
```

Display thumbnails

DotImage can be used to view thumbnail images using the ThumbnailView or FolderThumbnailView controls. These controls are available in DotImage Photo Pro and DotImage Document Imaging.

The ThumbnailView control displays a list of thumbnail images from a set of files, or in-memory AtalaImage objects. The FolderThumbnailView displays thumbnail images of images in a particular folder path.

The following ThumbnailView features are supported:

- Load an image from a file or AtalaImage with the Add() method.
- Bind the items in the ThumbnailView to a data source.
- Load thumbnails in a background thread pool by setting the Asynchronous property to true.

- Sort thumbnails on one of the ThumbViewAttribute's such as FileName, DateModified, Size, Type, and DisplayName controlled by the SortBy property.
- Control the caption displayed for each thumbnail with the DisplayText property.
- Include a background image for each thumbnail by setting the ThumbnailBackground property.
- Control the layout to display the thumbnails in a vertical or horizontal manner with the ThumbnailLayout property.
- Set the spacing between the thumbnails with the Spacing property.
- Control the margin area to provide space around all of the thumbnails by setting the Margins property.
- Enumerate through each Thumbnail in the ThumbnailView control with the Items property.
- Control the size of the thumbnails with the ThumbnailSize property.
- Set the number of lines allocated for the caption with the CaptionLines property.

Possible uses for the ThumbnailView include:

- Showing each page of a multipage TIFF, or
- Displaying thumbnails from a non-file database

The FolderThumbnailView extends the ThumbnailView and is intended for showing thumbnails of each image in a directory. It includes all features of the ThumbnailView plus those listed here:

- The ability to view thumbnails from any folder by setting the ImageFolder property.
- The ability to specify an extension filter determine which file extensions to attempt to load as a thumbnail with the ExtensionFilter property.
- The FolderThumbnailView watches for changes in the file system and automatically add or update thumbnails if files are added, removed, renamed, or modified from the current ImageFolder.
- Count the number of valid images in a specified folder with the GetFolderImageCount() method.
- Show progress information as each thumbnail is loaded with the FolderLoadProgress event which is fired for each thumbnail that is loaded from a folder.

Work with WPF images

The following instructions show how to use DotImage AtalaImageViewer in a Windows Presentation Foundation (WPF) application.

The example code that follows is written in C#; however any CLS compliant language can be used.

Add the AtalaImageViewer control to a WPF windows application

There are several possible ways to creating a WPF application. The following example uses Visual Studio and its XAML source editor to create the project.

1. Open Visual Studio and start a new WPF Windows Application project.
2. Add the following references:
   - Atalasoft.dotImage
   - Atalasoft.dotImage.Lib
   - Atalasoft.dotImage.Wpf
   - Atalasoft.Shared

3. To launch the editor, click Window1.xaml.
4. Use the form designer to set the window size, title and other common options
5. Switch to XAML source view.
6. Add the following XML namespace to the Window tag:

```
xmlns:atala="clr-namespace:Atalasoft.Imaging.Wpf;assembly=Atalasoft.dotImage.Wpf"
```

7. Inside the Grid tag, add the following code:

```
<Grid.ColumnDefinitions> <ColumnDefinition/> </Grid.ColumnDefinitions>
 <Grid.RowDefinitions> <RowDefinitionHeight="22"/> <RowDefinition/
> </Grid.RowDefinitions> <MenuGrid.Column="0"Grid.Row="0">
 <MenuItemHeader="_File"> <MenuItemHeader="_Open"Click="OnOpenFile"/
> <MenuItemHeader="_Save"Click="OnSaveFile"/> <Separator/>
 <MenuItemHeader="E_xit"Click="OnExit"/> </MenuItem> </Menu>
 <atala:AtalaImageViewer Name="Viewer"Grid.Column="0"Grid.Row="1"/>
```

8. Open the Window1.xaml.cs file and add the following file menu event handlers:
9. Build and run the application.

Use MouseTools

AtalaImageViewer has a MouseTool property which takes any class deriving from the MouseTool class. This allows you to create custom mouse tools for the viewer. The DotImage WPF component provides several commonly used tools including selection, panning, magnifier and zoom.

The following code tells the viewer to use the panning tool:

```
this.Viewer.MouseTool = new PanningMouseTool();
```

All The viewer mouse tools have common default values making it easy to switch between the tools. You may want to modify the look or behavior of a tool to better fit your application or preference. For instance, the PanningMouseTool can have two cursors; one for the normal cursor and another for a mouse down (grab) cursor.

Use ASP.NET WebForm controls

Work with remote events

The WebImageViewer control provides the ability for an ASP.NET Page object to receive an event when a client side script requests a remote invocation. When JavaScript performs a RemoteInvoke, an http POST is performed to send parameters back to the server side. To get similar capabilities without the complexity of events, see the section about remotely invoking Page() methods.

Remote Invoke Event Arguments

A handler for a Remote Invoke Event receives an object of type RemoteInvokeEventArgs. This object contains three properties: Page, Parameters, and ReturnValue.

Property	Description
Page	Object of type System.Web.UI.Page that contains the WebImageViewer that received the event.

Parameters	Object of type System.Collection.Specialized.NameValueCollection which contains all parameters provided by the POST.
ReturnValue	ArrayList which is used by event handlers to pass information back. Typically the return value is a one element array list containing a string that represents the return value of the method which has been remotely invoked.

Parameters

In addition to other keys provided to the WebImageViewer, there is a key with the name atala_rm. This key is associated with the name of the method requested to be invoked. To retrieve the method name from the Parameters property, do the following.

```
string methodName = eventArgs.Parameters.Get("atala_rm");
```

For each parameter passed in has a key with a name that follows this pattern:

```
atala_ra<type><parameter number>
```

<type> is s, b, or n, depending on whether this parameter is a string, a bool, or a number, respectively. The table that follows summarizes this relationship.

s	string
b	bool
n	number

<parameter number> is an integer starting from 0 that corresponds to the position of the parameter in the array passed into the JavaScript RemoteInvoke().

Parameters can be retrieved with code:

```
int i = 0;
ArrayList params = new ArrayList();
ArrayList types = new ArrayList();
while (true)
{
   string val;
   val = eventArgs.Parameters.Get("atala_ras" + i);
   if (val != null) {
      types.Add(typeof(string));
      params.Add(val);
      i++;
      continue;
   }
   val = eventArgs.Parameters.Get("atala_ran" + i);
   if (val != null) {
      types.Add(typeof(double));
      params.Add(Convert.ChangeType(val, typeof(double)));
      i++;
      continue;
   }
   val = eventArgs.Parameters.Get("atala_rab" + i);
   if (val != null) {
      types.Add(typeof(bool));
      params.Add(Convert.ChangeType(val, typeof(bool)));
      i++;
      continue;
   }
   break;
```

```
}
```

Write an event handler

To write a RemoteInvoke event handler, first create the method which will receive the event. This method must take an object and a RemoteInvokeEventArgs and have no return type. Such an event handler might look like the example shown below.

RemoteInvoke Event Handler

```
private void HandleRemoteInvoke(object sender, RemoteInvokeEventArgs args)
{
   // your event handling code goes here
}
```

To install the event handler, tell the WebImageViewer to add your event handler into its chain as shown in the following example.

```
webImageViewer1.RemoteInvoke += new RemoteInvokeHandler(this.HandleRemoteInvoke);
```

Remotely invoke ASP.NET page methods

The WebImageViewer control provides the ability to call methods in the owning ASP.NET Page object via client side JavaScript. In addition to being able to send typed information to the Page object, the return value for the remote method is sent back to the calling JavaScript code.

Terminology

The terms listed here are used in the documentation.

Term	Definition
Server side	Code or objects that are invoked on an ASP server
Client side	Code or objects that are invoked in a user's browser
Parameter	Value that is passed from one function or method to another
Signature	Combination of parameter types and the return type of a function or method

Prepare a server side method for remote invocation

To invoke a method remotely, it must meet the following criteria:

- The method must be a member of a Page object that contains a WebImageViewer
- The method must be public
- The method must be marked with the RemoteInvokable attribute
- Parameters of the method must be one of the following types:
  - int
  - double
  - bool
  - string
- The method must return a type that can be converted to a string. Null or no return value are also acceptable.

An example of a possible method is shown below.

```
[Atalasoft.Imaging.WebControls.RemoteInvokable]

public bool WaterMark(int x, int y, string message) { ... }
```

Call a method from JavaScript

To invoke a method within a server side Page object from JavaScript, the client side code must call the RemoteInvoke() method of the JavaScript object atalaWebImageViewer. The first argument is a string representing the name of the method to invoke. The second argument is an array of values that is passed to the remote method.

An example of a client side remote invocation is provided below.

```
WebImageViewer1.RemoteInvoke("WaterMark", new Array(100, 100, "Preview Only"));
```

Get the return value from a RemoteInvoke

RemoteInvokable() methods can have a return value, as long as they return a type that can be converted to a string. Because the return value is populated asynchronously, the JavaScript WebImageViewer.RemoteInvoked event needs to be handled. An example is shown below.

Parameter type conversion

JavaScript has a limited number of built-in data types that can be readily identified within a client side script. These are number, bool, and string. The JavaScript method RemoteInvoke() bundles up each parameter with information about its data type so that it can be correctly used on the server side. Server side code makes further effort to automatically distinguish between the JavaScript notion of a generic number and the .NET notion of an integer or a floating point number. If a JavaScript number arrives on the server which contains a decimal or an exponent, it will automatically be promoted to a floating-point number. Otherwise, the number is assumed to be an integer.

No attempt is made to interpret the contents of a string.

Method identification

.NET languages can define functions or methods with the same name but different signatures. These are called overloaded methods. Server side code attempts to find the version of a method that best matches the parameters passed from JavaScript. The match happens in two stages. Server side code first tries to find an exact match where each client-passed parameter type matches the server side parameter type exactly. If there are no matches, server side code then tries to find a method for which numeric parameters can be converted without loss of information.

Example

If the client side includes this remote invocation:

```
WebImageViewer1.RemoteInvoke("Overload", new Array(1, 2));
```

and the server side has the following methods defined:

```
[Atalasoft.Imaging.WebControls.RemoteInvokable]
public string Overload(int a, int b) { ... }
[Atalasoft.Imaging.WebControls.RemoteInvokable]
public string Overload(double a, double b) { ... }
```

then the RemoteInvoke matches the first method, since it takes two integers as parameters.

If, instead, the client side had the following remote invocation:

```
WebImageViewer1.RemoteInvoke("Overload", new Array(1.0, 2));
```

Then the RemoteInvoke matches the second method although it is not a perfect match.

Open images from a browser

This example code shows how to load a file from the browser, save it in the file cache, and load it into the control.

Load File. Save to Cache, and Load into Control

C#

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
namespace DotImageWebControlsDemo
 {
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected Atalasoft.Imaging.WebControls.WebImageViewer WebImageViewer1;
        protected System.Web.UI.WebControls.Button Button1;
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.HtmlControls.HtmlInputFile File1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }
        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // codeGEN: This call is required by the ASP.NET Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.Button1.Click += new System.EventHandler(this.Button1_Click);
            this.Load += new System.EventHandler(this.Page_Load);
        }
```

```
        #endregion
        private void Button1_Click(object sender, System.EventArgs e)
        {
            string cachePath = System.Configuration.ConfigurationSettings.
                AppSettings["AtalasoftWebControls_Cache"];
            string fullPath = Page.MapPath(cachePath);
            string fileName;
            if (File1.PostedFile.FileName.Length != 0)
            {
                fileName = System.IO.Path.GetFileName(File1.PostedFile.FileName);
                try
                {
                    // Save uploaded file to server
                    File1.PostedFile.SaveAs(fullPath + fileName);
                }
                catch (Exception exc)
                {
                    Label1.Text = "Error saving file <b>" + cachePath + fileName +
                         "</b><br>" + exc.Message;
                }
                try
                {
                    // Set main viewer to the users image
                    WebImageViewer1.OpenUrl(cachePath + fileName);
                }
                catch (Exception exc)
                {
                    Label1.Text = "Error opening file <b>" + cachePath + fileName +
                    "</b><br>" + exc.Message;
                }
            }
            else
                Label1.Text = "Error: You must specify a file name.";
        }
    }
 }


{
```

HTML to run the preceding code

```
<form id="WebForm1" method="post" runat="server" encType="multipart/form-data">

<table id=Table1 cellspacing=0 cellpadding=0 width=300 align=center border=0>
  <tr>
    <td>
      <p align=center>
      <cc1:WebImageViewer id=WebImageViewer1 runat="server"></cc1:WebImageViewer><br>
      <input id=File1 type=file name=File1 runat="server"><br>
      <asp:Button id=Button1 runat="server" Text="Submit"></asp:Button></p>
      <p align=center> </p>
      <p style="FONT-WEIGHT: bold; COLOR: red" align=center><asp:Label id=Label1
         runat="server"></asp:Label></p>
    </td>
  </tr>
 </table>
 </form>
```

Import namespaces

Atalasoft DotImage separates functionality into logical namespaces. To avoid typing the entire namespace when referencing a Atalasoft DotImage class, you may use the following Imports (or using in C#) statements.

C#

```
using Atalasoft.Imaging;
using Atalasoft.Imaging.codec;
using Atalasoft.Imaging.ColorManagement;
using Atalasoft.Imaging.Drawing;
using Atalasoft.Imaging.ImageProcessing;
using Atalasoft.Imaging.ImageProcessing.Channels;
using Atalasoft.Imaging.ImageProcessing.Document;
using Atalasoft.Imaging.ImageProcessing.Effects;
using Atalasoft.Imaging.ImageProcessing.Fft;
using Atalasoft.Imaging.ImageProcessing.Filters;
using Atalasoft.Imaging.ImageProcessing.Transforms;
using Atalasoft.Imaging.Metadata;
using Atalasoft.Imaging.WebControls;
```

Display image in the current output stream

You can use Atalasoft DotImage to display an image in the current output stream.

Use the class library

C#

```
workspace.Open(Server.MapPath("myimage.tiff"));
 Response.ContentType = "image/jpeg";
 workspace.Save(Response.OutputStream, new JpegEncoder(90));
```

WebControls

When using Atalasoft DotImage to display images in an ASP.NET Web application, we recommend that you use the WebImageViewer control. The namespace Atalasoft.Imaging.WebControls contains the user interface classes and controls that can be used in a WebForm application.

To add the Atalasoft DotImage WebControls to the toolbox:

1. Click Tools | Customize Toolbox on the Visual Studio .NET menu.
2. Make sure the .NET Framework Components tab is selected, and find the Atalasoft.DotImage.WebControls and Atalasoft.DotImage.WebControls.Annotations namespaces.
3. Select the checkboxes that correspond to the WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls.
4. Click OK.

WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer are included in your toolbox and you can drag and drop these controls onto your form.

File Cache

The WebImageViewer, WebThumbnailViewer, and WebAnnotationViewer controls require a writable folder for caching images and other data. This folder needs to be mappable from your application directory.

1. Create a directory for the image cache within your wwwroot directory ex: `/ImageCache/`.

   When using ASP.NET, you can only put the cache folder inside the application folder if the `AtalasoftWebControls_CacheFilesOnly Web.config` key is set to true. This is the default behavior in Atalasoft DotImage 4.0 and up. If the

   ```
   AtalasoftWebControls_CacheFilesOnly Web.config
   ```

   key is set to false, ASP.NET will recompile and end all sessions for the application every time images are deleted from the cache.

2. Make sure this directory is writable by applications, you can do this using the IIS control panel. You may also need to grant the MACHINE\ASPNET user account Modify permissions over this folder in Windows Explorer.

   When using impersonation, you will need to make sure that every user account that is being impersonated, has Modify permissions over this folder. Using the ASPNET user account is not enough, and can cause intermittent problems.

3. Modify your `Web.config` file by adding these lines inside the `<appSettings>` tags:

   ```
   <add key="AtalasoftWebControls_Cache" value="/ImageCache/" />
    <add key="AtalasoftWebControls_CacheLifeTime" value="60" />
   ```

   These two lines control where the cache files are saved, and how many minutes they stay in there. The control will not work at all if it cannot write the images to this location.

Control the WebImageViewer behavior

After you drop the WebImageViewer control onto your form, change the development environment (IDE) properties to modify WebImageViewer behavior. Many of the properties will look familiar to you because the WebImageViewer control inherits from Control.

The following properties are specific to the imaging aspects of this control and can be modified in the IDE:

- Image
- AntialiasDisplay
- Centered
- Zoom
- AutoZoom

JavaScript with Atalasoft DotImage WebControls

All Atalasoft DotImageWebControls offer a JavaScript API so that a rich client interface can be created in the browser. Many of the properties and events available in code-behind are available in the JavaScript API under the same name.

WebAnnotationViewer

There are several files used in the control, that are stored within the compiled `WebControls dll`.

You can put these files can in a location accessible from your application to speed up the loading of the control. By default, they are installed into the `C:\Program Files (x86)\Atalasoft \DotImage 11.4\bin\WebResources` directory.

To link to these files, do the following:

1. Create a directory for the resources within your `wwwroot directory ex: /files/`.
2. Copy all files that are in the `C:\Program Files (x86)\Atalasoft\DotImage 11.4\bin \WebResources` directory to the new one you made.
3. Set the PathToResources `web.config` AtalasoftWebControls_ResourcePath appSetting to the relative or virtual path to your `/files/` directory. If you have set up the path correctly, the JavaScript and images used load from the Resources directory instead of through the assembly and startup is faster.

   e.g. Modify your `Web.config` file by adding this lines inside the `<appSettings>` tags:

   ```
   <add key="AtalasoftWebControls_ResourcePath" value="/files/" />
   ```

   Previous versions of DotImageWebImageViewer had images for drawing the scrollbar. Scrollbars are now drawn by the browser and you cannot override the look and feel with images in the resource directory.

Image Capture

Web scanning

Web Capture Service includes a set of integrated components that can be used to easily capture-enable a website. It uses Javascript, supported by a local scanning service on the client which could be deployed either as a Windows service or a regular Windows application.

Also, Web Capture Service supports scanning in multiuser environments: MS Terminal Server and Citrix. In these environments, multiple users can work with Web Capture Service at the same time, from different Windows logon sessions with the same user experience as on a single-user machine.

The Web Capture Service SDK includes a demo Web application that can scan, upload and import documents into Kofax Capture.

See our Web Capture Service Guide for a step-by step tutorial of setting up a scanning a new scanning application and deploying it to an IIS server.

The Web Capture Service online documentation is available at https://atalasoft.github.io/web-capture-service. The offline verison can be downloaded from the public GitHub repository at https://github.com/Atalasoft/web-capture-service/tree/master/docs.

TWAIN scanning

Acquire Images

Acquisition

The Acquisition object is the primary class in DotTwain. You can drop this component onto a form after adding it to the toolbox, or you can instantiate it directly. This is the only class you need to add standard image acquisition capabilities to an application.

For greater control over the acquire process, this class contains a collection of Device objects that controls numerous properties used for the image acquisition.

TwainController

The TwainController object is a low-level TWAIN class used by the Device and Acquisition objects. By using this class you gain more direct access to TWAIN and you can use custom driver capabilities.

This class can be created as a standalone object or can be accessed through the Device.Controller property.

Only advanced users with knowledge of the TWAIN specification should use this class.

Device

The Device object provides full access to a TWAIN compatible source on the system. Use it to open a connection to the device, to get and set properties, and then to acquire one or more images. Because this class represents a system device resource, you cannot create an instance of it. You can obtain an instance to a Device object by calling ShowSelectSource, or from the Devices collection in the Acquisition object.

DeviceCollection

The DeviceCollection holds a read-only collection of Device objects which represent all of the TWAIN compatible system resources. You can obtain the system default device from this collection. A suitable device also can be found by enumerating through the collection. Should system conditions change, such as a device being unplugged, the ScanForChanges() method can be used to recreate this collection.

Document feeder control

DocumentFeeder

Many scanners have an automatic document feeder (ADF) allowing multiple images to be scanned in a single process. The DocumentFeeder class gives full control over the feeder, and can enable or disable its use.

Navigate files in a camera

FileSystem

The FileSystem object can be used to walk through the file system structure in a TWAIN compatible camera's internal storage. It also can be used to create, delete, copy and rename files and directories directly in the camera. Images are acquired from the camera by using the SetImageDataset() method in the Device object, then calling the Acquire() method of the same Device object.

Getting started with DotTwain

This section explains how to get started using DotTwain in your applications.

Add DotTwain to the toolbox

1. ON the Visual Studio .NET menu, select **Tools** > **Customize Toolbox**
2. Make sure the .NET Framework Components tab is selected, and find the **Atalasoft.Twain** namespace.
3. Select the checkbox that corresponds to the Acquisition component.
4. Click **OK**.

   Acquisition is now included in your toolbox. You can double-click the control to add it to your form.

Set application information

After you add the Acquisition control to your form, you should set the ApplicationIdentity properties which will be used by the TWAIN driver.

If you do not set the ApplicationIdentity properties, default values are used.

Setting the following properties is optional but recommended:

- Country
- Info
- Language
- Manufacturer
- ProductFamily
- ProductName
- VersionMajor
- VersionMinor

There is also a Parent property which is used by TWAIN when displaying dialogs and acquiring images. You must set the Parent property to the parent form or control in order to acquire images.

Modal acquisition

By default, the Acquire() method is asynchronous and returns before scanning is complete. This method can be made to work in a synchronous manner by using the Device object and setting its ModalAcquire property to true. The example below illustrates this technique.

C#

```
this.device = this.acquisition.Devices.Default;
this.device.ModalAcquire = true;
this.device.Acquire();
```

Set up events

You need to use events when acquiring images. When an image is acquired, the ImageAcquired event fires, providing an AcquireEventArgs object containing the image. At least, the ImageAcquired event must be handled, but it is recommended that the AcquireCanceled and AcquireFinished events also be handled.

Show the Select Source dialog

Your application should allow users to select which TWAIN device they want to use. This is accomplished by displaying the "Select Source" dialog using the ShowSelectSource() method. The code below assumes the Acquisition component is named acquisition.

C#

```
Device device = this.acquisition.ShowSelectSource();
```

Get and set properties

To get or set a device property, you must open a connection to the device using the Open() method. Whenever the Open() method is invoked, the Close() method must be invoked to close the connection. Closing a connection resets all of the device properties to their default values and therefore a device should be closed after the image or all desired properties have been acquired.

Open() and Close() only need to be used when getting or setting properties on the device.

The code below opens a connection to the device in order to retrieve the default Resolution and BitDepth values of the device, then closes the connection. This technique can be useful if you are looking for a device in the DeviceCollection with specific default properties or capabilities. See QueryCapability for more information.

Acquire an image

You can acquire an image through the Acquisition object or from a Device object. If you do not need to get or set any properties, the easiest to use the Acquire() method from the Acquisition object. This method uses the system default device. It is shown below.

C#

```
this.acquisition.Acquire();
```

You can choose to hide the device interface and/or ask that the device save the acquired image directly to file.

Once the image has been acquired, the ImageAcquired event fires and provides an AcquireEventArgs object containing the image. If you invoked the Open() method in order to set properties before the acquire, invoke the Close() method in the AcquireFinished event to close the device.

Acquire images with TWAIN

Unless your documents are already stored digitally, the first step in any document imaging application is to acquire the images via a scanner. The DotTwain Add-On to DotImage, included in some editions of DotImage or available separately, offers advanced TWAIN acquisition features. Once the document is in digital form, the image can be cleaned-up, displayed, compressed, archived, and recognized using other third party systems.

DotTwain returns System.Drawing.Bitmap images that can easily be converted to DotImage AtalaImage objects.

Acquire a selection region of the device

Some scanners allow you to select a rectangular region of the scanning bed to be acquired. If you know that you only need a specific area or page size, this can greatly increase your scanning speed.

This region is represented by the Frame property of the Device object. Alternatively you can use the ImageLayout property for the same purpose. In some cases a driver only supports one of these two approaches.

As Frame values are in Units, you need to know the value of the Units property before setting the Frame size.

The code example below assumes you have already opened a connection to the device by calling the Open() method. This example shows how to acquire a specific size while the second shows how to acquire any size and position.

Acquire a specific size

If you only need to specify a standard region size, you can use the FrameSize property as shown in the example below:

C#

```
// Make sure the FrameSize property is supported by the device.
if (this.device.QueryCapability(DeviceCapability.FrameSize, true)
{
    // Get a list of supported frames and choose the one that fits your needs.
    StaticFrameType[] frames = this.device.GetSupportedFrameSizes();
    foreach (StaticFrameType frame in frames) {
        if (frame == StaticFrameType.LetterUS)
        {
            this.device.FrameSize = frame;
            break;
        }
    }
}
```

Acquire any size and position

If you need a more control over the size and position, or if the device does not support the FrameSize property, you can attempt to set the acquisition area using the Frame property.

C#

```
C#
```

```
// Try to use Inches.
this.device.Units = UnitType.Inches;

if (this.device.Units != UnitType.Inches) return;
this.device.Frame = new System.Drawing.RectangleF(0, 0, 8.5, 11);
```

Acquire directly to a file

Some devices allow to you acquire an image and save it directly to a file, instead of returning the image data. While the overall process is simple, there are some steps you must take:

1. Check for device capabilities
2. Set filenames
3. Close the connection

Checking for device capabilities

Begin by testing the device to make sure it can save a file. If so, you then need to negotiate the type of file to save.

C#

```
// Open a connection to the device.
this.device.Open();
this.device.TransferMethod = TwainTransferMethod.TWSX_NATIVE;

// See if the device supports file transfer.
TwainTransferMethod[] methods = this.device.GetSupportedTransferMethods();
foreach (TwainTransferMethod method in methods)
{
if (method == TwainTransferMethod.TWSX_FILE2)
{
// Use TWSX_FILE2 when possible.
this.device.TransferMethod = method;
break;
}

if (method == TwainTransferMethod.TWSX_FILE)
this.device.TransferMethod = method;
}

// If it's not supported tell stop.
if (this.device.TransferMethod == TwainTransferMethod.TWSX_NATIVE)
{
// Close the connection.
this.device.Close();
MessageBox.Show("The current device does not support saving directly to a file.");
return;
}

// Find out which file types the device can save to.
SourceImageFormat[] formats = this.device.GetSupportedImageFormats();

// We want to save the image as a TIFF.
foreach (SourceImageFormat format in formats)
{
if (format == SourceImageFormat.Tiff)
{
```

```
// TIFF is supported, so set the FileFormat.
this.device.FileFormat = format;

// Now lets try to use Group4 or Group3 compression.
// We could use GetSupportedCompressionModes, but we
// will simply try setting the Compression property instead.
this.device.Compression = CompressionMode.Group4;
if (this.device.Compression != CompressionMode.Group4)
this.device.Compression = CompressionMode.Group3;

break;
}
}

// Start the acquire process, using the device's interface.
this.device.Acquire();
```

Set filenames

During the acquire process, the FileTransfer event is raised just before each file is acquired. You need to set the FileName property of the FileTransferEventArgs object passed into the event. This tells the device where the file should be saved.

C#

```
private void OnFileTransfer(object sender, FileTransferEventArgs e)
{
    e.FileName = @"C:\TwainImages\whatever.tif";
}
```

Close the connection

When all of the images have been acquired, the AcquireFinished event is raised. Close the connection here.

```
private void OnAcquireFinished(object sender, System.EventArgs e)
{
    this.device.Close();
}
```

Detect a camera device

There are times when you only want to use a camera device. Unfortunately, TWAIN does not provide a direct way of knowing what type of device is being used. You can, however, do a little investigating to pick out a camera from a scanner.

Using the QueryCapability() method, you can find out if a device supports certain features that are normally only supported by camera devices. A list of capabilities you can check is provided here:

- Flash
- Flash2
- ExposureTime
- BatteryMinutes
- BatteryPercentage
- CameraPreviewInterface
- PowerSupply

- ZoomFactor

Upload an image to a server

There may be times when you want to send an image to a server. The HttpPost class was written for this specific purpose.

Upload the image

The following example sends a single image, along with a user name and password to identify the sender, to a server.

Example

C#

```
private void UploadImage(Atalasoft.Imaging.AtalaImage image)
{
// Create an instance of HttpPost and use the default image encoder.
Atalasoft.Imaging.HttpPost post = new Atalasoft.Imaging.HttpPost();

// Add the image to the form data collection.
post.FormData.Add("image1", image, "filename.png");

// Add data to identify the user posting the image.
post.FormData.Add("username", user);
post.FormData.Add("password", password);

// Post the form data to the server and retrieve a return value.
string ret = post.PostData("http://www.website.com/postImage.aspx");
}
```

The first part of the code creates an instance of HttpPost and uses the default image format, which is PNG. The image format determines the file type of the image when saved on the server.

Then the image is added to FormDataCollection, which includes the field name, image and filename for the image. It is best to think of HttpPost as an HTML FORM object. The FormData represents the FORM elements; in this case a FILE input box. You can update multiple images by adding them to the collection; just be sure each has a unique field name.

Next, add the username and password of the person sending this image. In many cases, additional information needs to be sent with the image in order to perform a specific action on the server. In terms of an HTML FORM, this would be a TEXTBOX field.

Finally, the FormData is sent to the server using the PostData() method. The return value of PostData is a string that is sent back from the server. Normally this is used to confirm the success of the call.

Saving the image

Once the image has been sent, save it to the server using ASP.NET. If necessary, you can modify this code to store the image in a database instead.

Example

C#

```
private void Page_Load(object sender, System.EventArgs e)
{
if (Request.Files.Count == 0) return;

// Save the file to the server.
string fileName = System.IO.Path.GetFileName(Request.Files[0].FileName);
Request.Files[0].SaveAs(GetNewFileName(fileName));

// Return the path to this file.
Response.Clear();
Response.Write("success");
Response.End();
}

private string GetNewFileName(string fileName)
{
// Create a unique filename.
string path = Server.MapPath("./images") + "\\" + Session.SessionID + fileName;
return path;
}
```

In the code above, the SaveAs() method saves the image to the server. Notice that the FileName property gets the name of the posted file. This is the same value that was passed to the FormData.Add() method when you added the image.

The data written to the Response object is returned by the PostData() method. It is a good idea to use the Response.Clear() method before adding your return value. This example simply returns success

Deploy DotTwain

To distribute DotTwain along with your .NET application, you need to include `Atalasoft.DotTwain.dll` and `Atalasoft.Shared.dll` in the same folder as the assembly that references it.

Be sure that the `.dll` versions you provide match that used to compile the assembly.

Web-based deployment is not available.

ISIS scanning

DotImage ISIS is a .NET component for capturing images from scanners that use an ISIS driver. It takes advantage of the speed and stability of ISIS drivers available from most scanner manufacturers.

Supported Features

- Direct in-memory scanning
- Scanning directly to a variety of file formats (provided through ISIS drivers)
- Access to dozens of scanner property values

- Automatic Document Feeder support
- Supports custom interface creation or, use the default driver interface
- Support for saving and restoring scanner settings to a file or stream
- Barcode detection (when supported by the scanner)
- In-memory images can be returned as a .NET Bitmap or an AtalaImage
- Use the IsisController for more direct lower-level scanner control

DotImage ISIS classes

This introduces the basic classes you need to know about to gain a general understanding of DotImage ISIS.

Acquiring images

IsisAcquisition

The IsisAcquisition object is the primary class in DotImage ISIS. This component can be dropped onto a form after adding it to the toolbox, or it can be instantiated directly. To add standard image acquisition capabilities to an application, this is the only class you need. For additional control over the acquire process, this class contains a collection of IsisDevice objects that controls numerous properties that allow you to fine tune for image acquisition.

IsisController

The IsisController object is a low-level ISIS class used by the IsisDevice and IsisAcquisition objects. Using this class provides more direct access to the ISIS driver. This class can be created as a standalone object or can be accessed through the IsisDevice.Controller property.

IsisDevice

The IsisDevice object provides full access to the system's ISIS driver. Use it to connect to the device, get and set properties and acquire one or more images. You cannot create an instance of this class as it represents a system device resource. An instance to an IsisDevice object can be obtained from the IsisDeviceCollection in the IsisAcquisition object.

IsisDeviceCollection

The IsisDeviceCollection holds a read-only collection of IsisDevice objects representing all of the ISIS scanner drivers found on the system. You can obtain the system default device from this collection, or locate a suitable device by enumerating through the collection. If system conditions change, such as a device being added, use the RefreshList()method to recreate this collection.

IsisCodecManager

The IsisCodecManager class searches through ISIS file and compression drivers on the system accumulating a collection of IsisCodec objects. This is useful as many scanners will report that they do not support any file formats. DotImage ISIS can dynamically load ISIS plug-in drivers for use with the AcquireToFile() method.

Add ISIS to the toolbox

1. On the Visual Studio .NET menu, select **Tools** > **Customize Toolbox**.

2. Make sure the .**NET Framework Components** tab is selected, and find the **Atalasoft.dotImage.Isis** namespace.
3. Check the **IsisAcquisition** component checkbox.
4. Click **OK**.
   IsisAcquisition is now included in your toolbox.
5. Double-click the control to add it to your form.

Set up events

You use events to acquire images. When an image is acquired, the ImageAcquired event is raised, providing an IsisImageAcquiredEventArgs object containing the image.

You must handle the ImageAcquired event, but it is recommended that the AcquireCanceled and AcquireFinsihed events also be handled.

The following code demonstrates how the ImageAcquired event is handled:

Handling the image acquired event

C#

```
[C#]
privatevoid _acquisition_ImageAcquired(object sender, IsisImageAcquiredEventArgs e)
{
      // This event is raised for each page during an acquisition.
 if (this.picImage.Image != null) this.picImage.Image.Dispose();

      // Set the AcquiredImageType property on the IsisAcquisition or
 IsisController
      // to specify whether you receive an AtalaImage or a .NET Bitmap.
 if (e.Image != null)
      {
            this.picImage.Image = e.Image.ToBitmap();
            e.Image.Dispose();
      }
      else if (e.Bitmap != null)
      {
            this.picImage.Image = e.Bitmap;
      }

      if (e.JobSeparator) System.Diagnostics.Debug.WriteLine("Job Separator");
}
```

Show the Select Source

Your application should allow users to select which ISIS device they want to use. Accomplish this by displaying the Select Source dialog using the ShowSelectSource() method. The sample code below assumes the IsisAcquisition component is named acquisition.

Allow users to select an ISIS device

C#

```
IsisDevice dev = null;
if (this.acquisition.ShowSelectSource(this)) dev =
 this.acquisition.Devices.Default;
```

Get and set properties

To get or set a device property, use the Open() method to open a connection to the device. Whenever the Open() method is invoked, the Close() method also must be invoked to close the connection. Open() and Close() only need to be used when getting or setting device properties.

Open a device connection to set properties

C#

```
if (device.Open())
{
    device.Settings.Resolution = new Rational(200, 200);
    device.PixelFormat = IsisPixelFormat.Binary;
    device.Acquire();
    device.Close();
}
```

Acquire an image

Acquire an image through the IsisAcquisition object or from an IsisDevice object. If you do not need to get or set any properties, use the Acquire() method from the IsisAcquisition object. The images acquired are provided in the ImageAcquired event, which is raised once for each page scanned.

Acquire directly to file

If the ISIS drivers are available on the system, you can have the driver acquire directly to file instead of in memory. Do this with the IsisAcquisition or IsisDevice objects using the AcquireToFile() method. The FileAcquisition event must be handled to provide the filename for each page.

C#

```
If (device.Open())
{
    device.PixelFormat = IsisPixelFormat.Binary;
    device.AcquireToFile(IsisFileType.Tiff, IsisCompression.Group4);
    device.Close();
}
…
private void _acquisition_FileAcquisition(object sender,
 IsisFileAcquisitionEventArgs e)
{
                // This event is raised for each page during a file acquisition.
 if (this.chkSaveMultipage.Checked && (e.FileType == IsisFileType.Tiff ||
 e.FileType == IsisFileType.Pdf || e.FileType == IsisFileType.Dcx))
e.Append = true; // This can be true for the first page as well.

                e.FileName = myFileName;
}
```

Use the IsisDevice.Settings.GetSupportedFileTypes to determine which file types are supported by the scanner. If this method does not return the file type you want, take a look at the IsisCodecManager object, which can be retrieved from the IsisAcqusition and IsisController classes.

You also can perform an in-memory acquire and use the DotImage codecs for saving the image.

Image processing and cleanup

Atalasoft DotImage Advanced Document Cleanup (ADC) is an add-on module to Atalasoft DotImage Document Imaging providing document cleanup algorithms that can be applied to scanned documents to clean them up for better compression and archival, increased readability, and for improved OCR accuracy.

Atalasoft DotImage ADC uses proprietary algorithms developed by the Atalasoft research and development team. These are designed to automatically select the best parameters for fast and accurate processing.

The commands included in ADC extend the command interfaces already in Atalasoft DotImage.They can easily be applied to an existing Atalasoft DotImage application. See the online Advanced Document Cleanup Demo that demonstrates ADC with our AJAX-enabled Web Image Viewer.

For more information, see Advanced Document Cleanup and ADC features.

Manipulate colors with Lookup Tables

Using Lookup Tables (LUTs) is a convenient way to replace a specific color value in each pixel of an image with a different color value. A LUT consists of a 256 element array that defines a new value for each possible value of an 8-bit quantity.

A color image contains pixels consisting of three 8-bit samples (or channels), so three LUTs are needed, one for each color channel. A grayscale image requires one LUT.

You can apply a LUT to any continuous image with the ApplyLutCommand

Invert the Alpha in an RGBA Image

Suppose you have an image with some solid text on a transparent background, and you want to make the text transparent and the background solid. You can achieve this goal by inverting the alpha channel of the image.

You could split the image channels, apply the InverseCommand to the alpha channel, and then combine the channels again. However, this approach is slow. A LUT provides a simple and efficient method for invert the alpha in an RGBA image.

As was stated above, an Atalasoft DotImage LUT consists of a 256 element byte array. A LUT that does nothing to an image contains values from 0 to 255: LUT[0]=0, LUT[1]=1, and so on. A LUT that inverts values is just reversed: LUT[0]=255, LUT[1]=254, ... LUT[254]=1, LUT[255]=0. The following code shows how to reverse a channel. It assumes that you begin with an RGBA image.

Example

This example shows how to reverse an image using a look up table.

C#

```
byte[] lut = new Byte[256];
for (int i = 0; i < 256; i++)
```

```
    lut[i] = (byte)(255 - i);
myWorkspace.ApplyCommand(new ApplyLutCommand(null, null, null, lut));
```

ℹ You can apply this same method to all channels to invert (negate) the colors of the entire image.

Creating a psychedelic effect

The example in the previous section only changed one channel in the image. With the ApplyLutCommand, you can manipulate each channel in an image to create some interesting effects.

The following example creates a strange effect by twisting the colors around. It inverts the dark half of the color range, but leaves the bright colors alone.

Example

The following code uses a LUT to manipulate an image's colors.

Twisting colors

C#

```
byte[] lut = new Byte[256];
for (int i = 0; i < 127; i++)
lut[i] = (byte)(255 - i);
for (int i = 127; i < 256; i++)
lut[i] = (byte)i;
myWorkspace.ApplyCommand(new ApplyLutCommand(lut, lut, lut));
```

Resize images

Atalasoft DotImage has the ability to resize (or resample) an image using a number of different algorithms. These resampling algorithms can result in a high quality resized image at the cost of speed, or a high performance algorithm at the expense of quality. Some algorithms do a good job at both speed and quality depending on the image type.

Atalasoft DotImage includes a Thumbnail class which can be used to generate fast high quality thumbnail images from files. The sections below provide more flexibility than the simple Thumbnail class.

Simple Resizing and Thumbnails

Most imaging applications resize an image in some way. For display purposes, Atalasoft DotImage takes care of this with the WinForm and WebForm controls where the AutoZoom property is set to BestFit, and the image is quickly resized to fit the control. What if you wanted to work with this resized image? Using the ResampleCommand , an image can be resampled to any resulting size. The following code snippet shows how to create a thumbnail in the Workspace with a maximum width or height of 100 pixels while maintaining the aspect ratio.

Example

This example shows how to create a thumbnail with maximum height while maintaining aspect ratio.

C#

```
myWorkspace.ApplyCommand(new ResampleCommand(100);
```

The default resampling method for reductions is AreaAverage and for enlargements is Bi-Linear. This results in fast, high quality resampling. For very high quality thumbnails, we recommend the LanczosFilter() method, however this algorithm is quite a bit slower.

Resizing Continuous Tone Images

For resizing continuous tone images the ResampleCommand provides 18 different resampling algorithms, details of which can be found in the enumeration reference for ResampleMethod. As shown above, specifying one Integer in the constructor will resize the image to that maximum width or height while keeping the aspect ratio. A Size structure can also be passed which allows the width or height be explicitly defined. The following example demonstrates doubling the image size using the LanczosFilter ResampleMethod.

Example

This example shows how to double image size.

C#

```
myWorkspace.ApplyCommand(new ResampleCommand(new Size(myWorkspace.Image.Width * 2,
myWorkspace.Image.Height * 2), ResampleMethod.LanczosFilter);
```

The default resampling method for reductions is AreaAverage and for enlargements is Bi-Linear. This results in fast, high quality resampling. For very high quality thumbnails, we recommend the LanczosFilter() method, however this algorithm is quite a bit slower. The resample algorithms available include those listed in the table below.

NearestNeighbor	TriangleFilter	Cubic1Filter	HermiteFilter
BiLinear	HammingFilter	Cubic2Filter	HanningFilter
BiCubic	GaussianFilter	LanczosFilter	CatromFilter
AreaAverage	BellFilter	MitchellFilter	
BoxFilter	BsplineFilter	SincFilter	

Resizing Palette Images

The ResampleCommand can also be used to resample colormapped images. If the image pixel format is Pixel8bppIndexed then the Default and Simple ResampleMethod() resizes the image while maintaining the integrity of the palette and colormapped status. Choosing any of the other resampling methods will raise the ChangePixelFormat static event in the AtalaImage object and cause the image to be resized to a continuous format before resampling.

For the most efficient resampling of colormapped images to continuous tone images, use the ResampleColormappedToRgbCommand. This is what is used when the WorkspaceViewer control contains a colormapped images and is scaled with Antialiasing on. It performs faster than first converting the image to continuous tone, then resampling with ResampleCommand.

Resizing Binary Images

Atalasoft DotImage Document Imaging contains the ResampleDocumentCommand that is designed for document imaging. This command has three methods: Nearest Neighbor, Scale to Gray, and Area Average. This can be used for simple and fast 1-bit resizing, an efficient Scale to Gray resizing, or Favor Black / Favor White resizing. When Antialiasing is turned on in the WorkspaceViewer, this command is used for scaling binary images to gray and yields high quality, fast viewing of scaled down binary images.

The three basic document resampling methods are listed in the table below.

Resampling Method	Description
NearestNeighbor	Simple and fast 1-bit resizing.
ScaleToGray	High quality scaling of 1-bit images that automatically converts the images to grayscale.
AreaAverage	Fast 1-bit reductions that output black or white pixels depending on the source image data and weight factor. Used for Favor Black / Favor White.

ℹ The ScaleToGray method can be applied to all ResampleMethods specified with the ScaleToGrayMethod property with the exception of NearestNeighbor and Bi-Cubic.

Process an image using a Workspace object

To process an image when using the Workspace object:

1. Create a new instance of a command.
2. Pass the command into the ApplyCommand() method of the Workspace object.

Example

This example demonstrates how to apply a Gaussian blur filter to an image with a sigma value of 2.0.

C#

```
myWorkspace.ApplyCommand( new BlurGaussianCommand(2.0));
```

Example

This example demonstrates how to apply a Gaussian blur to multiple images. The same command can be applied to multiple images as shown below.

C#

```
BlurGaussianCommand blur = new BlurGaussianCommand(2.0);
foreach (AtalaImage image in myWorkspace.Images)
{
  myWorkspace.Image = image;
  myWorkspace.ApplyCommand(blur);
}
```

Process an image using the Apply method

If it is not desirable to use the Workspace, the Apply() method of the ImageCommand offers an alternative method for processing an AtalaImage object.

Example

The following example demonstrates how to apply a sharpen filter directly to an AtalaImage object.

C#

```
SharpenCommand sharpen = new SharpenCommand(0.5, 3);
AtalaImage newImage = sharpen.Apply(newImage).Image;
```

The Apply() method returns an object of type ImageResults . Unlike the ApplyToImage() method, Apply() never returns null. TheImageResultsobjects contains a property calledImagewhich always is set to a non-null value. In some cases, theImageproperty returned in the results is the same as that in the original image. You can verify this by looking at theImageResultspropertyIsImageSourceImage.

ℹ For backward compatibility, ImageCommand still provides the older ApplyToImage() method, but it is now deprecated.

Process a Bitmap image

To process a Bitmap image directly, convert the Bitmap to a temporary AtalaImage object and then process the effect. For more information, see Working with GDI+ Images.

Extend ImageCommands

The ImageCommand is an abstract base class that must be inherited by all image processing functions that modify the current image. The child class must override the several methods, allowing the Workspace object to manipulate the current image. By inheriting the ImageCommand base class, it is easy to create new effects and image processing functions.

If you plan to create your own custom commands, we recommend that you use C# rather than Visual Basic to code your application. C# is the recommended language because of its support for unsafe programming. Unsafe programming allows you manipulate an image directly. This is much faster than the GetPixel() and SetPixel() methods when changing image data.

You can access image data with the PixelMemory property in the AtalaImage object. For more information about accessing pixel data directly, see Access pixel data.

The naming convention in Atalasoft DotImage is such that all objects of type ImageCommand end with Command. The exception to this convention is a special type of ImageCommand object called a Transform that ends with Transform.

Atalasoft DotImage has a formal process for creating ImageCommands which reduces the amount of work needed by the command writer. The process has the following steps:

1. Allocate an object for holding image results.
2. Verify the integrity of the source image.
3. Verify command properties (if possible).
4. Change the pixel format of the source image, if necessary.
5. If the command supports region processing, crop the image to the region of interest.
6. If the command needs a destination image, construct it.
7. Perform the command.
8. Perform any recomposition of the processed image, if necessary.
9. Dispose any intermediate images.
10. Copy over the final image results.

In most cases, it is not necessary to worry about this process. The overall design handles nearly all the special cases that arise in processing images.

In the nearly all cases, you will only need to implement three abstract members. Each will be discussed in detail:

Member	Description
SupportedPixelFormats	Returns an array of all pixel formats that are natively supported by your command
VerifyProperties	Checks that all input properties are correct for this command. If not, this method should throw an exception
PerformActualCommand	Does the actual work needed to implement your new command

As much as possible, the underlying implementation of Apply adheres strongly to the contract that it should never pass protected members invalid arguments. In other words, all parameters passed into one of your methods will be valid for the context of the command.

SupportedPixelFormats

This property returns an array of pixel formats that your command operates on in an optimal fashion. Avoid returning all pixel formats and doing conversion within your command. This is managed for you by the CanApplyToAnyPixelFormat and ApplyToAnyPixelFormat properties.

A typical implementation of this property might look like the example shown below.

Example

The following example demonstrates how to apply a sharpen filter directly to an AtalaImage object.

C#

```
static PixelFormat[] _supportedPixelFormats = newPixelFormat[] {
    PixelFormat.Pixel1bppIndexed
};
public override PixelFormat[] SupportedPixelFormats { get
{ return _supportedPixelFormats; } }
```

VerifyProperties

VerifyProperties(AtalaImage sourceImage) is used to ensure that everything is suitable in the command to apply to the given image. If something is incorrect (properties or parameters that clash), this method will throw a suitable exception. The supplied sourceImage will always be a valid image and will always be the image provided to Apply. There is no need to check the PixelFormat of sourceImage as it will be verified later for you.

PerformActualCommand

PerformActualCommand(AtalaImage source, AtalaImage dest, Rectangle imageArea, ref ImageResults results) is the method that implements the actual command.

Source is the source image. It may not be the same image as supplied by the command, but it will always be non-null and will always be in a pixel format in the array returned by SupportedPixelFormats.

Dest is an AtalaImage that represents the destination image for this command. If the command returns true in the property InPlaceProcessing, then dest will always be null. If the command indicates that it allocates the destination image itself, dest is always null (see below).

If the image command is a subclass of ImageRegionCommand, imageArea will be set to the rectangular bounds of the region of interest. If there is no region of interest or the command is not an ImageRegionCommand, imageArea will be set to {0, 0, source.Width, source.Height}.

Results is a reference to the ImageResults object allocated for this command. You can set properties of the results or replace it with a completely new object if you choose. Most commands ignore the results.

PerformActualCommand will return null in most cases. If, for some reason, the command needs to allocate a destination image itself at the very last moment, it should return this image.

Atalasoft DotImage has the ability to manipulate image data without using unsafe code. Using an image's ImageData pointer directly is supported, but is considered obsolete. Instead, use the PixelMemory and PixelAccessor classes.

SimpleInversionCommand

The following is a complete example of a new ImageCommands that inverts 1 bit per pixel images.

C#

```
public class SimpleInversionCommand : ImageCommand
{
    public SimpleInversionCommand() { }
    // operate on the source image
 public override bool InPlaceProcessing { get { return true; } }
    // only natively support 1 bit images
 private static PixelFormat[] _supportedFormats = newPixelFormat[1]
{ PixelFormat.Pixel1bppIndexed };
    public override PixelFormat[] SupportedPixelFormats { get
{ return_supportedFormats; } }
    protected override void VerifyProperties(AtalaImage image)
    {
        // nothing needed
    }

    protected override AtalaImage PerformActualCommand(AtalaImage source,
AtalaImage dest, System.Drawing.Rectangle imageArea, refImageResults results)
    {
        // get the PixelMemory object
 int height = source.Height;
        PixelMemory pm = source.PixelMemory;
        using (PixelAccessor pa = pm.AcquirePixelAccessor())
        {
            byte[] row;
            while ((row = pa.AcquireNextScanline()) != null)
            {
                for (int i=0; i < row.Length; i++)
                {
                    row[i] = (byte)~row[i];
                }
            }
            pa.Release();
        }
        return null;
    }
}
```

Writing complex commands

To prepare yourself for writing a new ImageCommand, ask the following questions:

1. Will the command only operate on the source image or does it need a destination image as well?
2. Will the command operate on a region of the image or operate only on the entire image?
3. If the command requires a destination image, will it be the same PixelFormat and size as the source image?
4. Do I need to return more information about what happened in the command?
5. If my command supports multiple pixel formats and ApplyToAnyPixelFormat is set to true, which PixelFormat is the best to use?

If your command will only operate on the source image, you must override the property InPlaceProcessing (see above example) so that it returns true. In this case, your implementation of PerformActualCommand will only be passed a valid source image and should return null.

If your command will operate on a region of the image, your class should descend from ImageRegionCommand. When PerformActualCommand is called, imageArea will be set to the smallest rectangle that contains the region of interest. In addition, the dest image, if you have not indicated that you will allocate it yourself, will be set to a size that matches that rectangle.

If your command needs the destination image in a different pixel format or size than the source image, you should override the protected method ConstructFinalImage(AtalaImage image). In this method you should allocate a new image that matches the size of the provided image in a PixelFormat that you would prefer. This image is passed back to you in PerformActualCommand. If you have no way of knowing how big or what pixel format the final image should be, this method should return null.

This is the only circumstance in which this method returns null. If you attempt to construct an AtalaImage and fail, throw an appropriate exception.

If you need or want to provide more information about a command than the standard ImageResults object provides, you should create a subclass of ImageResults to implement the functionality that you need. You can either then override the protected factory method ConstructImageResults() or you can set the reference parameter results in your implementation of PerformActualCommand() to your custom type.

Atalasoft DotImage takes several steps in choosing the pixel format that your command will use. By default, if your command supports the source image's PixelFormat, then your command will be handed the source image, unchanged. If your command is set to operate on any pixel format (via the ApplyToAnyPixelFormat property) and the source image's pixel format is not natively supported, then Atalasoft DotImage will construct a new image using the pixel format of the 0^{th} entry in the array returned by SupportedPixelFormats.

If neither of these approaches are sufficient, you can override this behavior.

The protected method SelectPreferredPixelFormat(AtalaImage image, PixelFormat sourceFormat, PixelFormat[] supportedFormats) is called to choose a pixel format for the source image passed to PerformActualCommand. By default, this method returns sourceFormat. You can make this method return any PixelFormat at all. For example, some commands can operate on many pixel formats, but certain ones may perform optimally. You could conditionally return the optimal PixelFormat - or maybe the optimal format is different based on your command's properties. If you return a PixelFormat that is not in your array of supported PixelFormats, Atalasoft DotImage throws an IncompatiblePixelException if ApplyToAnyPixelFormat is false.

If you wish to change the way that the best PixelFormat is selected by Atalasoft DotImage when it is necessary to change the image's PixelFormat, override the protected method SelectBestAlternatePixelFormat(AtalaImage image, PixelFormat sourceFormat, PixelFormat[] supportedFormats). For example, you might want to return 8-bit gray if the source image is 1 bit, but 24 bit color when the source image is 4 or 8 bit.

If these rules do not work in your command, you can override the protected method ConstructChangedSourceImage(AtalaImage image). In this method, if you do not need to change the source image's PixelFormat, you should return null. Otherwise you should construct a new AtalaImage the same size as the supplied image and return it. Like ConstructFinalImage, this method should never return null as an error. It should instead throw an exception.

Finally, you can override the public method Apply() to implement your command, but you will lose all current and future benefits of the Atalasoft DotImage implementation. Therefore, it is only recommended that you override this method when there is no other way to do this.

ColorizeBlackCommand

This example takes a 1 bit per pixel image and turns every black pixel into a specified color. It overrides ConstructFinalImage to perform the work.

C#

```
public class ColorizeBlackCommand : ImageCommand
{
    private Color _blackColor = Color.Black;
    public ColorizeBlackCommand()
    {
    }
    public ColorizeBlackCommand(Color replacementColor)
    {
        _blackColor = replacementColor;
    }
    // only natively support 1 bit images
 private static PixelFormat[] _supportedFormats = new PixelFormat[1]
{ PixelFormat.Pixel1bppIndexed };
    public override PixelFormat[] SupportedPixelFormats { get
{ return _supportedFormats; } }
    // return a final image in 24 bit color
 protected override AtalaImage ConstructFinalImage(AtalaImage image)
    {
        AtalaImage finalImage = new AtalaImage(image.Width, image.Height,
PixelFormat.Pixel24bppBgr);
        if (finalImage == null) // always throw on a null image
 throw new OutOfMemoryException("Out of memory in ColorizeBlackCommand");
        return finalImage;
    }
    protected override void VerifyProperties(AtalaImage image)
    {
        // nothing needed
    }
    protected override AtalaImage PerformActualCommand(AtalaImage source,
AtalaImage dest, Rectangle imageArea, ref ImageResults results)
    {
        // get the source width and height
 int height = source.Height;
        int width = source.Width;
        // find out if a 1 bit is black or white
 bool oneIsBlack = IsColorBlack(source.Palette.GetEntry(1));
        // get the color that should be used to change to black
 byte red = (byte)_blackColor.R;
        byte green = (byte)_blackColor.G;
        byte blue = (byte)_blackColor.B;
        PixelMemory sourcePM = source.PixelMemory;
        PixelMemory destPM = dest.PixelMemory;
        using (PixelAccessor sourcePA = sourcePM.AcquirePixelAccessor(),
destPA = destPM.AcquirePixelAccessor)
        {
            for (int y=0; y < height; y++)
            {
                    byte[] sourceRow = sourcePA.AcquireScanline(y);
                    byte[] destRow = destPA.AcquireScanline(y);
                    for (int x=0; x < width; x++)
```

```
                    {
                        if (IsBitSet(sourceRow, x, oneIsBlack))
                        {
                            destRow[x * 3] = blue;
                            destRow[x * 3 + 1] = green;
                            destRow[x * 3 + 2] = red;
                         }
                         else
                         {
                            destRow[x * 3] = 255;
                            destRow[x * 3 + 1] = 255;
                            destRow[x * 3 + 2] = 255;
                         }
                    }
                }
                sourcePA.ReleaseScanline();
                destPA.ReleaseScanline();
            }
            return null;
        }
        // is the supplied color black
     private bool IsColorBlack(Color c)
        {
            return c.R == 255 && c.G == 255 && c.B == 255;
        }
        // is a bit set at the give x coordinate in the row
     private unsafe bool IsBitSet(byte[] row, int x, bool oneIsBlack)
        {
            byte theByte = row[x >> 3];
            int theBitIndex = 7 - (x & 0x7);
            bool bitSet = (theByte & (1 << theBitIndex)) != 0;
            return oneIsBlack ? !bitSet : bitSet;
        }
        public Color ReplacementColor { get { return _blackColor; } set
{ _blackColor = value; } }
}
```

Upload an image to a server

There may be times when you want to send an image to a server. The HttpPost class was written for this specific purpose.

Upload the image

The following example sends a single image, along with a user name and password to identify the sender, to a server.

Example

C#

```
private void UploadImage(Atalasoft.Imaging.AtalaImage image)
{
// Create an instance of HttpPost and use the default image encoder.
Atalasoft.Imaging.HttpPost post = new Atalasoft.Imaging.HttpPost();

// Add the image to the form data collection.
post.FormData.Add("image1", image, "filename.png");
```

```
// Add data to identify the user posting the image.
post.FormData.Add("username", user);
post.FormData.Add("password", password);

// Post the form data to the server and retrieve a return value.
string ret = post.PostData("http://www.website.com/postImage.aspx");
}
```

The first part of the code creates an instance of HttpPost and uses the default image format, which is PNG. The image format determines the file type of the image when saved on the server.

Then the image is added to FormDataCollection, which includes the field name, image and filename for the image. It is best to think of HttpPost as an HTML FORM object. The FormData represents the FORM elements; in this case a FILE input box. You can update multiple images by adding them to the collection; just be sure each has a unique field name.

Next, add the username and password of the person sending this image. In many cases, additional information needs to be sent with the image in order to perform a specific action on the server. In terms of an HTML FORM, this would be a TEXTBOX field.

Finally, the FormData is sent to the server using the PostData() method. The return value of PostData is a string that is sent back from the server. Normally this is used to confirm the success of the call.

Saving the image

Once the image has been sent, save it to the server using ASP.NET. If necessary, you can modify this code to store the image in a database instead.

Example

C#

```
private void Page_Load(object sender, System.EventArgs e)
{
if (Request.Files.Count == 0) return;

// Save the file to the server.
string fileName = System.IO.Path.GetFileName(Request.Files[0].FileName);
Request.Files[0].SaveAs(GetNewFileName(fileName));

// Return the path to this file.
Response.Clear();
Response.Write("success");
Response.End();
}

private string GetNewFileName(string fileName)
{
// Create a unique filename.
string path = Server.MapPath("./images") + "\\" + Session.SessionID + fileName;
return path;
}
```

In the code above, the SaveAs() method saves the image to the server. Notice that the FileName property gets the name of the posted file. This is the same value that was passed to the FormData.Add() method when you added the image.

The data written to the Response object is returned by the PostData() method. It is a good idea to use the Response.Clear() method before adding your return value. This example simply returns success

Annotations

The following sections cover various aspects of annotations.

Annotations

Atalasoft DotImage contains powerful annotation capabilities that can be used to markup, draw, and visualize objects on an image or document. These objects include primitive shapes, text, freehand, sticky notes, images and hot spots. Atalasoft DotImage includes an extensible interface that allows you to add your own custom annotation objects.

Annotations can be independently resized, moved, rotated, placed on different layers or groups, or saved to a separate file. The annotations can be embedded within an image* using the standard WANG format or our published format based on Adobe's XMP standard.

Features

Atalasoft DotImage annotations have the following features:

- Add an arbitrary number of annotation objects to an image.
- Annotation objects can be moved, resized, and rotated independently of the image.
- Annotations can be placed on layers.
- Annotations can be grouped. Once grouped they are resized and moved as one object.
- Annotations or layers can be locked/unlocked or visible/invisible.
- A fast and smooth display. Objects can be repainted while being resized.
- Uses GDI+ graphics allowing any object to be rendered at variable transparency.
- Extensible object model allows you easily to create your own custom drawn annotation objects with very little code.
- Save or load annotations as a separate XML file using Adobe's standard XMP format.
- Save or load WANG compatible annotations to or from a separate file.
- A custom cursor can be display the cursor hovers over an annotation.
- Annotations can be rotated along with the image in 90 degree increments.
- Respond to varied mouse clicks and events with flexible hyperlinking or other custom actions.
- Annotations can be defined in any unit system, including your own unit system.
- Annotations can be placed on any control using the IAnnotate interface.
- Individual points from annotations supporting multiple points (Freehand, Polygon, etc.) can be repositioned to change the shape of the object.

- Grips can be customized to display different shapes, custom mouse-over effects and more.
- Built-in AnnotateViewer (Atalasoft.dotImage.WinControls.dll) supports the IAnnotate interface and all viewing features of Atalasoft DotImage WinForms including scale to gray display, mouse tools, magnifier, etc.
- Save or load WANG compatible annotations to/from a TIFF image*.
- Save or load annotations embedded in a TIFF, JPEG, or PDF image in XML using Adobe's standard XMP format.
- Annotations can be burned onto the image with a single method.
- Annotations can be printed with the base image using the AnnotatePrintDocument class.
- Supports proper viewing of FAX images with differing X and Y resolutions.

Supported Annotations

The following is a list of supported Atalasoft DotImage annotation types.

ℹ Some of the annotations below use the same annotation object with different property settings. See the object reference for detailed information and examples.

- Rectangle
- Highlighter
- Ellipse
- Line
- Freehand
- Text
- Note
- Rectangular HotSpot
- Freehand HotSpot
- Freehand Highlighter
- Embedded Image / Stamps
- Referenced Image
- Polygon
- Lines
- RubberStamp
- CalloutAnnotation

Import and export annotations

Starting with Atalasoft DotImage 7.0, Atalasoft DotImage Annotations contain a more formalized way for importing and exporting annotations. Both importing and exporting are represented by abstract objects that contain partial implementations that do most of the work for you.

Importing means to read the contents of a stream that contains an external representation of annotations and translates them into equivalent Atalasoft DotImage Annotation objects.

Exporting means to translate Atalasoft DotImage Annotation objects into some other equivalent data format and write to another location. In addition, exporting includes the notion of exporting

annotations *over* an existing set within a file without disturbing the rest of the file. In most cases, this means simply writing annotations to a stream. In other cases it means being able to understand the format of an existing file to adjust it to compensate for new annotation data.

Both AnnotationDataImporter and AnnotationDataExporter may need to understand different annotation coordinate systems and scaling.

Unlike ImageDecoder and ImageEncoder objects which are stateless and can be used for many different streams, a unique AnnotationDataImporter or AnnotationDataExporter object is used per stream. This is done to allow caching and other performance optimizations.

Dual use objects

AnnotationDataImporter and AnnotationDataExporter objects are considered dual-use objects. They can serve as either a factory or as an importer/exporter of annotations.

As a factory, an AnnotationDataImporter can identify if a stream contains correct data, and can construct a new AnnotationDataImporter for a given stream. As an importer, an AnnotationDataImporter can provide information about the number of pages of annotations in a stream, the number of annotations per page, and it can translate the data into Atalasoft DotImage Annotation objects.

As a factory, an AnnotationDataExporter can identify if a stream contains correct data, and can construct new AnnotationDataExporter objects. As an exporter, an AnnotationDataExporter can indicate whether or not it can export annotations directly, whether or not it can export annotations over an existing document, and it can translate Atalasoft DotImage Annotation objects into its own format.

Import annotations

To import annotations, client code needs to be able to identify the correct AnnotationDataImporter to use for a given stream and then to construct a new AnnotationDataImporter object to use for the actual importing.

Example

C#

```
List<AnnotationDataImporter> _myImporters;

protected virtual void GetImporter(Stream stream)
{
      foreach (AnnotationDataImporter importer in _myImporters) {
            if (importer.IsValidFormat(stream))
                  return importer.FromStream(stream);
      }
      return null;
}

public AnnotationDataCollection ImportAllAnnotations(Stream stream)
{
      AnnotationDataImporter import = GetImporter(stream);
      try {
            if (importer != null) {
                  return importer.Import();
            }
```

```
            return new AnnotationDataCollection();
      }
      catch (Exception err) {
            MessageBox.Show("Unable to load annotations: " + err.Message);
      }
}
```

In addition, it is possible to suppress errors encountered during importing by setting the ThrowOnError property of the AnnotationDataImporter to false. After calling import, all the Message properties from every exception thrown will be appended to the LastErrors property of the AnnotationDataImporter. LastErrors is automatically cleared by the AnnotationDataImporter before importing or loading annotations.

Even if ThrowOnError is set to false, there is no guarantee that any particular AnnotationDataImporter will recover from an error. Most AnnotationDataImporters will halt at the first error on a page. For importing multiple pages, this means that one or more pages may have incomplete annotations.

Write a custom AnnotationDataImporter

In order to create a custom AnnotationDataImporter, a client object must inherit from AnnotationDataImporter and at a minimum, implement the following abstract methods:

```
public abstract bool IsValidFormat(Stream stm);
```

`IsValidFormat` returns true if the stream is in the correct format for this AnnotationDataImporter. It is vital that the stream is left in the same state as when `IsValidFormat` was entered.

```
public abstract AnnotationDataImporter FromStream(Stream stm);
```

`FromStream` constructs a new AnnotationDataImporter attached to the given stream object.

```
protected abstract void LLLoad();
```

`LLLoad` is the low-level implementation of Load. `LLLoad` should do any pre-flight or caching of annotation data. For example, some AnnotationDataImporters do all their work in `LLLoad` and simply access constructed data in other methods. `LLLoad` will only be called once per AnnotationDataImporter object.

```
protected abstract int LLGetPageCount();
```

Returns the total number of pages in the given stream. `LLGetPageCount` is only called after a call to Load().

```
protected abstract int LLGetAnnotationCount(int page);
```

Returns the total number of annotations on a given page. `LLGetAnnotationCount` is only called after a call to `Load()`. Page is always greater than or equal to zero and less than `LLGetPageCount`.

```
protected abstract AnnotationData LLImport(int page, int annotIndex);
```

Returns an imported annotation by index from a given page. Page is always greater than or equal to zero and less than `LLGetPageCount(). annotIndex` is always greater than or equal to zero and less than to `LLGetAnnotationCount(page)`.

In addition, there is an implementation of `LLImport` which is not abstract:

```
protectedvirtual LayerData LLImport(int page) { /* … */ }
```

Some client code will be much more efficient if they can import all the annotations on a given page rather than doing it piecemeal. The default implementation gets the number of annotations on the given page, makes a new LayerData, fires the OnAnnotationPageImporting event, then called `LLImport(page, index)` for every annotation on the page, firing the OnAnnotationImported event, then adding the item to the layer.

`LLImport(page)` does NOT fire the OnAnnotationPageImported event. That event is fired by the high `level Import()` method.

Handle unknown annotation types

Some data formats for annotations are living standards that are expected to grow in the future. As such, there needs to be the ability to handle unknown annotation types during importation. In the AnnotationDataImporter, there is a property called FailsafeAnnotationFactory. This property, if non-null, is a delegate to a method that can be used to construct an Atalasoft DotImage Annotation if the AnnotationDataImporter can't. This delegate gets passed the AnnotationDataImporter that attempted to import the annotation, a RectangleF representing the bounds of the annotation in the coordinate system of the destination, and an object representing the data for the annotation. This object is constructed by the specific AnnotationDataImporter and its class and contents are up to the author of the AnnotationDataImporter.

Coordinate systems

Annotation data formats may or may not have their own coordinate system. In general, an AnnotationDataImporter is expected to understand the coordinate system of the source annotations and be able to convert it to Atalasoft DotImage coordinates. In Atalasoft DotImage, the coordinate system is similar to most imaging. It starts with (0, 0) in the upper left corner with X increasing to the right and Y increasing downward. The units of the coordinate system may depend image on top of which the Atalasoft DotImage annotation will be displayed. For example, the XmpAnnotationDataImporter requires an AnnotationController in order to determine units and size. The PdfAnnotationDataImporter uses PDF coordinates for everything.

If client code implements this delegate, it is expected to return either a new object of type AnnotationData or null. Returning null means that the annotation will be ignored.

The PdfAnnotationDataImporter

New to Atalasoft DotImage 7.0 was a special AnnotationDataImporter for PDF documents. At present, it reads all flavors of PDF except for encrypted PDF and PDF with binary cross-reference tables (an option in PDF 1.5 and above). It will import the following PDF annotation types: Caret, Circle, Highlight, Line, Polygon, Polyline, Square, Squiggly line, Strikeout, and Underline. Other standard PDF Annotation types will be represented by a TextAnnotation object with text that starts "Unknown annotation of type ..."

When the PdfAnnotationDataImporter encounters a PDF Annotation that is outside of the PDF spec, it will call the FailsafeAnnotationFactory. The object passed in to the factory will be a .NET Hashtable object. It will contain a set of key/value pairs that represent data inside the PDF annotation. The

keys are all strings and correspond to the names of dictionary entries within the annotation according to the Adobe PDF Specification. We do not yet specify what the values will be under all circumstances and their data type and contents are subject to change. In general, however, the values will be the closest corresponding .NET data type (ie, PDF strings will be .NET strings, PDF numbers will be doubles or integers, PDF arrays will be ArrayList, etc.).

New to Atalasoft DotImage 8.0 was the ability for PDF annotations to contain embedded data. Atalasoft DotImage can interpret this data as a serialized AnnotationData object that can be deserialized into the original object. A PDF annotation that contains embedded data is acting as a proxy. If that proxy has been moved/sized/edited by a user in the PDF, DotImage Annotations have a mechanism for merging changes in the proxy annotation into the embedded annotation. Most of this is handled automatically, by copying over the most commonly used editable properties, but through the use of collection of "mergers" it is possible to customize how proxy annotations are merged into the embedded annotations.

The XmpAnnotationDataImporter

Atalasoft DotImage Annotations also include the XmpAnnotationDataImporter. Given an annotation controller, the XmpAnnotationDataImporter can read XMP data from either TIFF or JPEG streams and convert them into Atalasoft DotImage Annotations.

Export annotations

In addition to being able to serialize Atalasoft DotImage Annotations, it is possible to export them to other formats. Exporting is different from serializing. In many cases, simple serialization is exactly what is needed for a task. Serialization produces one chunk of data that fully encompasses all elements of a set of Atalasoft DotImage Annotations. Exporting involves taking the annotation data and possible changing it from Atalasoft DotImage Annotations into some other data type and inserting it into or appending it onto some other file. The final file may not have an exact representation of the original annotations. Exporting is done through an AnnotationDataExporter object. Like AnnotationDataImporter, it serves two purposes. It acts as a factory that can make new AnnotationDataExporter objects and it can act as a writer.

Exporting annotations can happen in two ways. The first is simple exporting. The annotation is given a source stream from which represents an existing file and a destination stream for writing the annotations. In general, the source stream will be copied to the destination stream then the annotations will be added on. Some exporters will read sections of the source stream and intersperse annotations into the output stream. Some exporters will ignore the source stream entirely.

The second way that annotations can be exported is by exporting *over* an existing file. This means writing annotations into an existing file in such a way that it supersedes existing content. For example, PDF is defined in such a way that exporting over is an existing file is a natural operations. It is straightforward to modify a document to include new annotations without affecting the existing document otherwise.

AnnotationDataExporter reports which methods are best handled via the properties CanExport and CanExportOver. An AnnotationDataExporter needs to return true for at least one of these.

In spite of the restrictive appearance of Export and ExportOver, these methods are actually implemented in terms of each other. If an AnnotationDataExporter reports that it is unable

to do ExportOver, the source stream will be copied to a temporary file and then Export will be called with the temporary file as the source and the original file as the destination. If an AnnotationDataExporter reports that it is unable to do Export, Export will copy the source stream to the destination stream and then call ExportOver.

In general, client code should not care whether or not an AnnotationDataExporter supports one or the other means of exporting except in the case of efficiency or resource management. The properties CanExport and CanExportOver are meant to reflect what an AnnotationDataExporter can do *best*, thereby allowing the client to choose the most efficient mode of operation.

Exporting is somewhat more complicated than importing in that it is necessary to know the coordinate system of every page before exporting. To translate coordinate systems correctly, an AnnotationDataExporter needs to know the size of each page in source page units, the resolution of the source page and the units of all pages.

Write a custom AnnotationDataExporter

In order to create a custom AnnotationDataExporter, client code must inherit from AnnotationDataExporter and at a minimum implement the following abstract methods:

```
public abstract bool CanExportOver { get; }
```

Indicates that the AnnotationDataExporter can export over an existing stream.

```
public abstract bool CanExport { get; }
```

Indicates that an AnnotationDataExporter can export from a source stream to a destination stream.

```
public abstract bool IsValidFormat(Stream stm);
```

Returns true if the stream is the correct format for this Exporter.

```
public abstract AnnotationDataExporter Construct();
```

Constructs a new AnnotationDataExporter.

```
protected abstract void LLExport(Stream sourceStream, Stream destStream,
SizeF pageSize, AnnotationUnit units, Dpi resolution, LayerData layer, int
frameIndex);
```

Exports a single page of annotations from a source stream to a destination stream. If `CanExport` returns false, this method is never called.

```
protected abstract void LLExport(Stream sourceStream, Stream
destStream, SizeF[] pageSizes, AnnotationUnit units, Dpi[] resolution,
AnnotationDataCollection layers);
```

Exports an entire set of annotations from a source stream to a destination stream. If `CanExport` returns false, this method is never called.

```
protected abstract void LLExportOver(Stream sourceStream, SizeF pageSize,
AnnotationUnit units, Dpi resolution, LayerData layer, int frameIndex);
```

Exports a page of annotations over an existing stream. If `CanExportOver` returns false, this routine is never called.

```
protected abstract void LLExportOver(Stream sourceStream, SizeF[] pageSizes,
AnnotationUnit units, Dpi[] resolution, AnnotationDataCollection layers);
```

Exports an entire set of annotations from a source stream to a destination stream. If `CanExportOver` returns false, this routine is never called.

The PdfAnnotationDataExporter

The PdfAnnotationDataExporter can export annotations to a PDF document, translating them into annotations. The PdfAnnotationDataExporter translates the following Atalasoft DotImage annotation types into PDF annotations:

- Rectangle
- Ellipse
- Text
- Freehand
- Polygon
- Lines
- Line
- PdfMarkup
- PdfLine

There are properties of AnnotationData that does not have equivalent properties in the PDF annotations. Here is the list of those properties:

- **Rotation** from all AnnotationData objects.
- **Shadow** from RectangleData, EllipseData and TextData.
- **LineType** from FreehandData and PolygonData.
- **Fill** from FreehandData.
- **Alignment** from TextData.
- **FormatFlags** from TextData.

These properties are ignored when converting.

The PdfAnnotationDataExporter can handle exporting annotations that do not necessarily make sense in PDF. For example, if you have created an annotation in Atalasoft DotImage that includes hooks into a database, it does not work in Acrobat or other PDF viewers without a custom plug-in. Atalasoft DotImage annotations are, by default, embed a serialized version of the annotation inside a PDF Rectangle annotation. The Rectangle annotation in turn will be given a custom appearance take from the custom annotation itself. Although the PDF annotation will not have the same behavior as in Atalasoft DotImage annotations, the embedded data can be deserialized into the original object.

In addition, users can customize the annotation appearance in PDF via policies set within the PdfAnnotationDataExporter. PdfAnnotationExportPolicy can be used to instruct the library on how to handle AnnotationData and how it will appear within the PDF.

The PdfAnnotationDataConverter

The PdfAnnotationDataConverter can convert annotations to PDF annotations. The converted annotations can be added to the PdfGeneratedDocument or PdfDocumentSigner. The PdfAnnotationDataConverter has the same capabilities and limitations as the PdfAnnotationDataExporter. (See The PdfAnnotationDataExporter for more information.)

Examples of using the PdfAnnotationDataConverter for converting annotations can be found in the API Reference.

Create an annotation enabled control

The annotation component is designed to be very easy to implement and use. This section describes how to quickly create an annotation enabled control.

Integrate with Atalasoft DotImage

When integrating with Atalasoft DotImage, an AnnotateViewer can be added to a Windows Form or user control. This control is located in the Atalasoft.DotImage.Annotate assembly and references the Atalasoft.DotImage and Atalasoft.DotImage.WinControls assemblies, part of Atalasoft DotImage. The AnnotateViewer control derives from the WorkspaceViewer control with added Annotation functionality. The AnnotationController is accessed through the Annotations property. The AnnotateViewer also has a Burn() method which, as its name suggests, burns the annotation onto the image.

Printing Annotations with the AnnotatePrintDocument makes printing annotations on top of the image very easy. It derives from the ImagePrintDocument available in Atalasoft DotImage WinControls. This control is also available only when integrating with Atalasoft DotImage.

Interactively create an annotation

To interactively create an annotation, an Annotation instance needs to be passed to the CreateAnnotation() method. Once this method is called, a mouse down action adds the annotation to the current layer and it is resized until the mouse is depressed. The resulting annotation can then be edited (resized and repositioned) if the InteractMode in the AnotationController is set to Author. If InteractMode is set to View, then the resulting annotation is locked and cannot be edited. Setting InteractMode to None tells the AnnotationController to ignore mouse messages, allowing the viewer to handle any mouse tools that may be used. All annotation types, and even custom annotation types can be created in this manner.

The following code is shows how to set up code to create a "sticky note" annotation.

Example

C#

Creating a Sticky Note Annotation

```
TextAnnotation myAnnotation = new TextAnnotation();
myAnnotation.Text = "Annotations are cool";
myAnnotation.Fill = new AnnotationBrush(Color.Yellow);
myAnnotation.Shadow = new AnnotationBrush(Color.Gray);
```

```
myAnnotation.ShadowOffset = new PointF(4, 4);
annotationController1.CreateAnnotation(myAnnotation);
```

See Create a Template Annotation to see how to allow the user repeatedly to create the same annotation.

Create an annotation programmatically

To create an annotation programmatically and add it to an image, one needs to simply add the annotation object to the a LayerAnnotation in the Layers collection. The position of the annotation in the collection defines the Z-order or order it's painted. The Add() method in the CurrentLayer adds the annotation to the end of the collection, and is painted on top of all other annotations in that layer.

The following code shows how to add annotations programmatically.

Example

C#

Adding Annotations Programmatically

```

TextAnnotation myAnnotation = new TextAnnotation();
myAnnotation.Text = "Annotations are cool";
myAnnotation.Fill = new AnnotationBrush(Color.Yellow);
myAnnotation.Shadow = new AnnotationBrush(Color.Gray);
myAnnotation.ShadowOffset = new PointF(4, 4);
myAnnotation.Location = new PointF(100, 100);
myAnnotation.Size = new SizeF(100, 200);
annotationController1.CurrentLayer.Items.Add(myAnnotation);
```

Annotation assemblies

The following DotImage assembly is available for work with Annotations.

Assembly	Description
`Atalasoft.DotImage.WinControls.dll`	Contains a control called AnnotateViewer derived from our Atalasoft DotImage toolkit's WorkspaceViewer control and an AnnotatePrintDocument to print an image with annotations

The AnnotationController component contains all the annotations in a viewer whether built in or custom. It has a collection of LayerAnnotations, each with a collection of annotations. The AnnotationController contains properties that affect how the user interacts with the view port. The InteractMode property can be set to View, Author, or None. When in Author mode, individual annotations can be interactively moved, resized and rotated. View mode is for image display, without the ability to interactively edit annotations and make hot spot annotations active.

The AnnotationUI object is an abstract base class that all annotations derive from. New annotations can be added to a document by creating an instance of any one of the annotation objects derived from the AnnotationUI class and passing it into the CreateAnnotation() method in the

AnnotationController. This allows the user to interactively place an annotation onto the IAnnotate view port or AnnotateViewer.

LayerAnnotations can be used to define a set of related objects that have specific permissions or attributes. They can also be used when annotating multipage documents. The Layers property contains a collection of LayerAnnotations loaded in the AnnotationController and each LayerAnnotation can contain additional LayerAnnotation objects.

A group is a set of related annotations that act as one object. It is represented by a LayerAnnotation with its GroupAnnotation property set to true. Groups can be created manually by setting the GroupAnnotation property or by using the `Group()` method of the AnnotationController. Top level LayerAnnotations in the Layers collection cannot be used as groups.

Create a template annotation

To allow a user repeatedly to create an annotation with the same properties, you can setup a template and clone it after the earlier annotation is added to the control.

To do so, handle the AnnotationCreated event in the AnnotationController or AnnotateViewer and clone the template annotation stored as a field. The user is then able to add multiple instances of the same annotation onto a document.

Example

The following code demonstrates how to do this.

C#

```
public MainForm()
{
    InitializeComponent();

    // create template annotation (allow users to modify these properties if required)
    _template = new RectangleAnnotation();
    // start creating annotations interactively
    annotationViewer.Annotations.CreateAnnotation(_template.Clone());
}

public void annotationViewer_AnnotationCreated(object sender, AnnotationEventArgs e)
{
    annotationViewer.Annotations.CreateAnnotation(_template.Clone());
}
```

Print annotations

Atalasoft.DotImage.WinControls.dll includes the AnnotatePrintDocument component in the Atalasoft.DotImage.Annotate assembly. This component derives from ImagePrintDocument which derives from the .NET PrintDocument class. The .NET components that accept PrintDocument also accept AnnotatePrintDocument which gives the application the ability to change printer settings, page settings, and show a print preview.

Example

This example demonstrates how to show a print setup dialog, then print all images with annotations.

C#

```
PrintDialog myPrintDialog = new PrintDialog();
AnnotatePrintDocument myAnnotatePrintDocument = new AnnotatePrintDocument();
myPrintDialog.Document = myAnnotatePrintDocument;
if (myPrintDialog.ShowDialog(this) == DialogResult.OK)
{
    myAnnotatePrintDocument.Image = this.Viewer.Image;
    myAnnotatePrintDocument.Annotations = this.Viewer.Annotations;
    myAnnotatePrintDocument.Print();
}
```

Using IAnnotate

A few approaches are available for printing annotations.

1. Paint the annotations onto the image with the RenderAnnotations() method in AnnotationController, then print the image using the PrintDocument class.
2. Using the PrintDocument class, handle the PrintPage event and draw the document, then draw each annotation using its associated IAnnotationRenderer.
3. Derive from PrintDocument yourself and create your own annotation print document.

In each case, you must determine the scaling and offset required depending on the document size and the page size and depending on how you wish the image to be printed.

Serialize to XMP And WANG data

DotImage Annotations support saving and reading XMP and WANG annotation data. Data can be saved into a separate file or stored in an image usable with Atalasoft DotImage.

WANG annotations

This data format allows basic annotations to be stored into TIFF images. Many image viewers are able to read WANG data, making it a good choice if the images are viewed in other products. However, because only basic shapes and colors can be used, some annotation data will not be reproduced by other products.

When integrating with Atalasoft DotImage, WANG data is saved into TIFF images using TAG ID 32932.

To retrieve WANG data from an image, use the `GetTiffTag()` method of the TiffDecoder.

To save WANG data into an image, create a new TiffTagCollection, adding a TiffTag with the ID of 32932, and set the Data object to the WANG data created with the WangFormatter class.

Saving annotations

This was developed by Adobe so applications can share metadata information using a standard format. The data can be saved as a separate XML file or stored into TIFF, JPEG or PDF files.

In order to include annotation data, an annotation schema was developed to extend XMP. By following this schema, other applications will be able to read and write the annotation data.

When integrating with Atalasoft DotImage, WANG data can be stored into JPEG, TIFF, and PDF images, and retrieved from JPEG and TIFF images. To retrieve XMP data from an image, use the `BytesFromImage()` method of the XmpParser class, and pass the returned byte array into the Load() method of AnnotationController. To save XMP data including DotImage Annotations, use the Save(Formatter) method of the AnnotationController,passing in an XmpFormatter and set the Xmp property of the TiffEncoder, JpegEncoder, or PdfEncoder to the byte array returned from the Save() method.

Custom annotations

If you have developed a custom annotation, the annotation data and UI classes should implement the ISerializable interface. This serialization is used by the WANG and XMP formatters when saving and loading annotations. An example appears below.

Example

C#

```
public TriangleData(SerializationInfo info, StreamingContext context) :
base(info, context)
{
    this._fill = (AnnotationBrush)SerializationInfoHelper.GetValue(info, "Fill",
new AnnotationBrush(Color.Blue));
    base.SetBrushEvents(this._fill);
}
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter=true)]
public override void GetObjectData(SerializationInfo info, StreamingContext context)
{
    base.GetObjectData(info, context);
    info.AddValue("Fill", this._fill);
}
```

Work with unit systems

DotImage Annotations are designed to work in any unit system. You can set the unit system using the Resolution property in IAnnotate . When this value is set to 1.0, the annotation objects are sized to pixel coordinates. Adjusting this value scales the coordinate system relative to pixels.

Example

To position and size objects in inches on a 96 dpi display, set the Resolution value to 96.

Units in the AnnotateViewer control

The AnnotateViewer control has a Units property. This automatically sets the Resolution to the appropriate value based on the resolution of the image in the control. Any value other than Custom overrides the Resolution to a value based on the image's resolution scaled to the appropriate unit.

This is convenient when viewing images such as maps in units such as kilometers. When the X and Y resolutions in the image differ, the control adjusts the aspect ratio and extends the height of the image. Annotation objects can be precisely positioned in familiar coordinates without the need to convert or track units in your own code.

Units in a custom IAnnotate control

Setting the Resolution property to a value (pixels per unit) allows you to specify the location and size of all annotation objects using any unit you wish. The effect is the same as setting the Units property in the AnnotateViewer to Custom and specifying a custom Resolution.

Render custom grips

With the DotImage Annotations polymorphic event-based model, many aspects of the user interface can be customized including the grips.

By default, grips are represented by 10-pixel solid white squares with a black border. They can be customized to any shape or size. You can also code special behaviors depending on the mouse status (hovering, mouse down, and so forth).

There are a two ways to enable custom grip drawing:

- If you have a custom annotation rendering engine that derives from AnnotationRenderingEngine, override the protected `RenderGrips()` method.
- Create a new class that implements the IAnnotationGripRenderer interface.

Overriding the RenderGrips method

Overriding the `RenderGrips()` method of AnnotationRenderingEngine is the easiest way to provide custom rendering because the matrix manipulation is handled, leaving you with the simple task of drawing the grips. To provide a greater understanding of how to deal with matrices when drawing grips, the example that follows takes the challenging route of implementing IAnnotationGripRenderer.

Example

Create a class that implements IAnnotationGripRenderer.

C#

```
public class MyCustomGripRenderer : Atalasoft.Annotate.Renderer.IAnnotationGripRenderer
```

Implement the `RenderGrips()` method by breaking this into section.

C#

```
```

```
public void RenderGrips(IAnnotationGrips grips, AnnotationData annotation,
Atalasoft.Annotate.Renderer.RenderEnvironment e)
{
    if (annotation == null || grips == null || e == null) return;

   // Section: Where will have to get the scaling being applied to the viewer matrix
 (vm)
   //and undo this change because a grips size should not scale.
   // Multiple the annotation transform to the viewer matrix.
   System.Drawing.Drawing2D.Matrix vm = null;
   if (e.Transform != null)
     vm = e.Transform.Clone();
   else if (e.Graphics.Transform != null)
     vm = e.Graphics.Transform.Clone();
   float scaleX = vm.Elements[0];
   float scaleY = vm.Elements[3];
   // The annotation transform.
   System.Drawing.Drawing2D.Matrix m = annotation.GetRenderTransform();
   if (vm != null)
   {
     vm.Multiply(m);
     m.Dispose();
   }
   else
    vm = m;

   // This will undo the scaling the transformation
   // matrix wants to perform, since grips don't scale.
   vm.Scale(1f / scaleX, 1f / scaleY);
   System.Drawing.Drawing2D.GraphicsState state = e.Graphics.Save();
   e.Graphics.Transform = vm;
  // Section: Now we will create our drawing objects
  // and perform the actual drawing.
  // This is the code you would write if overriding
  // the RenderGrips method in a rendering engine.
 // Create the brush and pen objects.
  Brush brush =
 Atalasoft.Annotate.Renderer.AnnotationRenderingEngine.CreateBrush(grips.Fill);
  Pen pen =
 Atalasoft.Annotate.Renderer.AnnotationRenderingEngine.CreatePen(grips.Outline);
  Brush rotateBrush = (grips.RotationFill == null ? brush :
Atalasoft.Annotate.Renderer.AnnotationRenderingEngine.CreateBrush(grips.RotationFill));
  Pen rotatePen = (grips.RotationOutline == null ? pen :
Atalasoft.Annotate.Renderer.AnnotationRenderingEngine.CreatePen(grips.RotationOutline));
  SizeF size = grips.Size;
  float w2 = size.Width / 2f;
  float h2 = size.Height / 2f;
  foreach (AnnotationGrip grip in grips)
  {
    // Ignore grips that should not be drawn.
 if (!grip.Visible) continue;
    if (grip.Action == AnnotationGripAction.Rotating && !annotation.CanRotate)
 continue;
    // While the grip size does not scale, the position of the grip does.
    RectangleF rc = new RectangleF(Convert.ToInt32(grip.Position.X * scaleX - w2),
    Convert.ToInt32(grip.Position.Y * scaleY - h2), size.Width, size.Height);
    if (grip.Action == AnnotationGripAction.Rotating && rotateBrush != null)
       e.Graphics.FillRectangle(rotateBrush, rc);
    else if (brush != null)
       e.Graphics.FillEllipse(brush, rc);
    if (grip.Action == AnnotationGripAction.Rotating && rotatePen != null)
       e.Graphics.DrawRectangle(rotatePen, rc.X, rc.Y, rc.Width, rc.Height);
    else if (pen != null)
       e.Graphics.DrawEllipse(pen, rc);
```

```
    }
    // Clean up
 if (rotateBrush != null && rotateBrush != brush)
      rotateBrush.Dispose();
    if (rotatePen != null && rotatePen != pen)
      rotatePen.Dispose();
    if (brush != null)
      brush.Dispose();
    if (pen != null)
      pen.Dispose();
    // Section: Now we need to restore the Graphics
    // state so any additional rendering will
    // not be affected by our change.
    e.Graphics.Restore(state);
    vm.Dispose();
}
```

Respond to events

Clicking, double clicking, hovering over, resizing, moving, or activating an annotation results in an event firing. That fact allows for easy customization and user interaction. AnnotationController events can be fired for all annotations, or for a specific annotation object. The ActiveAnnotation is always the annotation being interacted with.

HotSpot annotations allow end users to click an area of a document so that the application can respond.

ℹ Individual Annotation objects can respond to actions as events. Most events in the AnnotationController have the same event in the annotation object itself.

This example below shows a message box with the name of the HotSpotAnnotation the user clicked.

Example

Triggering a Message Box from a HotSpot AnnotationHotSpot.

C#

```
annotationViewer1.AnnotationClicked +=
 AnnotationEventHandler(annotation_AnnotationClicked);
...
private void annotation_AnnotationClicked(object sender, AnnotationEventArgs e)
{
    if (e.Annotation is HotSpotAnnotation)
    {
        MessageBox.Show(this, "The hotspot named " + e.Annotation.Name +
          " was clicked!", "Hot Spot Notification");
    }
}
```

Add a context menu to an annotation

The ContextMenu property of the Annotation class allows a pop-up menu to be displayed when any Annotation object is right clicked. This is a nice way to allow the user to edit and change properties of the active annotation.

The Annotation object in focus is set to the ActiveAnnotation property in the AnnotationController component. When using the AnnotateViewer, the active annotation property is accessed as annotateViewer1.Annotations.ActiveAnnotation. In the context menu item clicked event, you can change the properties of the active annotation, then refresh the controller.

Example

The following example demonstrates changing the background color of the annotation by:

1. Showing a color dialog
2. Checking for a Fill property and then
3. Adjusting the color.

Changing the Background Color

C#

```
private void menuBackColor_Click(object sender, System.EventArgs e){
// Display the pick color dialog.
ColorDialog dlg = new ColorDialog();
if (dlg.ShowDialog() == DialogResult.OK) {
    AnnotationBrush brush = new AnnotationBrush(dlg.Color);
    AnnotationUI annotation = annotateViewer1.Annotations.ActiveAnnotation;
    SetProperty(annotation.GetType(), "Fill", annotation, typeof(AnnotationBrush),
 brush);
}
dlg.Dispose();
}
private void SetProperty(Type annType, string propertyName, AnnotationUI annotation,
 Type valueType, object value)
{
    PropertyInfo info = annType.GetProperty(propertyName);
    if (info != null && info.CanWrite)
    {
        if (info.PropertyType == valueType)
            info.SetValue(annotation, value, null);
    }
}
```

Highlight a document

To highlight an area of a document, use any annotation that implements the IHighlighter interface and set the Translucent property to true.

Bright colors in the base image are replaced with the color of the annotation object. Dark colors will show through. The results are very similar to those you would get using a highlighter on a paper document.

The following annotations currently implement IHighlighter:

- EllipseAnnotation
- FreehandAnnotation
- LinesAnnotation
- PolygonAnnotation
- RectangleAnnotation

Work with layers and groups

This section discusses how to work with layers and groups.

Layers

LayerAnnotations are container objects that can hold an arbitrary number of annotations, including other LayerAnnotation objects. Layers can be used to differentiate pages in a multipage document. Typically, a layer holds annotations from each document page.

When the Visible property of a LayerAnnotation is set to false, none of its child annotations are rendered. This provides a convenient way to show or hide a collection of annotations.

Groups

A Group is a LayerAnnotation with its GroupAnnotation property set to true. When a layer works as a group, the grips are shown around the layer instead of the annotations it contains. Moving or resizing a group moves or resizes all of its annotations.

To create a group, begin by creating a new LayerAnnotation, adding annotations into the layer, and set its GroupAnnotation property to true. The AnnotationController has Group() and Ungroup() methods to simplify this process.

ℹ Top level layers, i.e. those in the Layers property of AnnotationController, cannot be used as a group. Only LayerAnnotations contained in other layers can act as a group.

Create a custom annotation

DotImage Annotations, like all of Atalasoft DotImage, are designed to be extensible. Should you encounter a case where the built-in annotation objects do not suffice, the polymorphic design of DotImage Annotations allows you to create arbitrary objects defined by any number of points.

The AnnotationData and AnnotationUI classes are the base of all annotations. The PointBaseData and PointsBaseAnnotation classes can be derived from to define an object as a series of points that can be interactively edited either by redefining the overall rectangle, or by repositioning any point represented as a grip.

In addition to the data and UI classes, each annotation has a renderer associated with the data class. The renderer implements IAnnotationRenderer and handles rendering the annotation and its grips.

The example that follows demonstrates on how to create a custom annotation, in this case, a triangle-shaped annotation.

Example

Step 1: Create the TriangleData class derives from AnnotationData.

Create the TriangleData class derives from AnnotationData. It implements ISerializable and IConeable.

C#

```
public class TriangleData : Atalasoft.Annotate.AnnotationData, ISerializable,
 ICloneable
```

Step 2: Add a Fill Property to the Annotation

Add a Fill Property to the Annotation.

The code below demonstrates raising the PropertyChanging event when a property value is being modified as well as adding an entry in the AnnotationUndoManager for this change.

C#

```
public AnnotationBrush Fill
{
 get { return _fill; }
 set
 {
     // If there is no change just ignore it.
     if (Equals(value, _fill))
         return;

     // Raise the PropertyChanging event so any derived classes are notified.
     var e = new AnnotationPropertyChangingEventArgs(this, "Fill", _fill, value);

     // If IgnoreDataChanges is true, we should not raise events. This can
     // happen when making internal changes that we don't want propagated.
     if (!IgnoreDataChanges)
     {
         OnPropertyChanging(e);
         if (e.Cancel) return;
     }

     // If we want this property to work with the UndoManager, we must
     // pass in an AnnotationUndo object for this change.
     var undo = new AnnotationUndo(this, "Fill", _fill, "Fill Change");
     var newValue = (AnnotationBrush)e.NewValue;

     // If dynamic changes are required for brush properties,
     // the SetBrushEvents and RemoveBrushEvents methods should
     // be used. These allow the brush properties to notify
     // the AnnotationController that a change was made.
     RemoveBrushEvents(_fill);
     _fill = newValue;
     SetBrushEvents(_fill);
```

```
    // Finally, pass the undo object to the AnnotationController.
    if (!IgnoreDataChanges)
    {
        OnAnnotationControllerNotification(
            new AnnotationControllerNotificationEventArgs(
                Atalasoft.Annotate.AnnotationControllerNotification.Invalidate,
 undo));
    }

    var eChanged = new AnnotationPropertyChangedEventArgs("Fill", _fill, newValue);
       OnPropertyChanged(eChanged);
 }
}
```

Step 3: Override the `Clone()` Method`Clone()`

You must override the `Clone()` method when deriving from AnnotationData.

C#

```
public override object Clone()
{
    TriangleData data = new TriangleData();
    base.CloneBaseData(data);
    data._fill = (this._fill == null ? null : this._fill.Clone());
    return data;
}
```

Step 4: Add serialization code

Add serialization code so this annotation can be serialized to XMP or WANG.

C#

```
public TriangleData(SerializationInfo info, StreamingContext context) : base(info,
 context)
{
    this._fill = (AnnotationBrush)SerializationInfoHelper.GetValue(info, "Fill", new
 AnnotationBrush(Color.Blue));
    base.SetBrushEvents(this._fill);
}
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter=true)]
public override void GetObjectData(SerializationInfo info, StreamingContext context)
{
    base.GetObjectData(info, context);
    info.AddValue("Fill", this._fill);
}
```

Step 5: Add serialization code

Add a method to return the points of the annotation in annotation space.

C#

```
public PointF[] GetTrianglePoints()
{
    PointF[] points = new PointF[3];
    points[0] = new PointF(0, this.Size.Height);
    points[1] = new PointF(this.Size.Width, this.Size.Height);
    points[2] = new PointF(this.Size.Width / 2f, 0);
```

```
    return points;
}
```

Step 6: Create a TriangleAnnotation class that derives from AnnotationUI and implements ISerializable

Add a method to return the points of the annotation in annotation space.

C#

```
public class TriangleAnnotation : Atalasoft.Annotate.UI.AnnotationUI, ISerializable
```

Step 7:Add the default and serialization constructor.

Add the default and serialization constructor as shown below.

C#

```
private TriangleData _data;
public TriangleAnnotation() : base(new TriangleData())
{
     this._data = this.Data as TriangleData;
     base.SetGrips(new RectangleGrips());
}
public TriangleAnnotation(SerializationInfo info, StreamingContext context) :
 base(info, context)
{
     this._data = this.Data as TriangleData;
     base.SetGrips(new RectangleGrips());
}
```

Step 8: Override the `GetRegion()` method

Override the `GetRegion()` method.

C#

```
public override AnnotationRegion GetRegion(AnnotateSpace space)
{
    AnnotationRegion region = new AnnotationRegion();
    SizeF size = this.Data.Size;
    // Specify the points in annotation space.
    PointF[] points = this._data.GetTrianglePoints();
    // Convert to the requested space if required.
    if (space == AnnotateSpace.Document)
 AnnotateSpaceConverter.AnnotationSpaceToDocumentSpace(this.Data, points);
    else if (space == AnnotateSpace.View)
    AnnotateSpaceConverter.AnnotationSpaceToViewSpace(this.Controller.Parent,this.Data,
 points);
    region.Path.AddPolygon(points);
    // Be sure to add the grips to the region.
    base.AddGripsToRegion(region);
    return region;
}
```

Step 9: Create a TriangleRenderingEngine class that will be used to render the annotation

Create a TriangleRenderingEngine class that will be used to render the annotation.

C#

```
public class TriangleRenderingEngine :
 Atalasoft.Annotate.Renderer.AnnotationRenderingEngine
```

Step 10: Override the `RenderAnnotation()` method

Override the `RenderAnnotation()` method and draw the triangle.

C#

```
public override void RenderAnnotation(AnnotationData data, RenderEnvironment e)
{
     // Perform a basic check.
     TriangleData data = annotation as TriangleData;
     if (data == null) return;
     if (data.Fill == null) return;

     // SetGraphicsTransform handles combining multiple
     // transformation matrix objects so you can render normally.
     base.SetGraphicsTransform(annotation, e);
     Brush b = base.CreateBrush(data.Fill);
     if (b != null)
     {
        PointF[] points = data.GetTrianglePoints();
        e.Graphics.FillPolygon(b, points);
        b.Dispose();
     }

     // If you call SetGraphicsTransform you must also
     // call RestoreGraphicsTransform when finished.
     base.RestoreGraphicsTransform(e);
}
```

Step 11: Add Annotation's Engine to AnnotationRenderers Collection

Before the annotation can be rendered, its rendering engine must be added to the AnnotationRenderers collection. A convenient way of doing this is with a static (Shared in VB) constructor in the data class.

C#

```
static TriangleData()
{
     Atalasoft.Annotate.Renderer.AnnotationRenderers.Add(typeof(TriangleData), new
 TriangleRenderingEngine());
}
```

Password-based authentication

A locked annotation object can have an associated password. Individuals with knowledge of that password are able to unlock and remove the object. Password based authentication is the simplest method for creating redactions.

The `Lock ()` method in each annotation has an optional password that is encrypted in the object and in the serialized data.

The `Unlock ()` method can then be invoked with the associated password to allow editing of the annotation.

Typically, the RectangleAnnotation with a solid alpha channel is used for redactions.

NT user authentication

If the application that uses redaction annotations needs NT user or role based authentication, use the Microsoft .NET Framework by creating a custom security class which implements the IAnnotationLock interface. See MSDN documentation for more details.

Burning annotations

The most secure way to prevent a user from viewing a section of a document is to permanently remove that section. You can do this interactively by placing an annotation over the protected area and burning it. If you are integrating with Atalasoft DotImage and using the AnnotateViewer component, see the Burn() method. Otherwise use the RenderAnnotations() method in the AnnotationController.

Example

Locking an Annotation.

C#

```
//code to lock an annotation so it cannot be edited interactively
RectangleAnnotation myAnnotation =
  new RectangleAnnotation(new RectangleF(2.0, 2.5, 4.2, 6.0));
myAnnotation.Fill = new AnnotationBrush(Color.Black);
myAnnotation.Data.Security = new AnnotationLock();
annotationViewer1.Annotations.CurrentLayer.Items.Add(myAnnotation);
myAnnotation.Data.Security.Lock(password);

//add code here to interact with the image
 //to unlock the annotation, simply call the unlock method.
//This allows the annotation to be edited interactively
if (myAnnotation.Data.Security.Unlock(password))
    MessageBox.Show(this, "Annotation Unlocked Successfully");
else
    MessageBox.Show(this, "Incorrect Password");
```

Create a sticky note

A sticky note is a TextAnnotation that appears within a shadowed rectangle with a yellow background and black text. To prevent the note from rotating with the image, a common preference with sticky notes, set the `CanRotate` property to false.

Example

Create a Sticky Note

C#

```
TextAnnotation myAnnotation = new TextAnnotation();
myAnnotation.Text = "This is a sticky note";
myAnnotation.Fill = new AnnotationBrush(Color.Yellow);
myAnnotation.Shadow = new AnnotationBrush(Color.Gray);
```

```
myAnnotation.ShadowOffset = new PointF(4, 4);
myAnnotation.Data.CanRotate = false;
annotationViewer1.Annotations.CreateAnnotation(myAnnotation);
```

Work with unit systems

DotImage Annotations are designed to work in any unit system. You can set the unit system using the Resolution property in IAnnotate . When this value is set to 1.0, the annotation objects are sized to pixel coordinates. Adjusting this value scales the coordinate system relative to pixels.

Example

To position and size objects in inches on a 96 dpi display, set the Resolution value to 96.

Units in the AnnotateViewer control

The AnnotateViewer control has a Units property. This automatically sets the Resolution to the appropriate value based on the resolution of the image in the control. Any value other than Custom overrides the Resolution to a value based on the image's resolution scaled to the appropriate unit.

This is convenient when viewing images such as maps in units such as kilometers. When the X and Y resolutions in the image differ, the control adjusts the aspect ratio and extends the height of the image. Annotation objects can be precisely positioned in familiar coordinates without the need to convert or track units in your own code.

Units in a custom IAnnotate control

Setting the Resolution property to a value (pixels per unit) allows you to specify the location and size of all annotation objects using any unit you wish. The effect is the same as setting the Units property in the AnnotateViewer to Custom and specifying a custom Resolution.

Annotate multipage documents

DotImage Annotations have built-in support for annotating multipage documents such as multipage TIFF images. Each page of the TIFF has its own annotations that are displayed when the page changes. When working with multipage images, Layers are used to separate annotations into each page.

Handling multipage images using AnnotateViewer

In the AnnotateViewer, annotating multipage images is very straightforward. If you set the `MultipageAnnotateMode` property to true, whenever a specific image in the ImageCollection is set to current, the annotations associated with that page are displayed and annotations associated with other pages are hidden. A layer is created for each page in the multipage document. The index of each layer in the AnnotationController is associated with the same index of each image in the ImageCollection in the same order.

When inserting, removing, or reordering images from the ImageCollection, you must manually insert, remove, or reorder layers to their respective positions.

Handling multipage images using a custom viewer

The best way to handle multipage images is by using a separate layer for each page in the image. If you do so, you need only set the `Visible` property for each layer to control which annotations are visible for a specific page. This approach is similar to that used in the AnnotateViewer.

Example

C#

```
this.annViewer.Annotations.Layers[page].Visible = true;
```

WANG annotations

When WANG data is read from a multipage TIFF, the data is stored separately for each page of the image. As a result, you must pull the annotation data from each TIFF page and add it to a separate layer.

The TiffDecoder can be used to read WANG data from each page. This is done by specifying which frame (page) index to read from when calling the GetImageInfo() method.

If the WANG data is being saved to a separate file instead of embedded into a TIFF, DotImage Annotations save each layer in such a way that it can be retrieved later, while still maintaining WANG compliance.

Metadata

Metadata is data that describes other data. Atalasoft DotImage Photo Pro and Document Imaging allow viewing and manipulation of metadata stored in an image.

The Atalasoft.DotImage.Metadata namespace contains classes that handle image metadata.

Metadata is a convenient way to store textual information in an image. Atalasoft DotImage allows this information to be accessed and manipulated. For example, it is possible to store the metadata information in a database, build a metadata viewer application , and to add your own metadata in the form of EXIF, IPTC, XMP, or COM markers.

See the Metadata Demo installed with Atalasoft DotImage for an example of metadata use.

Supported metadata types

Atalasoft DotImage supports the following metadata types:

- Digital camera EXIF tags
- IPTC tags
- COM Text markers
- Adobe XMP data
- TIFF Tags

- Photoshop Resources

In JPEG images, metadata is stored in "APPn markers". EXIF information is stored in an "APP1 marker", and IPTC and Photoshop Resource information is stored in an "APP13" marker. These markers are created automatically when a JPEG image is encoded. Alternatively, you can use a method to copy metadata without re-compressing JPEG images.

Image formats supporting metadata

The following Image Formats support Metadata:

Image Format	Operations	Metadata Types
JPEG	read/write	EXIF, IPTC, COM Text Markers, JpegMarkers, XMP, Photoshop Resources
PNG	read/write	COM Text Markers
TIFF	read/write	EXIF, IPTC, standard TIFF Tags, XMP, Photoshop Resources
PDF	write	XMP
PSD	read/write	Photoshop Resources, IPTC
RAW	read	EXIF

See the TiffFile class for lower level access to TIFF files. This allows you to read or write non-standard TIFF and EXIF Tags.

IPTC metadata

Together, the Newspaper Association of America (NAA) and the International Press Telecommunications Council (IPTC) have designed a model to store multiple types of data (metadata) in an image. This metadata is commonly known as IPTC. Adobe Photoshop, along with many members of the newspaper and press industry, uses IPTC to store information in images.

Atalasoft DotImage can read and write IPTC information in both JPEG and TIFF images.

Read IPTC data

Similar to EXIF metadata, you can read IPTC metadata from an image by using the IptcParser class which returns an IptcCollection. To read the metadata, pass the image filename or stream into the ParseFromImage() method of the IptcParser. The IptcCollection is populated with each IPTC tag it finds in the image. Then, you can modify, add, or remove IPTC tags as necessary.

The IptcTag object contains the IPTC information such as Section, ID, Index and Data. The Section is usually "2", the ID is the unique identification of the type of tag, the Index is the tag number (there can be multiple tags of the same ID), and Data is the actual value of the tag.

Modify and saving IPTC data

You can add, remove, or modify IPTC tags in the IptcCollection by using the appropriate collection Add, Insert, and Remove methods. Your changes to the tags are saved with the image by setting the IptcTags property of the JpegEncoder or TiffEncoder class.

Lossless modification of IPTC data

When you modify or add IPTC tags to an existing JPEG image, you should consider saving this information losslessly in order to avoid re-compressing the JPEG image.

The following example demonstrates how to load IPTC metadata, add and modify tags, and then save it back to the image losslessly.

Example

Losslessly Modifying IPTC Metadata

C#

```
string sourceFile = @"c:\C79A2086.jpg";
//First get metadata from an image without reading the entire image
IptcParser iptcParse = new IptcParser();
IptcCollection iptcData = iptcParse.ParseFromImage(sourceFile);

//Determine  if there is IPTC info
//Modify the caption if it exists, or add a caption if it does not
//An ID of 5 is the caption
if (iptcData == null)
iptcData = new IptcCollection();

if (iptcData != null && iptcData.LookupTag(5, 0) != null)
{
  //Get the index of this tag
  IptcTag tag = iptcData.LookupTag(5, 0);
  int index = iptcData.IndexOf(tag);
  iptcData[index].Data = "Atalasoft DotImage Created This";
}
else
{
  //Add the caption tag
  iptcData.Add(new IptcTag(2, 5, 0, "Atalasoft DotImage Created This"));
}

//Get a temporary filename
string tempFile = System.IO.Path.GetTempFileName();

//Save the metadata changes back to the image losslessly
JpegEncoder jpeg = new JpegEncoder();
jpeg.AppMarkers = new JpegMarkerCollection(sourceFile);
jpeg.IptcTags = iptcData;
jpeg.CopyJpegWithNewMarkers(sourceFile, tempFile);

//Delete the source file
System.IO.File.Delete(sourceFile);

//Move the temp file to the source file location
```

```
System.IO.File.Move(tempFile, sourceFile);
```

EXIF metadata

EXIF (Exchangeable Image File Format) is a standard for storing interchange information in image files, particularly JPEG images.

EXIF metadata is commonly used in digital camera images to store information specific to digital photography such as shutter speed, date taken, aperture, GPS information, and information specific to the make of the camera.

Atalasoft DotImage parses the EXIF metadata information from a JPEG, TIFF, and RAW images using the ExifParser class and loads each tag in the ExifCollection class.

An EXIF Tag is a special type of TIFF Tag, included in it's own SubIFD. Each EXIF tag is specified with an ID, an IFD (Image File Directory), and TIFF Tag Type. The IFD indicates a section of the EXIF data that holds a specific type of data.

The following table lists the available IFDs.

IFD	Description
Main	Primary image data information
Exif	EXIF camera information
Interoperability	Tags that store the information to ensure interoperability
Thumbnail	Thumbnail stored with the image
GPS	GPS satellite information
MakerNoteUnknown	Information specific to the make of an unknown camera
MakerNoteFujiFilm	Information specific to the make of a FujiFilm camera
MakerNoteOlympus	Information specific to the make of an Olympus camera
MakerNoteNikonType1	Information specific to the make of a Nikon Type 1 camera
MakerNoteNikonType3	Information specific to the make of a Nikon Type 2 or 3 camera
MakerNoteCasio	Information specific to the make of a Casio camera
MakerNoteCanon	Information specific to the make of a Canon camera

Maker notes

Atalasoft DotImage supports maker notes in some cameras. Because there is very little conformity between camera manufacturer's maker notes, support is limited. The camera manufacturer is free to change their own specification. We do our best to respond to the changes.

Atalasoft DotImage currently supports parsing of the following maker notes:

- FujiFilm
- Olympus
- Nikon

- Casio
- Canon

For maker note data that Atalasoft DotImage does not support, the data can still be persisted and manually parsed as in this case Atalasoft DotImage stores the entire chunk of data as a byte array in a single tag with an ID of 0xFFFF and the IFD set to ExifTagIfd.MakerNoteUnknown.

The EXIF Parser reads unknown EXIF Tags, that is tags whose format is not known to Atalasoft DotImage. This is particularly an issue with Maker Note data as manufacturers tend to add new tags in newer camera models. These tags are returned with their ID, and a byte array of data. It is up to the developer to parse these unknown tags. The byte array could represent a single integer value, a string, or a series of additional sub-tags.

Each unknown tag is identified as a separate tag unlike the case when the entire Maker Note chunk is unrecognized.

For more information about EXIF, please visit http://www.exif.org/.

COM text

PNG and JPEG images support a type of metadata called COM Text. COM text is simple textual information that you can store into an image. With Atalasoft DotImage, you can read and write this information to an image.

Read COM text

Similar to EXIF and IPTC metadata, COM text data can be read invoking the `ParseFromImage ()` method in the ComTextParser class to return a ComTextCollection of all tags in a PNG or JPEG image.

PNGs support a key/value pair, while JPEGs support only a value. In other words, the COMTextTag has both a key and text. The key is ignored when saving COM text into a JPEG image.

Modify and saving COM text

You can add, remove, or modify COM Text tags in the ComTextCollection by using the appropriate add, remove, and insert methods. To save changes to a JPEG or PNG image, set the ComText property in the JpegEncoder or PngEncoder classes.

TIFF tags

There are two interfaces for working with TIFF Tags. Both of the methods listed below can be used to retrieve and store standard TIFF tags.

- TiffDecoder's `GetTiffTag()` method
- TiffEncoder's TiffTags properties

The TiffFile class provides far more flexibility, and low level access to TIFF structure including retrieving and editing arbitrary TIFF Tags.

TIFF tag structure

Every TIFF image has a set of TIFF Tags containing descriptive information about the image data such as width, height, compression, color depth, strip size, and so forth.

TIFF Tags can also contain custom metadata such as image description, date, and custom binary data. A TIFF Tag is defined by

- a uniqueID,
- a Type (i.e ushort, long, rational, byte, ascii),
- the object Data associated with the tag.

The Data can be a single value, or an array of values. Test the the myTag.Data.GetType().IsArray value to determine if the data is an array of values.

Some tag values are actually 32-bit pointers to the actual tag data. When this is the case, the IsReference property is true. To access the actual data, invoke the LoadReferenceTagData in the TiffDirectory class.

TIFF tag ID's

The following TIFF Tag ID's are tags that are supported using GetTiffTag and when saving Tiff Tags using the TiffEncoder. The TiffFile class is not limited to these TIFF Tags. In that case, use the below as a quick reference.

ℹ See the official TIFF specification for a full list of TIFF Tags.

```
Tag name                ID      TIFF DataType
  TIFFTAG_ORIENTATION        274      VT_UI2
     Values:
        image orientation
        ORIENTATION_TOPLEFT     1    row 0 top, col 0 lhs
        ORIENTATION_TOPRIGHT    2    row 0 top, col 0 rhs
        ORIENTATION_BOTRIGHT    3    row 0 bottom, col 0 rhs
        ORIENTATION_BOTLEFT     4    row 0 bottom, col 0 lhs
        ORIENTATION_LEFTTOP     5    row 0 lhs, col 0 top
        ORIENTATION_RIGHTTOP    6    row 0 rhs, col 0 top
        ORIENTATION_RIGHTBOT    7    row 0 rhs, col 0 bottom
        ORIENTATION_LEFTBOT     8    row 0 lhs, col 0 bottom

  TIFFTAG_IMAGEWIDTH         256         VT_UI4
  TIFFTAG_IMAGELENGTH        257         VT_UI4
  TIFFTAG_BITSPERSAMPLE      258         VT_UI2

  TIFFTAG_COMPRESSION        259         VT_UI2
     Values:
        COMPRESSION_NONE        1       dump mode
        COMPRESSION_CCITTRLE    2       CCITT modified Huffman RLE
        COMPRESSION_CCITTFAX3   3       CCITT Group 3 fax encoding
        COMPRESSION_CCITTFAX4   4       CCITT Group 4 fax encoding
        COMPRESSION_LZW         5       Lempel-Ziv  & Welch
        COMPRESSION_OJPEG       6       6.0 JPEG
        COMPRESSION_JPEG        7       JPEG DCT compression
        COMPRESSION_NEXT        32766   NeXT 2-bit RLE
        COMPRESSION_CCITTRLEW   32771   #1 w/ word alignment
        COMPRESSION_PACKBITS    32773   Macintosh RLE
```

```
      COMPRESSION_THUNDERSCAN 32809   ThunderScan RLE

      codes 32895-32898 are reserved for
      ANSI IT8 TIFF/IT <dkelly@etsinc.com)
      COMPRESSION_IT8CTPAD    32895   IT8 CT w/padding
      COMPRESSION_IT8LW       32896   IT8 Linework RLE
      COMPRESSION_IT8MP       32897   IT8 Monochrome picture
      COMPRESSION_IT8BL       32898   IT8 Binary line art

      compression codes 32908-32911 are reserved for Pixar
      COMPRESSION_PIXARFILM   32908   Pixar companded 10bit LZW
      COMPRESSION_PIXARLOG    32909   Pixar companded 11bit ZIP
      COMPRESSION_DEFLATE     32946   Deflate compression
      COMPRESSION_ADOBE_DEFLATE   8   Deflate compression,
                                      as recognized by Adobe

      compression code 32947 is reserved for
      Oceana Matrix <dev@oceana.com>
      COMPRESSION_DCS         32947   Kodak DCS encoding

      COMPRESSION_JBIG        34661   IS3O JBIG
      COMPRESSION_SGILOG      34676   SGI Log Luminance RLE
      COMPRESSION_SGILOG24    34677   SGI Log 24-bit packed

TIFFTAG_PHOTOMETRIC     262         VT_UI2
   Values:
      PHOTOMETRIC_MINISWHITE  0    min value is white
      PHOTOMETRIC_MINISBLACK  1    min value is black
      PHOTOMETRIC_RGB         2    RGB color model
      PHOTOMETRIC_PALETTE     3    color map indexed
      PHOTOMETRIC_MASK        4    holdout mask
      PHOTOMETRIC_SEPARATED   5    color separations
      PHOTOMETRIC_YCBCR       6    CCIR 601
      PHOTOMETRIC_CIELAB      8    1976 CIE L*a*b*
      PHOTOMETRIC_LOGL        32844    CIE Log2(L)
      PHOTOMETRIC_LOGLUV      32845    CIE Log2(L) (u',v')

TIFFTAG_RESOLUTIONUNIT  296         VT_UI2
   Values:
      RESUNIT_NONE            1   no meaningful units
      RESUNIT_INCH            2   english
      RESUNIT_CENTIMETER      3   metric

TIFFTAG_XRESOLUTION        282         VT_R4
TIFFTAG_YRESOLUTION        283         VT_R4
TIFFTAG_NUMBEROFINKS       334         VT_UI2
TIFFTAG_DOCUMENTNAME       269         VT_BSTR
TIFFTAG_IMAGEDESCRIPTION   270        VT_BSTR
TIFFTAG_MAKE               271         VT_BSTR
TIFFTAG_MODEL              272         VT_BSTR
TIFFTAG_PAGENAME           285         VT_BSTR
TIFFTAG_SOFTWARE           305         VT_BSTR
TIFFTAG_DATETIME           306         VT_BSTR
TIFFTAG_ARTIST             315         VT_BSTR
TIFFTAG_HOSTCOMPUTER       316         VT_BSTR
TIFFTAG_INKNAMES           333         VT_BSTR
TIFFTAG_TARGETPRINTER      337         VT_BSTR
TIFFTAG_TILEWIDTH          322         VT_UI4
TIFFTAG_TILELENGTH         323         VT_UI4

TIFFTAG_INKSET             332         VT_UI4
   Values:
   INKSET_CMYK                1  CMYK image
```

```
   TIFFTAG_DOTRANGE           336         Binary

   TIFFTAG_EXTRASAMPLES       338         1-D SAFEARRAY or VT_UI2
      One value per 'extra' channel in the image.
      Values:
      EXTRASAMPLE_UNSPECIFIED 0    unspecified
      EXTRASAMPLE_ASSOCALPHA 1    associated alpha (pre-multiplied)
      EXTRASAMPLE_UNASSALPHA 2    unassociated alpha

   TIFFTAG_EXTRASAMPLES       338               Binary
       Purpose of any extra channels in the TIFF image.
   TIFFTAG_XMP_DATA           700          Binary
      Adobe XMP data
   TIFFTAG_ANNOTATIONS        32932        Binary
      Wang annotations data
   TIFFTAG_RICHTIFFIPTC       33723        Binary
      IPTC Data from the RichTIFF specification
   TIFFTAG_PHOTOSHOP          34377        Binary
      Private tag registered to Adobe for PhotoShop (IPTC data)
   TIFFTAG_ICCPROFILE         34675        Binary
      ICC profile block
```

XMP

Adobe's XMP Metadata is an effort to standardize on a well documented, easy to use metadata format, replacing the existing metadata formats such as EXIF and IPTC. Based on ASCII XML data, it's simple to parse and easily extensible.

Example

C#

```
XmpParser xmpParser = new XmpParser();
IXPathNavigable doc = xmpParser.ParseFromImage( file );
```

Example

Saving XMP Data

JPEG, TIFF, and PDF images can contain XMP data. To save an image with XMP, set the Xmp property in the JpegEncoder, TiffEncoder, or PdfEncoder classes.

Save metadata with an image

To save an image with metadata, the metadata specific properties must be set in the Image Encoder. For example, the IptcTags in the JpegEncoder and TiffEncoder, the Xmp property in the JpegEncoder, TiffEncoder, or PdfEncoder, and the AppMarkers property in the JpegEncoder. Saving EXIF Metadata in TIFF Files, requires use of the TiffFile object, in which case ExifTags must be set to an ExifCollection

Example

This example demonstrates how to load a JPEG image with EXIF information, querying the DateTime field, then re-saving the image back to a JPEG with just the EXIF tags intact.

C#

```
ExifParser exifParse = new ExifParser();
ExifCollection exifTags = exifParse.ParseFromImage(@"c\in.jpg");
//get JPEG Markers
JpegMarkerCollection appMarkersIn = new JpegMarkerCollection(@"c\in.jpg");
//read image
Workspace myWorkspace = new Workspace();
myWorkspace.Open(@"c\in.jpg");
//get the DataTime Tag from the image and display the value
if (exifTags != null)
{
   ExifTag tag = exifTags.LookupTag("DateTime");
   if (tag != null)
       MessageBox.Show("This photo was taken on " + tag.Data.ToString());
}
JpegEncoder jpeg = new JpegEncoder(75);
//only write EXIF Tags back to the image (APP1)
JpegMarkerCollection appMarkersOut = new JpegMarkerCollection();
foreach (JpegMarker mk in appMarkersIn)
   if (mk.Type == JpegMarkerTypes.MarkerApp1)
       appMarkersOut.Add(mk);
jpeg.AppMarkers = appMarkersOut;
myWorkspace.Save("c:\\out.jpg", jpeg);
```

Retrieve metadata from an image

Metadata can be retrieved from an image, without loading the image data at all, by using the ExifParser, IptcParser, ComTextParser, or XmpParser classes. Each metadata parser class allows a filename or stream to be passed into the `ParseFromImage()` method, which returns a collection with all tags of the associated type in the image.

See each of the Parser classes for an example of its use.

Example

The following code returns a collection of EXIF tags from a JPEG image.

C#

```
ExifParser exifParser = new ExifParser();
ExifCollection exifTags = exifParser.Parse("myimage.jpg");
```

Set metadata values

There are several ways to set PDF Translator's metadata.

PdfTranslator itself via its properties. When a document is translated through an OcrEngine object, the PdfTranslator imposes its document properties onto the OcrDocument. The PdfTranslator properties listed below are passed on directly to the output PDF.

Property Name	Type	Meaning
Title	string	Title of the document, initially set to the empty string ("")
Subject	string	Subject of the document, initially set to the empty string ("")
Author	string	Author of the document, initially set to the empty string ("")
Creator	string	Creator of the document, initially set to Atalasoft DotImage
Producer	string	Producer of the document, initially set to Atalasoft DotImage
Keywords	string	Key words associated with the document, initially set to the empty string ("")

Each property corresponds to a piece of metadata within the PDF specification. Additionally, the CreationDate property is set automatically to the current system time.

Example

Setting Metadata Properties Directly

While this is a simple mechanism for setting the metadata, there are times when it might not be optimal for the circumstances. For example, if the PdfTranslator is used within several different contexts, it might be difficult to set or reset the metadata values. In this case, it might be easier to set the OcrDocument metadata properties directly. This is best done using the DocumentProgress event within OcrEngine as shown below.

C#

```
public void SetMetadataHandler()
{
    engine.DocumentProgress += new OcrDocumentProgressEventHandler(myMetadataHandler);
}
private void myMetadataHandlerOnPagePreprocessing(object sender,
 OcrDocumentProgressEventArgs e)
{
    if (e.Document != null)
    {
        e.Document.Title = "An Illustrated Guide to Exhaust Manifold Cruciforms";
        // etc.
    }
}
```

ℹ It is important to check the Document property for null. In native translators (i.e., translators that are built-in to an OcrEngine), there may never be an OcrDocument. Since the event is a function of the engine and not the translator, the client must be prepared to handle this case.

It is possible to pre-populate the metadata fields of OcrDocument by creating a subclass of BasicOcrFactory and overriding the OcrDocument() method. In this method, the client instantiates a new OcrDocument object then sets its metadata properties directly.

By default, the PdfTranslator overwrites any existing metadata within the OcrDocument. Setting the PdfTranslator property SetDocumentMetadata to false prevents metadata from being overwritten.

Control PDF output characteristics

The properties are available for controlling the characteristics of the PDF Output are listed below.

Property Name	Type	Effect on Output	Default Value
UseTempFiles	bool	When set to true, creates temporary files for each page and each thumbnail. When set to false, a copy of each image and thumbnail is kept in memory.	true
TempPath	string	String representing a path to a folder for temporary image files.	
OutputType	PdfTranslatorOutputType	Controls the means of laying out text and images on each page of the file.	
UseDocument TextColor	bool	When set to true, the PdfTranslator uses text color as reported by the OcrEngine. When set to false, all text is rendered with the TextColor property.	
TextColor	Color	When UseDocumentTextColor is set to true, this color is used to render all text.	black
CompressionSelector	PdfCompressionSelector	Sets a delegate to use for selecting the compression used by an image.	
GenerateThumbnails	bool	When true, the PdfTranslator generates a thumbnail image for each page.	true
UseNormalized Baseline	bool	When set to true, the PdfTranslator makes sure that every word in a line has the same baseline, unless it is clearly superscript or subscript text. When set to false, individual word baselines are used.	true

Attach metadata to objects

You can attach metadata to both the OcrDocument and OcrPage objects. Metadata is stored in a Hashtable contained within the object. In the case of OcrDocument, there are certain pseudo properties, which actually access the Metadata object instead.

While Atalasoft DotImage dictates a small number of guidelines for working with OCR metadata, the use of Metadata is otherwise entirely up to the client.

In .NET any object type can act as a key to access a value within a Hashtable. Atalasoft DotImage metadata is always accessed with enumerated values.

Atalasoft DotImage reserves all metadata that is associated with enumerated values or integral data types.

In general, a client should adopt the following organizational mechanism for metadata:

- Use strings as keys.
- Use a hierarchy for metadata for different applications
- Group logically associated data

Retrieve a document title

It is strongly recommended that clients do not add application-specific values to this DocumentInfo Hashtable. While we intend to support this hashtable in the future, the layout may change significantly. Client applications should store their own metadata within the Metadata hashtable, associating it with non-numeric keys (such as strings).

Example

C#

```
string docTitle = (string)GetDocumentInfoValue(OcrDocumentInfoKey.Title);
```

Read EXIF information

When Atalasoft DotImage reads EXIF data, it attempts to read all supported tags and populate them in the ExifCollection object. You can then enumerate through the collection or retrieve a specific tag, ID or IFD. This example reads EXIF tags from a JPEG image and outputs each tag it to the console.

Example

C#

```
ExifParser exifparse = new ExifParser();
ExifCollection exifTags = exifparse.ParseFromImage("exif.jpg");
foreach (ExifTag tag in exifTags)
{
   Console.WriteLine(tag.ToString());
}
```

Read EXIF thumbnails

Most images that contain EXIF data also contain a thumbnail in the Thumbnail IFD. Atalasoft DotImage will read this thumbnail if it exists by setting the ThumbnailStream property in the ExifCollection. Note that in TIFF images, the thumbnail is usually the second frame in the image and it can be read normally by specifying 1 as the frame index.

Example

C#

Obtaining DPI information from a PSD File

```
```

```
private void ReadExif(string filename)
{
   //read the EXIF metadata and thumbnail
   ExifParser exifParse = new ExifParser();
   ExifCollection exif = exifParse.ParseFromImage(filename);
   AtalaImage thumb = new AtalaImage(exif.ThumbnailStream);
}
```

Store EXIF information

Atalasoft DotImage can store existing EXIF information in a new or existing JPEG image by copying the APP1 JPEG marker from the source to destination image, or it can recreate EXIF data after the data has been altered.

To store EXIF information when saving a new JPEG image, set the AppMarkers property of the JpegEncoder containing the APP1 EXIF marker. To transfer the data from one JPEG image to another without editing the EXIF data, use the CopyJpegWithNewMarkers() method in the JpegEncoder class.

Add an object to document metadata

A client might establish metadata for an OcrDocument by adding an object to an OcrDocument's metadata as shown below.

Example

C#

```
private static string frobozzKey = "Frobozz, Inc.";
...
myOcrDocument.Metadata[frobozzKey] = frobozzMetadata;
```

Since the client might be producing several applications that operate in OcrDocuments, it might make sense to make their Metadata itself a Hashtable which is accessed with a key defined by the application itself. Assume that the client is building an application called POSmart to sort out purchase orders from general correspondence. Assume further that there is an object of class FrobozzRoutingInfo which should be associated with the document.

Example

Accessing the FrobozzRoutingInfo Object

C#

```
private static string POSmartKey = "Frobozz, Inc.";
static Hashtable EstablishFrobozzMetadata(OcrDocument document)
{
    Hashtable ht;
    ht = document.Metadata[frobozzKey];
    if (ht == null) {
        document.Metadata[frobozzKey] = ht = new Hashtable();
    }
    return ht;
}
```

```
static Hashtable GetFrobozzMetadata(OcrDocument document)
{
    Hashtable ht;
    ht = document.Metadata[frobozzKey];
    if (ht == null) {
        throw new Exception("Unable to get metadata.");
    }
    return ht;
}
static void EstablishPOSmartMetadata(OcrDocument document, FrobozzRoutingInfo info)
{
    Hashtable ht = EstablishFrobozzMetadata(document);
    ht[POSmartKey] = info;
}
static FrobozzRoutingInfo GetPOSmartMetadata(OcrDocument document)
{
    Hashtable ht = GetFrobozzMetadata(document);
    return ht[POSmartKey];
}
```

Obtain DPI information from a .PSD File

Atalasoft DotImage supports reading and writing Adobe Photoshop Resource Blocks from PSD, JPEG, and TIFF images. These resources store non-pixel data such thumbnails, ICC Color Profiles, IPTC Data, DPI resolution information, and textual information.

The PhotoshopResourceParser class obtains a collection of PhotoshopResource items. Each resource contains an ID, an object containing a byte array of the raw data, and an optional description of the resource. This data could be string data with a header, an integer (Motorola Byte Order), a byte array, or other custom data structures. To obtain the format specification, join the Adobe's developer network.

The object stored in the Data property of the PhotoshopResource must be cast to a byte array. Future versions may include a setting to request parsing of the byte array data into a typed object such as an integer or string.

This example demonstrates how to obtain the DPI information from a PSD file.

Example

C#

Obtaining DPI information from a PSD File

```
PhotoshopResourceParser psdParser = new PhotoshopResourceParser();
PhotoshopResourceCollection psdResources = psdParse.ParseFromImage("test.psd");
byte[] dpiData = (byte[])psdResources.LookupResource(1005);
```

Losslessly copy metadata

The CopyJpegWithNewMarkers() method in the JpegEncoder class can be used to losslessly save existing metadata to a copy of an existing JPEG image.

One use of this function is to modify IPTC data in a JPEG.

If you do not want to create a copy of the file, you must save the output to a temporary file, and then overwrite the existing file with your own code.

Document and image formats

Atalasoft DotImage supports the following document and image formats.

Introduction to PDF technology

Atalasoft DotImage provides customers industry leading PDF technology included with Atalasoft DotImage Document imaging and the following Atalasoft DotImage add-ons:

- Atalasoft DotImage PDF Reader Module for high speed PDF viewing and text extraction.
- Atalasoft DotImage Document Imaging for generating image-only PDF documents.
- Atalasoft DotImage OCR Searchable PDF Module for generating searchable PDF documents from images using OCR.
- Atalasoft DotImage PDF Annotations for annotating PDF documents.
- Atalasoft DotImage PDF Document editing tools

New to Atalasoft DotImage 7.0 is the ability to generate PDF/A-1b documents. PDF/A is a specific standard, ISO 19005-1:2005 (external link), which is suited for long-term electronic archiving. It restricts certain features as well as enforcing requirements to preserve the visual appearance of the document. All images must include color profiles to ensure proper color reproduction. All fonts must be embedded within generated PDF documents. Image compression is restricted to a set that does not include JBIG or JPEG2000. Atalasoft DotImage handles this work for you.

Starting with version 11.1, Atalasoft DotImage allows you to create PDF/A-2b documents. The second part of the PDF/A standard allows you to use Compressed objects and XRef streams (for smaller file sizes) and JBIG and JPEG2000 compressions.

Note that adding PDF annotations to an existing PDF/A document will not create PDF/A compliant documents.

Text extraction in PDF Reader

The Atalasoft DotImage PDF Reader add-on (formerly PDF Rasterizer) provides the ability to extract text from PDF files. It provides two classes as detailed in the table below.

Class	Description
PdfTextReader	• Derived from System.IO.TextReader • Use to read text from a PDF page, a set of pages, or the entire document in a stream-like fashion
PdfTextPage	• Determine the number of characters on a page • Extract characters based on character index or count • Determine the bounding box for a character • Determine the bounding box for a range of characters • Determine the character index for a given point • Convert between PDF user coordinate space (paper) and image coordinate space

Opening PDF's with a PdfTextDocument

The first step in extracting text from a PDF is to create a PdfTextDocument. PdfTextDocument objects implement IDisposable, so you must call `Dispose()` when you are done with the object. The easiest way to do so is to create the object in a using block.

The example that follows shows how to get the number of pages from a PdfTextDocument given a Stream that contains a PDF:

C#

```
private void ReadExif(string filename)
public int GetPageCount(Stream s)
{
     using (PdfTextDocument doc = new PdfTextDocument(s))
     {
         return doc.PageCount;
     }
}
```

Once you have a document, you can get PdfTextPage objects from it. The code below gets the number of characters on the first page of a PDF from a Stream.

```
public int GetCharCount(Stream s, int pageNum)
{
   using (PdfTextDocument doc = new PdfTextDocument(s))
   {
         PdfTextPage textPage = doc.GetPage(pageNum);
         return textPage.CharCount;
    }
}
```

Reading text with a PdfTextReader

PdfTextReader inherits from TextReader thereby giving you access to the text in a PDF. PdfTextReader objects are obtained from PdfTextDocument objects by calling GetPdfTextReader().

The example that follows shows you how to create a PdfTextReader and read all of its text.

Example

C#

```
public String ReadTextFromPages(Stream s)
{
     using (PdfTextDocument doc = new PdfTextDocument(s))
     {
         PdfTextReader rdr = doc.GetPdfTextReader();
         return rdr.ReadToEnd();
      }
}
```

You can call GetPdfTextReader with a single page number, or a range of pages.

Using a PdfTextPage to Extract Text

A PdfTextPage can do much more than simply telling you the number of characters on a page. The following example show how to get text from a page by providing an index and a count.

Example

C#

```
public String GetText(Stream s, int pageNum, int index, int count)
     {
         using (PdfTextDocument doc = new PdfTextDocument(s))
         {
             PdfTextPage textPage = doc.GetPage(pageNum);
             return textPage.GetText(index, count);
         }
}
```

As a range of text can span lines, it may have multiple bounding boxes. For this reason, a request for the bounding boxes returns an array of bounding boxes (which need not be rectangular). This next example shows how to determine the bounding boxes of a range of text.

C#

```
public QuadrilateralF[] GetText(Stream s, int page, int index, int cnt)
{
  using(PdfTextDocument doc = new PdfTextDocument(s))
  {
     PdfTextPage textPage = doc.GetPage(page);
     return textPage.GetBoxes(index, cnt);
  }
}
```

These quadrilaterals returned are in PDF User Space, that is, they are in the PDF coordinate system with the origin at bottom-left, increasing Y values as you go from bottom to top.

If you need to translate one of these quadrilateral to one that could be used on a rasterized version, use PdfTextPage. ConvertPdfUnitsToPixels(), which takes the PDF User Space quadrilateral and the resolution that the image was rendered at.

In the PDF Demo project, the class PdfFindHighlighter shows how to turn quadrilaterals from PdfTextPage into RectangleAnnotation objects you can use to highlight characters. You can also use this class in your own projects.

Using a PdfTextPage to Search for Text

A PdfTextPage can also search for text automatically. The following example shows how to do such an automatic search.

C#

```
public void Search(Stream s, int pageNum, int index, string txt)
{
   using (PdfTextDocument doc = new PdfTextDocument(fs))
   {
      PdfTextPage p = doc.GetPage(pageNum);
      using (PdfSearchResults res = p.Search(index, txt, false, false))
      {
         while (res.FindNext())
      {
            // res.StartIndex and res.CharCount have
            // the results of the search
            // you could call p.GetText() or p.GetBoxes() here
         }
      }
   }
}
```

See the Atalasoft DotImage PDF Demo project for an example of how to search across an entire document with the class PdfDocumentSearch as well as how to call delegates every time a new instance is found. You can use this class in your own projects.

In the demo, it is wired to a PdfFindHighlighter object that puts RectangleAnnotation objects on the image to show found text.

Editing PDF documents

Atalasoft DotImage provides tools for high level editing and composition of PDF documents.

With these tools you can:

- Rearrange, add, or remove pages from existing PDF documents.
- Split existing PDF documents into separate documents.
- Combine any number of existing PDF documents into a single document.
- Create or edit book marks for navigation.
- Create or edit document metadata, including title, subject, author, creator, keywords and custom metadata.
- Encrypt or decrypt documents.

All this functionality is tied into a simple object model that doesn't require the programmer to memorize the PDF specification. Much of this functionality has been extended to cover existing PDF generation tools, including the Document Imaging tool for generating image-only PDF documents and the OCR Searchable PDF module.

Create searchable PDFs with OCR

The PdfTranslator class allows client applications to generate high quality PDF documents from scanned documents. The Atalasoft DotImage PdfTranslator provides the following features:

- Setting PDF Metadata fields
- High quality thumbnail images
- Accurate text placement
- Text-Under-Image placement
- Optional placement of picture regions
- Automatic or client-controlled image compression
- Advanced codec support (JBIG2, JPEG 2000)
- Insertion of client synthesized pages
- Creation of PDF/A-1b and PDF/A-2b documents

PDF file format

Adobe created the PDF file format to enable the encapsulation of any document that could be printed digitally so that it retains its content as text, images or graphics in as high quality as possible, typography as intended, and accurate color representation. The file format is an object-oriented format that describes a document as a series of pages, each of which is represented by a list of high-level drawing and compositing operations that were modeled after the PostScript imaging model. In addition to page content, a document could also include interactive features, navigation tools, dynamic forms, and multimedia. PDF was meant to be a publication format, rather than an editable format. Generation of PDF was considered to be a final step in the creation of a document.

Atalasoft DotImage provides the means to break into the PDF model for the purpose of making PDF documents accessible for common operations. For example, an insurance company could keep a stock collection of informational documents that could be assembled into a customized PDF document tailored for a client. This type of operation in Atalasoft DotImage can be done in a single line of code.

Where possible, the details of PDF structure are hidden from client code. Instead, client code works with higher level objects, such as documents, pages, bookmarks and actions, packaged in familiar .NET objects and collections.

PDF page coordinates

Pages in PDF documents have a strongly defined coordinate system. Pages are based on standard Cartesian coordinates in the first quadrant. In other words, when looking at a page, the coordinates of the lower left hand corner are (0, 0), with X extending positively to the right and Y extending positively up.

The PDF format is without resolution. This means that pages should be displayable at any zoom with consistent fidelity. Pages, however do have measurements. Pages are measured in default user space units, which are 1/72 inch. Although it is possible for a PDF file to specify other units for page measurement, it is rare.

A letter sized page will therefore be 612 units wide and 792 units high. An A4 sized page will be approximately 595.276 units wide and 841.89 units high.

Conversion of units from one space to another can be done with a set of classes for that manage conversion to and from PDF page space and pixels measured in either inches or centimeters. These classes are DpiToPdfCoordinateConverter, DpcmToPdfCoordinateConverter, PdfToDpiCoordinateConverter and PdfToDpcmCoordinateConverter. See *How To Convert AtalaImage Coordinates to PDF Coordinates* for more information.

PDF document objects

At the heart of the PDF manipulation API is a class called PdfDocument. This class represents the main structure of a document and the pages it contains. When a PdfDocument object is created, the PDF is briefly scanned to extract information about the pages and other document structures, but none of the pages themselves are loaded. Within a PdfDocument object is the metadata associated with the document, the tree of bookmarks, and a collection of PdfPage objects, one for each page in the PDF.

PDF bookmarks

PDF documents can contain a collection of bookmarks that can be used to help navigate the document. The structure of the bookmarks is hierarchical, but there are no requirements imposed on the hierarchy. The bookmarks can be a single list or a tree. Any structure imposed on the collection is at the discretion of the author.

In PDF, a bookmark is string displayed in the UI, some text appearance properties, and a collection of actions to be performed when the bookmark is clicked. PDF offers a wide variety of possible actions, all of which can be read and rewritten in Atalasoft DotImage, but at the release of 9.0 Atalasoft DotImage only offers the direct authoring of "Go To View" actions. This type of action specifies a destination within the document. This destination specifies the page, a mode for viewing the page and possible parameters to more finely control the mode. For example, a destination may include "Page 18, fit page to width, view from the top of the page". Another might be "Page 2, view on location (0, 0), use the current zoom".

To author PDF bookmarks, all of the PDF generation tools in Atalasoft DotImage or its add-ons include a property called BookmarkTree which represents the desired collection of bookmarks for the final document. In all cases, the BookmarkTree object is used late in the process of generating the document. This allows client code the freedom to build the bookmark tree incrementally before the PDF document is finalized.

View a PDF image

The PdfReader assembly includes PdfDecoder which derives from ImageDecoder. It acts like any other Atalasoft DotImage decoder in that it has a `Read()` method which returns an AtalaImage of the decoded image, in this case a PDF page. It also can be included in the RegisteredDecoders collection which is used when opening images using the AtalaImage constructor or Workspace.Open.

In the example that follows demonstrates how to view a PDF image. Included with the SDK is source for a full multi-threaded multipage PDF Viewer.

Example

C#

```
using Atalasoft.Imaging;
using Atalasoft.Imaging.Codec;
using Atalasoft.Imaging.Codec.Pdf;
...
//register the PdfDecoder
PdfDecoder pdf = new PdfDecoder();
RegisteredDecoders.Decoders.Add(pdf);
//read and display the PDF
myWorkapceViewer.Open("pdfdocument.pdf");
```

Translate a set of images to searchable PDF

The example that follows shows how to translate a set of images to PDF.

To translate a set of images to PDF, write code similar to that shown below.

Example

C#

```
public void TranslateToPdf(OcrEngine engine, ImageSource images, Stream outputStream)
{
    engine.Translate(images, "application/pdf", outputStream);
}
```

Use advanced PdfTranslator controls

Managing the translation of an OcrDocument into PDF creates a number of challenging timing issues as described below.

Timing issues

The Atalasoft DotImage OCR does not translate an OcrDocument until the document is complete. A client can manipulate a recognized document (spell check, re-order pages, and so forth.) before the translation happens.

Unfortunately, by the time translation actually occurs, the images used for OCR are no longer available or have been substantially altered by the engine. This makes it tricky to create thumbnail images for pages or to place the original image over the text in its maximum or native bit depth.

To overcome this problem, when a set of images is translated directly by the engine, the PdfTranslator hooks itself into OcrEngine events, make copies of images, and saves them in temporary files that can be used to create thumbnail images. When translation is complete, the PdfTranslator removes itself from the OcrEngine's events.

If an OcrDocument has been recognized (via one of the Recognize() methods), there are no overlay images associated with pages nor are there thumbnail images since the PdfTranslator was not asked to generate this information.

There are two ways to handle this problem:

- The client can request that the PdfTranslator hook itself into events before recognition and unhook itself at the end.
- The client can add the extra information manually.

To add the information manually, the client should hook into the DocumentProgress event as shown in the example below.

The following example demonstrates how to hook into the document progress event.

Example

C#

```
public void HookIntoDocumentProgress(OcrEngine engine)
{
   engine.DocumentProgress += new
 OcrDocumentProgressEventHandler(MyDocumentProgressHandler);
}
```

Next, allow the PdfTranslator to hook into the engine.

C#

```
public void MyDocumentProgressHandler(object sender, OcrProgressEventArgs e)
{
    if (e.Stage == OcrDocumentStage.BeginDocument) {
        myTranslationObject = myPdfTranslator.Prepare(e.Engine, e.Document);
    }
}
```

After the document has been recognized and the client has completed any editing, call the translator as shown below.

C#

```
myPdfTranslator.Translate(myEngine, recognizedDocument, "application/pdf",
destinationFile, myTranslationObject);
myPdfTranslator.Finish(myEngine, recognizedDocument, true, myTranslationObject);
```

If you need to add in pages that have not been generated by the OcrEngine itself, you must generate the extra information manually.

Add PdfTranslator to Engine's translator collection

The following example shows how to add PdfTanslator to an engine's translator collection.

To use the PdfTranslator, you must add it to the Translators collection of the desired engine as shown below. This installs the PdfTranslator into the engine so that it can be used like any other type of translator.

Example

MIME type for PDF

The MIME type for PDF is application/pdf.

C#

```
private PdfTranslator _myPdfTranslator;
private void InstallPdfTranslator(OcrEngine engine)
{
    _myPdfTranslator = new PdfTranslator();
    engine.Translators.Add(_myPdfTranslator);
}
```

Manually generate PdfTranslator metadata

ℹ The level of access to PdfTranslator internal structures described below is highly advanced and should not be undertaken lightly.

An OcrPage has metadata available for general client use. While a document is being recognized and translated by the PdfTranslator, the PdfTranslator stores metadata in each OcrPage.

The PdfPageInfo class contains information on how to build this particular page and how to set options for it. The properties listed in the table that follows typically are inherited from the PdfTranslator:

Name	Type	Meaning
OutputType	PdfTranslatorOutputType	Type of page generated. For example, text only, image over text, and so forth.
UseDocumentTextColor	bool	Value is true if the text color should be taken from the page rather than overridden.
TextColor	Color	Color to be applied to document text if UseDocumentTextColor is false
CompressionSelector	PdfCompressionSelector	A delegate that can be used to select the compression used for images.
GenerateThumbnails	bool	Property used by the PdfTranslator during recognition, not during translation. Its value is inconsequential if set on an OcrPagein an existing OcrDocument.

Retrieve metadata
The following examples show how to:

- Retrieve metadata from an object
- Access the PageInfo object

To access this information, retrieve the object in the page's metadata using the PdfPageInfo key as shown below.

C#

```
object pdfMetadata = page.Metadata[OcrPageMetadataKey.PdfPageInfo];
```

Access PageInfo object

This object is a Hashtable. The PageInfo object within the Hashtable can be accessed using the key OcrPdfPageMetadataKey.

To access this information, retrieve the object in the page's metadata using the PdfPageInfo key as shown below.

C#

```
Hashtable ht = (Hashtable)pdfMetadata;
 object pdfData = ht[OcrPdfPageMetadataKey.PageInfo];
 if (pdfData == null || !(pdfData is PdfPageInfo))
       throw new Exception("No PdfPageInfo in OcrPage Metadata.");
 PdfPageInfo pdfInfo = (PdfPageInfo)pdfData;
```

Add support for reading PDFs

Use the following code to add support for reading PDFs.

C#

```
Atalasoft.Imaging.Codec.RegisteredDecoders.Decoders.Add(
    new Atalasoft.Imaging.Codec.PdfDecoder());
```

Print a PDF image

The following samples demonstrate how to print a PDF image efficiently.

C#

```
using System.IO;
using System.Drawing.Printing;
using Atalasoft.Imaging.Codec.Pdf;
...
private Pages imagesToPrint = null;
private int current = 0;
private void button1_Click(object sender, System.EventArgs e)
{
        OpenFileDialog d = new OpenFileDialog();
        d.Filter = "PDF files(*.pdf) | *.pdf";
        if (d.ShowDialog() == DialogResult.OK)
        {
                using (var fs = new FileStream(d.FileName, FileMode.Open))
                {
                Document theDoc = new Document(fs);
                this.imagesToPrint = theDoc.Pages;
                // Use System.Drawing.Print.PrintDocument
                PrintDocument thePrintDoc = new PrintDocument();
                thePrintDoc.PrintPage +=new
 PrintPageEventHandler(thePrintDoc_PrintPage);
                this.current = 0;
                thePrintDoc.Print();
                }
        }
}
private void thePrintDoc_PrintPage(object sender, PrintPageEventArgs e)
{
        e.HasMorePages = true;
        Page p = imagesToPrint[current++];
        // fit to page, only when image is too large.
 float newX = (float)(e.PageBounds.Width / p.Width);
        float newY = (float)(e.PageBounds.Height / p.Height);
        if (!(newX > 1 && newY > 1))
                e.Graphics.ScaleTransform(newX, newY);
        // Draw pdf image onto graphics object here.
        p.Draw(e.Graphics);
        if (current >= imagesToPrint.Count)
                e.HasMorePages = false;
}
```

Create PDF/A documents

Atalasoft DotImage has the ability to generate PDF/A-1b or PDF/A-2b compliant documents. Both the PdfEncoder object and the OCR PdfTranslator object support this.

To support this in the PdfEncoder, you can set it in the constructor

C#

```
using Atalasoft.Imaging.Codec.Pdf;
using Atalasoft.PdfDoc;
// set the document type in the constructor
PdfEncoder encoder = new PdfEncoder(PdfDocumentType.PdfA1b);
```

In addition, you can set this using the DocumentType property.

C#

```
using Atalasoft.Imaging.Codec.Pdf;
using Atalasoft.PdfDoc;
// set the document type by changing a property
PdfEncoder encoder = new PdfEncoder();
encoder.DocumentType = PdfDocumentType.PdfA1b;
```

The PdfTranslator add-on to the OCR searchable PDF module has an identical constructor and an identical DocumentType property and can be used in the same manner.

Compressed Object and XRef streams

Object streams and XRef streams allow to reduce size of the document, but to increase the creation time of the document.

To create a PDF document with object streams and XRef streams, the `UseCompressedObjectStreams` flag should be set to true:

```
PdfEncoder encoder = new PdfEncoder
{
  UseCompressedObjectStreams = true
};

PdfTranslator trans = new PdfTranslator
{
  UseCompressedObjectStreams = true
};
```

PDF/A-1 does not support compressed object streams.

Compressed object streams in linearized PDFs are not supported.

Create PDF 2.0 documents

PdfEncoder and PdfTranslator support the creation of PDF 2.0 documents.

PDF 2.0 in PdfTranslator and PdfEncoder enables you to create password-protected PDF documents with AES-256 encryption. To do this, set `PdfVersion = 2.0` and pass the password to the method SetOneTimePasswords().

Sample for PdfEncoder:

```
var encoder = new PdfEncoder {PdfVersion = 2.0};
    encoder.SetOneTimePasswords(password);

using (var image = new AtalaImage(@"image.tif"))
using (var result = File.Create("pdf_2_0_document.pdf"))
```

```
{
     encoder.Save(result, image, null);
}
```

Sample for PdfTranslator:

```
using (GlyphReaderEngine engine = new GlyphReaderEngine())
{
     PdfTranslator xlate = new PdfTranslator {PdfVersion = 2.0};
     xlate.SetOneTimePasswords(password);
     engine.Translators.Add(xlate);

     engine.Initialize();
     FileSystemImageSource source = new FileSystemImageSource(
          new string[] { file }, false);

     using (var stm = new FileStream(outputFile, FileMode.Create,
 FileAccess.ReadWrite))
           engine.Translate(source, "application/pdf", stm);
}
```

When saving a password-protected documents with version 1.7 and earlier, RC4 encryption algorithm is used.

All other features of PDF 2.0 standard are not applicable to PdfTranslator and PdfEncoder.

Create an image only PDF document

There are several ways to create an image-only PDF document. If you're working with a single AtalaImage, the simplest way is to use a PdfEncoder directly.

C#

```
public void SaveAsPdf(Stream stm, AtalaImage image)
{
    PdfEncoder encoder = new PdfEncoder();
    encoder.Save(stm, image, null);
}
```

C#

```
using Atalasoft.Imaging.Codec.Pdf;
using Atalasoft.PdfDoc;
// set the document type by changing a property
PdfEncoder encoder = new PdfEncoder();
encoder.DocumentType = PdfDocumentType.PdfA1b;
```

You can generate PDF/A-1b files by passing in PdfDocumentType.PdfA1b to the PdfEncoder constructor.

In many cases you may want to generate PDF files from several images. Keeping them all in memory at once is inefficient. In these cases, it's best to use an ImageSource object such as FileSystemImageSource to provide the images to PdfEncoder.

C#

```
public void SaveAsPdf(Stream stm, ImageSource images)
{
    PdfEncoder encoder = new PdfEncoder();
```

```
    encoder.Save(stm, images, null);
}
```

In this case, PdfEncoder takes care of loading and disposing images for you.

Convert AtalaImage coordinates to PDF coordinates

Given an AtalaImage, it is easy to convert its coordinate space to PDF coordinate space. Given the image's resolution units, you select an appropriate converter. This can be done with the following code:

```
public static PdfCoordinateConverter GetConverter(AtalaImage image)
{
    if (image.Resolution.Units == DotsPerCentimeter) {
        return new DpcmToPdfCoordinateConverter(image.Resolution.X, new
 Size(image.Width, image.Height));
    }
    return new DpiToPdfCoordinateConverter(image.Resolution.X, new Size(image.Width,
 image.Height));
}
```

Convert one space to another.

```
PdfCoordinateConverter converter = GetConverter(myImage);
Point pdfPoint = convert.Convert(new Point(x, y)); // x and y in image coordinate space
```

Author PDF bookmarks

To author PDF bookmarks, you need three basic objects: a PdfBookmarkTree object to hold a collection of bookmarks, one or more PdfBookmark object to represent the actual bookmarks and a PdfAction object which represents what will happen when the PdfBookmark has been clicked.

Bookmark tree:

Making a bookmark tree is easy.

```
PdfBookmarkTree tree = new PdfBookmarkTree(); // makes a new empty bookmark tree
```

This initial tree has no bookmarks in it. It's main property, Bookmarks, is a list of PdfBookmark objects that represent the top-level bookmarks for the PDF document. To make a bookmark, you simply call the constructor. The default constructor will make a PdfBookmark with no text and with no action. There are other constructors that let you set the various properties as well. The most commonly used constructor takes a string and an action.

Make a Bookmark

```
PdfBookmark bookmark = new PdfBookmark("Introduction", null); // a null action will be
 ignored - we can set it later
```

The only available action presently is the "Go To View" action. This is usually constructed with a destination for the action. A destination is a combination of a viewing mode, some parameters and a page reference. Page references are an abstract notion of a page. Presently, they only represent a page in the current document associated with the action, but in the future they may also be references to embedded or external PDF documents or external files. To make a destination, you can use the default constructor directly, but the PdfDestination class has a number of static factory methods to make it easier to author specific view modes and to ensure that the parameters are correct.

Make an Action

```
PdfDestination dest = PdfDestination.FitPage();
dest.Page = new PdfIndexedPageReference(3); // page numbers a 0-based.  0 is the first
 page.  3 is the 4th page.
PdfGoToViewAction action = new PdfGoToViewAction(dest);
```

Putting it all together

Tying this together is a matter of association these objects with each other. We can do this in any logical order:

```
PdfBookmarkTree tree = new PdfBookmarkTree(); // make the tree
PdfBookmark bookmark = new PdfBookmark("Introduction", null); // make a bookmark
PdfDestination dest = PdfDestination.FitPage(); // make a destination
dest.Page = new PdfIndexedPageReference(3);
PdfGoToViewAction action = new PdfGoToViewAction(dest); // make an action
bookmark.ClickAction.Add(action); // add the action to the bookmark
tree.Bookmarks.Add(bookmark); // add the bookmark to the tree
```

The collection of bookmarks in PDF can be a tree structure. In order to describe the hierarchy, each PdfBookmark object has a property called Children, which is a collection of PdfBookmark objects that will be presented as a sub tree of that bookmark. By default, the list of the Children in any PdfBookmark object is empty. The maximum depth to which PdfBookmarks can nest is limited only by the maximum number of objects that are allowed within a PDF document - about 10 billion. The only real restriction is that the tree may not have cycles in it. In other words, a PdfBookmark object may not be put in any Children for which it is a parent or in a parent chain.

Add bookmarks with the PDF encoder

To author PDF bookmarks, you need three basic objects: a PdfBookmarkTree object to hold a collection of bookmarks, one or more PdfBookmark objects to represent the actual bookmarks and a PdfAction object which represents what will happen when the PdfBookmark is clicked.

```
FileSystemImageSource source = new FileSystemImageSource(@"multipage.tif", true);
PdfEncoder encoder = new PdfEncoder();
encoder.BookmarkTree = new PdfBookmarkTree();
using (FileStream stm = new FileStream("output.pdf", FileMode.Create))
{
    int i = 0;
    encoder.Save(stm, source, (sender, e) =>
    {
        if (e.Current == 0)
            return;
        PdfDestination dest = PdfDestination.FitWidth(null);
        dest.Page = new PdfIndexedPageReference(i);
        PdfGoToViewAction action = new PdfGoToViewAction(dest);
        PdfBookmark mark = new PdfBookmark("Page " + (i + 1), action);
        encoder.BookmarkTree.Bookmarks.Add(mark);
        i++;
    });
}
```

Add bookmarks with the PdfTranslator

To create bookmarks on a page-by-page basis with the PdfTranslator, it is easiest to hook into the PageConstructed event of the OcrEngine object to generate bookmarks for that page. In this event, you will have access to the entire structure of the OcrPage that has just been recognized, so it is

possible to use contextual information on the page to generate more specific bookmarks or to generate hierarchical information for the bookmark tree.

In this example, the code generates a simple bookmark with the page number and the view mode set to FitPage.

```
void engine_PageConstructed(object sender, OcrPageConstructionEventArgs e)
{
    OcrEngine engine = sender as OcrEngine;
    if (engine == null)
        return;
    PdfTranslator pdfTranslator = null;
    foreach (ITranslator translator in engine.Translators)
    {
        pdfTranslator = translator as PdfTranslator;
        if (pdfTranslator != null)
            break;
    }
    if (pdfTranslator == null)
        return;
    PdfDestination dest = PdfDestination.FitPage();
    dest.Page = new PdfIndexedPageReference(_pageNumber);
    PdfGoToViewAction action = new PdfGoToViewAction(dest);
    PdfBookmark mark = new PdfBookmark("Page " + (_pageNumber + 1), action);
    pdfTranslator.BookmarkTree.Bookmarks.Add(mark);
    _pageNumber++;
}
int _pageNumber;
void TranslateToPdf(FileSystemImageSource source, string outputPath)
{
    GlyphReaderEngine engine = new GlyphReaderEngine();
    PdfTranslator xlator = new PdfTranslator();
    engine.Translators.Add(xlator);
    xlator.BookmarkTree = new PdfBookmarkTree();
    engine.PageConstructed += new
 OcrPageConstructionEventHandler(engine_PageConstructed);
    _pageNumber = 0;
    try
    {
        engine.Initialize();
        using (FileStream stm = new FileStream(outputPath, FileMode.Create))
        {
            engine.Translate(source, "application/pdf", stm);
        }
    }
    finally
    {
        engine.ShutDown();
    }
}
```

Combine PDF documents

To combine PDF documents, you can use the PdfDocument class. There is a static method called Combine, which takes either an output Stream or an output path and any number of input paths or input Streams. The input PDF documents are combined in order to create one output PDF document. This is essentially a one line task:

```
PdfDocument.Combine("output.pdf", "MobyDick.pdf", "Pride and Prejudice.pdf",
 "Metamorphosis.pdf");
```

You can also use a stream as the output:

```
using (Stream outputStream = GetStream())
{
    PdfDocument.Combine(outputStream, "A Room With a View.pdf", "Howard's End.pdf", "A
 Passage to India.pdf");
}
```

Finally, you can pass in streams for the output as well as the input. In this case, it is your responsibility to close the Stream objects when you're done.

ℹ If at least one of the documents being merged is PDF 2.0, the resulting document will be version 2.0.

Encrypt a PDF document

The `PdfDocument` class can be used to encrypt existing PDF documents. To do this, you create a PdfDocument object from your existing file and save with a password to a new file.

```
PdfDocument doc = new PdfDocument("plaintext.pdf");
PdfSaveOptions options = new PdfSaveOptions();
options.SetOneTimePasswords(password);
doc.Save(output, options, null);
```

There are no restrictions on password length or content.

To create password-protected PDF documents with version 1.7 and lower, the RC4 encryption algorithm is used.

When saving a password-protected PDF 2.0 documents, AES-256 encryption algorithm is used automatically.

Decrypt a PDF document

The PDF Document class can be used to decrypt existing PDF documents. To do this, you create a PdfDocument object from your existing file and password and save to a new file without a password

```
try
{
    PdfDocument doc = new PdfDocument("encrypted.pdf", password);
    doc.Save("cleartext.pdf");
}
catch (CodecException e)
{
    Console.WriteLine("Unable to open document: " + e.Message);
}
```

.TIF files

Atalasoft DotImage provides the ability to manipulate TIFF IFD's, Tags, and Pages using the TiffFile, TiffDirectory, and TiffDirectoryCollection classes located in the Atalasoft.Imaging.Codec.Tiff namespace. This section introduces these classes and provides an overview of the structure of a TIFF document.

A TIFF file, represented by the TiffFile class in Atalasoft DotImage consists of a header, which identifies the data as a TIFF and points to the first Image File Directory (IFD), represented by a

TiffDirectory in Atalasoft DotImage. Every TIFF has at least one IFD, which contains TIFF Tags and usually image data. Certain TIFF Tags are required for readers to read the encoded image data, such as ImageWidth and ImageLength. Other TIFF Tags can be stored within an IFD, including arbitrary Tags that can store data in any of the available TIFF Tag data types.

A TIFF Tag can contain a single scalar value, or an array of scalars. Tags must be one of the following available data types listed in the table.

Data Type	Size
Byte = 1	8-bit unsigned integer
Ascii = 2	8-bit bytes w/ last byte null
Short = 3	16-bit unsigned integer
Long = 4	32-bit unsigned integer
Rational = 5	64-bit unsigned fraction
SByte = 6	8-bit signed integer
Undefined = 7	8-bit untyped data
SShort = 8	16-bit signed integer
SLong = 9	32-bit signed integer
SRational = 10	64-bit signed fraction
Float = 11	32-bit IEEE floating point
Double =12	64-bit IEEE floating point
Ifd = 13	32-bit unsigned integer (offset)

A TIFF File contains at least one image encoded to one of the supported compression formats. The TiffFile class does not actually handle the compression or decompression of image data. It only handles the TIFF Tags and supports reordering, inserting, removing, and adding existing TIFF images into a new TIFF File.

TiffFile is also used to read all TIFF Tags from a file, and writes new TIFF Tags to a new file, including EXIF data.

TiffFile basics

The TiffFile class can be used to manipulate images and Tags in a multipaged TIFF. This section provides code examples that demonstrate how to perform these actions.

Read a TIFF file

The steps for reading a TIFF file are as follows:

1. Create an instance of the TiffFile class.
2. Invoke the Read() method to read the tags and IFD's from an existing TIFF document.
3. As the Read() method requires a Stream, open a Stream must be open for the entire time TIFF File is being used.
4. Close the Stream after the image is saved, or after all Tags have been accessed.

Edit a TIFF file

Once a TIFF File has been read, the TiffDirectory objects can be modified, removed, inserted from another TiffFile by accessing the Images property, and it's Tags can be edited.

A new image can be added or inserted from an existing AtalaImage object by using the appropriate constructor in the TiffDirectory class. In this case, the image is compressed using a TiffEncoder, then extracted into a new TiffDirectory just as if the image was saved as a separate file, then opened in it's own TiffFile class.

Save a TIFF file

After a TIFF File has been read, you can save it by invoking the Save() method. This saves current state of the TiffFile object to a new file or stream, including any modifications to the Tags and the collection of TiffDirectories.

The filename or stream of the saved TIFF File must be different than that of the source.

When adding or editing TIFF Tags and saving to a file, ensure that the tags being edited are not critical to decoding the TIFF. If they are, there is a chance that the resulting file will be corrupt. For example, editing the Compression tag to a value other than what the underlying compressed data is causes all readers, including Atalasoft DotImage to yield a corrupted image. Knowledge of the TIFF Specification is needed if you edit TIFF Tags.

Catch errors and warnings

TiffFile is designed to parse as much as it can from TIFF images, even if there are errors with the file. Instead of throwing exceptions when a problem is encountered while reading a particular tag, the CodecError event is raised. Handle the CodecError event in order to view any warnings associated with reading the TIFF directories.

Multipage TIFF file

A multipage tiff file consists of data structures called image file directories (IFD). Each IFD holds information of an individual page of a TIFF image. As a singly-linked list, insertion/deletion/rearrangement of IFDs is easy to do.

Managed code

Written in managed code, TiffFile provides functionalities to control IFDs of given tiff files. TiffFile regards a multipage TIFF file as a collection of IFDs, and an IFD as a collection of directory entires (DEs). Using TiffFile, clients can organize (insert/delete/swap pages) multipage TIFF and manage (insert/delete arbitrary tiff tags) an individual TIFF page.

TIFF file interfaces

The TiffFile interfaces consist of TiffFile, TiffDirectoryCollection, TiffDirectory.

TiffFile contains TiffDirectoryCollection obtained through extracting IFDs from a tiff image. Use the methods of TiffDirectoryCollection to control multi-TIFF images. The methods include swap and anything provided by CollectionBase.

Designed for manipulation of TIFF Tags, TiffDirectory contains a TiffTagCollection obtained automatically by extracting IFD.

TIFF tools

Atalasoft DotImage provides several different tools for creating or manipulating TIFF images. Each tool is designed for specific a purpose, but there is also a certain amount of overlap in functionality. The following table lists each of the tools and describes their features. With this information, you can select the appropriate tool for your job.

Tool	Description
TiffDecoder	Reads images from TIFF files. Provides read-only access to TIFF tags within a file.
TiffEncoder	Creates new TIFFs or appends images to existing TIFFs. Add TIFF tags as images are added.
TiffFile, TiffDirectory, TiffDirectoryEntry	Reads and writes entire TIFF files allowing full access to tags and images. Replace or insert images and tags in any order. Must save to a new file. Can read images from files, but TiffDecoder is more efficient.
TiffDocument, TiffPage	Simple, highly efficient set of objects for editing pages in a TIFF file. Easily insert pages from multiple files. Trivially combine multiple files into a single file. Does not provide access to TIFF tags.

TiffDocument and TiffPage

Atalasoft DotImage includes two classes for manipulating TIFF files:

- The TiffDocument class represents a high-level model of the pages within a TIFF file.

  When a TiffDocument object is constructed from an existing TIFF file, it contains a collection of TiffPage objects for each page within the file. TiffDocument attempts to be as efficient as possible in retrieving information from the files and does not ever load the actual images into memory. As such, it is possible to reorder and remove pages from files of arbitrary length. In addition, you can mix pages from other TiffDocument objects as easily as working with a document's own pages. Finally, TiffDocument contains a set of static methods that can be used for combining existing files, making the task of merging TIFF files into a simple one-liner.
- TiffPage is a high-level model of a page within a TIFF file. It contains a number of read-only properties that reflect the elements of each page within the file, including size, pixel format, resolution and frame number. TiffPage objects do not provide the means to edit these details. This is considered very advanced and is best handled by using the TiffFile, TiffDirectory, and TiffDirectoryEntry objects.

In addition to being constructed from the contents of existing TIFF files, it is possible to create new TiffPage objects from AtalaImage objects.

When making a TiffPage from an AtalaImage, the TiffPage contains a compressed copy of the AtalaImage.

For the reason just stated, the code that follows is perfectly valid.

C#

```
public TiffPage FromImage(AtalaImage image)
{
      TiffPage page = new TiffPage(image);
      image.Dispose(); // image memory no longer needed
 return page;
}
```

Be aware that creating a TiffPage from an AtalaImage consumes memory. The intent when creating TiffPage objects from AtalaImage objects is to make it easy to add a few pages to a document from in-memory. If you need to create dozens of TiffPage objects, it is more memory efficient to save the images into temporary files first using the TiffEncoder.

Work with TiffDocuments

Use TiffDocument to combine files. Either one of the code examples that follow combines several TIFF documents into a single file.

```
TiffDocument.Combine(destFile, sourceFiles); // sourceFiles is an array
```

or

```
TiffDocument.Combine(destFile, sourceFile1, sourceFile2, … sourceFileN);
```

This takes all of the source files and combines them in order into the destination file.

sourceFiles and destFile can be either streams or path names. destFile cannot have the same name as one of the source files as this damages the TIFF files.

Suppose you have job of adding a cover sheet onto a set of existing TIFF files. Using TiffDocument, you can write the following method:

```
public void AddTPSCoverSheet(AtalaImage coverSheet, string sourceFile, string destFile)
{
      TiffPage coverPage = new TiffPage(coverSheet);
      TiffDocument doc = new TiffDocument(sourceFile);
      doc.Pages.Insert(0, coverPage);
      doc.Save(destFile);
}
```

AddTPSCoverSheet injects a new TiffPage into a TiffDocument and saves it into a new file.

Determine if the pages of TIFF document are 1-bit black and white

Write the following method to determine if all of the pages of a TiffDocument are 1-bit black and white.

```
public bool IsBlackAndWhite(string sourceFile)
{
      using (FileStream stm = new FileStream(sourceFile, FileMode.Open,
 FileAccess.Read))
            {
            TiffDocument doc = new TiffDocument(stm);
```

```
            foreach (TiffPage page in doc.Pages)
            {
                  if (page.PixelFormat != PixelFormat.Pixel1bppIndexed)
                        return false;
            }
            return true;
      }
}
```

The stream is closed after all the work with a document is complete. Closing a stream that is contained in an active TiffDocument is an error. Using TiffDocument is more efficient than calling TiffDecoder.GetImageInfo() multiple times.

To intersperse the pages of two documents, use the following method:

```
public void Intersperse(Stream stm1, Stream stm2, Stream output)
{
      TiffDocument final = new TiffDocument(stm1);
      TiffDocument mixin = new TiffDocument(stm2);
      int index = 1;
      foreach (TiffPage page in mixin.Pages) {
            if (index < final.Pages.Count) {
                  final.Pages.Insert(page);
            }
            else {
                  final.Pages.Add(page);
            }
            index += 2;
      }
      final.Save(output);
}
```

Read TIFF tags

To read a TIFF tag from an image, use the static GetSingleTag from the TiffDecoder. Standard TIFF Tag ID's are stored in the TiffTagID enumeration.

The following example demonstrates how to obtain WANG annotation data from a TIFF image using GetSingleTag.

C#

```
TiffTag tag = TiffDecoder.GetTiffTag((int)TiffTagID.WangAnnotations,
   "myimagewithwang.tif", 0);
byte[] data = (byte[])tag.Data;
```

Write TIFF tags

To write TIFF Tags with an image, set the TiffTags property of the TiffEncoder class to a collection of TIFF Tags with the TiffTagCollection.

This example demonstrates how to save WANG annotation data with the image.

C#

```
TiffTagCollection tags = new TiffTagCollection();
 tags.Add(new TiffTag(TiffTagID.WangAnnotations, data);
 TiffEncoder tiffEncoder = new TiffEncoder();
 tiffEncoder.TiffTags = tags;
```

```
 myImage.Save("myimagewithwang.tif", tiffEncoder, null);
```

Save an image to a multipage TIFF file

Use these samples to save an image into an existing multipage tiff file.

C#

```
FileStream saveFile = new FileStream(@"C:\existingTiff.tif", FileMode.Open);
TiffFile theFile = new TiffFile(saveFile);

// Remove the old Image
theFile.Images.Remove(5);

// Add new Image from viewer as a new TiffDirectory
theFile.Images.Insert(5,new TiffDirectory(WS_Viewer.Image));

// Save to a temp file.
theFile.Save(@"C:\temp.tif");
saveFile.Close();

// Copy, and delete temp file.
File.Copy(@"C:\temp.tif", @"C:\existingTiff.tif", true);
File.Delete(@"C:\temp.tif");
```

Join two TIFF streams

Use these samples to save an image into an existing multipage tiff file.

C#

```
// open two TiffFiles using either stream or string
TiffFile oneTiff = new TiffFile(streamOne);
TiffFile twoTiff = new TiffFile(streamTwo)

// add two files
oneTiff.Image.Add(twoTiff.Image);

// save
oneTiff.Save(targetStream);
```

Add an AtalaImage to a TIFF stream

C#

```
// open a TiffFile and an AtalaImage
TiffFile oneTiff = new TiffFile(streamOne);
TiffDirectory twoTiff = new TiffDirectory(AtalaImage)

// add two files
oneTiff.Image.Add(twoTiff.Image[0]);

// save
oneTiff.Save(targetStream);
```

Remove pages from a multipage TIFF

C#

```
// open multipage TiffFile using either stream or string
TiffFile oneTiff = new TiffFile(streamOne);

// remove a page
oneTiff.Image.RemoveAt(1);

// save
oneTiff.Save(targetStream);
```

Merge two TIFF files

This code sample combines all images in two multipage TIFF files into one file.

C#

```
// open two TiffFiles using either stream or string
TiffFile oneTiff = new TiffFile(streamOne);
TiffFile twoTiff = new TiffFile(streamTwo)

// add two files
oneTiff.Image.Add(twoTiff.Image);

// save
oneTiff.Save(targetStream);
```

Swap pages in a TIFF file

C#

```
TiffFile file = new TiffFile();

using (Stream fs1 = new FileStream("file1.tif", FileMode.Open, FileAccess.Read,
    FileShare.Read))
{
        file.Read(fs1);
        file.Images.Swap(0, 1);
        file.Save("swappedtiff.tif");
}
```

Add arbitrary TIFF tags

C#

```
// open a TiffFile
TiffFile oneTiff = new TiffFile(streamOne);

// access to a TiffDirectory
TiffDirectory image = oneTiff.images[0];

// access to TiffTagCollection
TiffTagCollection tags = image.Tags;

// add Tiff Tags (add copyright)
Tags.Add(33432,”Atalasoft”);

// add Tiff Tags
string datetime = “2005:07:26 09:31:23\0”;
Tags.Add(306,datetime,TiffTagDataType.Ascii);

// save
```

```
oneTiff.Save(targetStream);
```

Delete a TIFF tag

C#

```
// open a TiffFile
TiffFile oneTiff = new TiffFile(streamOne);

// access to a TiffDirectory
TiffDirectory image = oneTiff.images[0];

// access to TiffTagCollection
TiffTagCollection tags = image.Tags;

// remove Tiff Tags (remove copy-right)
Tags.Remove(33432,”Atalasoft”);

// save
oneTiff.Save(targetStream);
```

View all tags in a TIFF file

```
TiffFile file = new TiffFile();
using (Stream fs1 = new FileStream("file1.tif", FileMode.Open, FileAccess.Read,
    FileShare.Read))
{
        file.Read(fs1);
        foreach (TiffDirectory image in file.Images)
        {
                foreach (TiffTag tag in image.Tags)
                {
                        Console.WriteLine(tag.ToString());
                }
        }
}
```

Add a TIFF tag

The following example adds new TIFF tags to an existing image.

C#

```
TiffFile file = new TiffFile();
using (Stream fs1 = new FileStream("file1.tif", FileMode.Open, FileAccess.Read,
    FileShare.Read))
{
        file.Read(fs1);
        TiffTagCollection tags = file.Images[0].Tags;
        tags.Add(new TiffTag(TiffTagID.Copyright, "(c) Atalasoft, Inc."));
        tags.Add(new TiffTag(306, "2005:07:26 09:31:23", TiffTagDataType.Ascii));
        file.Save("newtags.tif");
}
```

Add and retrieve binary TIFF tag

The following example adds and retrieves a binary TIFF tag.

This code embeds an image inside an arbitrary tag, and retrieves that image back.

C#

```
AtalaImage img1;
AtalaImage img2;
TiffFile file = new TiffFile();
using (Stream fs1 = new FileStream("file1.tif", FileMode.Open, FileAccess.Read,
   FileShare.Read))
{
   file.Read(fs1);
   img1 = new AtalaImage("file2.tif");
   TiffTag tag = new TiffTag(65535, img1.ToByteArray(new TiffEncoder()),
       TiffTagDataType.Byte);
   file.Images[0].Tags.Add(tag);
   file.Save("embeddedimagetag.tif");
}
//verify
TiffFile file2 = new TiffFile();
using (Stream fs1 = new FileStream("embeddedimagetag.tif", FileMode.Open,
 FileAccess.Read,
   FileShare.Read))
{
   file2.Read(fs1);
   TiffTag tag = file2.Images[0].Tags.LookupTag(65535);
   MemoryStream ms = new MemoryStream((byte[])tag.Data);
   img2 = new AtalaImage(ms);
}
```

Get All TIFF tags in image

The TiffFile class can be used to obtain all TIFF Tags in an image.

This code displays all of the TIFF tags in a TIFF document.

C#

```
int count = 1;
 TiffFile file = new TiffFile("multi.tif");
foreach (TiffDirectory image in file.Images)
 {
    Console.WriteLine("Tags in page: " + count);
    foreach (TiffTag tag in image.Tags)
    {
        Console.WriteLine(tag.ToString());
    }
    count++;
    Console.WriteLine();
 }
```

Convert between TIFF and JPEG

After parsing EXIF data, EXIF Tags and the thumbnail can be added, removed, and edited.

To save the edited ExifCollection back into a JPEG image, use the ToByteArray in the ExifCollection to create a new APP1 JpegMarker. Then set the AppMarkers in the JpegEncoder prior to saving the image, or prior to using CopyJpegWithNewMarkers. To save EXIF information into a TIFF image, use the TiffFile class and set the TiffDirectory object's ExifTags property to the ExifCollection containing the EXIF data.

With the EXIF editing capabilities of Atalasoft DotImage, it is possible to convert an image from JPEG to TIFF, and retain the EXIF information. Likewise it is possible to convert from TIFF to JPEG, also retaining the EXIF data. The following example demonstrate both techniques.

Convert from JPEG to TIFF with EXIF

C#

```
ExifParser exifparse = new ExifParser();
 ExifCollection exifTags = exifparse.ParseFromImage(filename);
 TiffFile tFile = new TiffFile();
 tFile.Images.Add(new TiffDirectory(new AtalaImage(filename),
 TiffCompression.JpegCompression));
 tFile.Images[0].ExifTags = exifTags;
 tFile.Save("convertedTiffWithExif.tif");
```

Convert from TIFF to JPEG with EXIF

```
string filename = "exif.tif";
 ExifParser exifparse = new ExifParser();
 ExifCollection exifTags = exifparse.ParseFromImage(filename);
 JpegMarkerCollection markers = new JpegMarkerCollection();
 markers.Add(new JpegMarker(JpegMarkerTypes.MarkerApp1, exifTags.ToByteArray()));
 JpegEncoder jpg = new JpegEncoder();
 jpg.AppMarkers = markers;
 AtalaImage image = new AtalaImage(filename);
 image.Save("convertedJpegWithExif.jpg", jpg, null);
```

Work with multipage TIFFs

Many document imaging applications require the use of multipage TIFF images. Atalasoft DotImage Document Imaging is designed to handle this with the MultiFramedImageDecoder (inherited by the TiffDecoder), the IAppendable interface (implemented by the TiffEncoder), and the ImageSource class which is designed to handle multi-framed images.

For manipulating TIFF pages (inserting, removing) and tags directly without re-encoding or decoding image data, see the TiffFile overview.

Open a multipage TIFF image

There are two ways to open a multipage TIFF image. To load each image in the TIFF at the same time for easy navigation and manipulation, use the Open() method in the Workspace or WorkpaceViewer object without specifying a frame index. This method sets the Images property with an ImageCollection containing each image in the file. We no longer recommend this. However, if the TIFF contains many pages, it is best to open only one image at a time, as it opens faster and conserves memory. This can be done manually, but we recommend using the FileSystemImageSource object to loop over each frame in the file.

Read each page in the TIFF

The following example demonstrates how to load each image into the Images property, and then loop through the ImageCollection to access information from each image.

C#

```
//load each image into the ImageCollection
myWorkspace.Open(filename);
```

```
//display information from each image
foreach (AtalaImage image in myWorkspace.Images)
    Console.WriteLine(image.ToString());
```

Read each page in the TIFF using a FileSystemImageSource

The following example demonstrates how to loop over each page in an image using the FileSystemImageSource object.

```
FileSystemImageSource source = new FileSystemImageSource(new string[] { filename } );
while (source.HasMoreImages()) {
    AtalaImage image = source.AcquireNext();
    Console.WriteLine(image.ToString());
    source.Release(image);
}
```

Read a single page in the TIFF

You can open a specific page from a multipage TIFF file by specifying the frame index. The following example loads only the first page by specifying a frameindex of 0.

C#

```
//load a specified page in the image
int frameindex = pagenumber - 1;
myWorkspace.Open(filename, frameindex);
```

Display and manipulate images in the ImageSource

To display a particular image in the FileSystemImageSource, or to apply an ImageCommand to an image other than the first image in the collection, you need to release the old image, if any, and then acquire the new image and set it to the current image. At that point, operations accessed through the Workspace object will act on that image. For coding convenience, the Workspace.Image property is a pointer to Workspace.Images.Current.

The following example demonstrates how to change the current image to the second image in the image source.

Change the current image to the second in the image source

C#

```
// select the next image
if (myWorkspace.Images.Current != null)
    source.Release(myWorkspace.Images.Current);
```

ℹ The FileSystemImageSource object can be treated as an array of images. This is somewhat confusing because the act of getting an element of the array does an implicit Acquire(). The Acquire() must be paired with a matching Release().

Replace the current image

The Workspace.Image property has two purposes:

- To get the current image in the ImageCollection (same as Workspace.Images.Current),
- To replace the current AtalaImage with a new image (same as Workspace.Images.Replace).

Add an image to the ImageCollection

If you open an image with the Open() method, the image is automatically added to the ImageCollection. However, there are situations where you may want to assemble a multiple page file. You can add an existing AtalaImage into the existing ImageCollection with the Add() or Insert() method of the ImageCollection class.

The following example shows how to load a series of single page images into one ImageCollection

C#

```
for(int i = 1; i < 4; i++)
{
    //opens the images image1.tif, image2.tif, and image3.tif
    //and adds each image to the tiffAssembler workspace
    myWorkspace.Images.Add(new AtalaImage("c:\\image" + i.ToString() + ".tif"));
}
```

In addition to adding an image to the collection, you can also insert an image into a specified position in the ImageCollection, or replace an existing image with another image.

Save or append a multiple page TIFF image

To save an existing ImageCollection as a multiple page TIFF, invoke the Workspace.Save() method and specify the TIFF encoder image type. You do not need to specify a frameindex to save.

The following example demonstrates how to save the current ImageCollection as a multiple page TIFF file with G4 FAX compression.

C#

```
for (int i = 1; i < 4; i++)
{
    //opens the images image1.tif, image2.tif, and image3.tif
    //and adds each image to the tiffAssembler workspace
    myWorkspace.Open("c:\\image" + i.ToString() + ".tif");
    myWorkspace.Save("c:\\multiimage.tif", new TiffEncoder(
        TiffCompression.Group4FaxEncoding, true));
}
```

Save an image specifying compression type

The following example shows how to save an image as a TIFF, specifying the compression type.

C#

```
myWorkspace.Save("myImage.tif", new TiffEncoder(TiffCompression.Group4FaxEncoding));
```

DICOM

Digital Imaging and Communications in Medicine (DICOM) is the standard format used to manage medical imaging information and its related workflow. Developed in 1993, the DICOM standard consists of a file format definition and a network communications protocol.

DICOM is rapidly becoming the standard for all electronic health record systems that include imaging information as part of patient records. Adherence to the DICOM standard allows DICOM compliant devices from multiple manufacturers to work together in a seamless fashion as every DICOM complaint device must specify the DICOM classes it supports. Software developers who are DICOM conformant ensure that every medical imaging facility can use their software and that their tools can integrate with any electronic health records system.

DICOM was developed the the DICOM Standards Committee and is managed by the the Association of Electical and Medical Imaging Manufacturers. For more information, go to http://DICOM.nema.org/.

The DicomDecoder is an ImageDecoder that decodes DICOM images into an AtalaImage. The basic information you need to create a DotImage Decoder for DICOM images is provided here.

- Assembly `Atalasoft.dotImage.Dicom.dll`
- Namespace Atalasoft.Imaging.Codec.Dicom

Use the DicomHeaderParser to read metadata from a DICOM image.

Some classes allow manipulation of DICOM datasets and images closer to the raw formats provided by the file format. The classes DicomDataset and DicomImage allow more efficient repeated dynamic adjustments to be made to images without having to repeatedly decode them.

Extra classes for handling DICOM images

In addition to the DicomDecoder object, Atalasoft DotImage includes classes for more efficient manipulation of DICOM Images. In particular, it is often important to be able to manipulate the brightness and contrast of a Dicom image and display the changes in an active user interface. While this can be done by decoding a DICOM image into an AtalaImage object and repeatedly applying the BrightnessContrastCommand, it can be important to perform these transformations in the native image space that is produced by a specific imaging device. In DICOM parlance, this is called applying window and leveling.

Three native DICOM image spaces are available to clients of Atalasoft DotImage:

- Raw: Format that represents the raw samples that are contained within the file.
- Modality Transformed: Format that represents the image transformed by a modality transform supplied by the imaging device.
- Presentation: Format that represents the image transformed into ranges that are suitable for presentation, usually via a window and leveling transform defined in the file.

DicomDataset

The DicomDecoder creates all three images to provide an AtalaImage. First the raw image is read, then it is transformed into the modality image, then into the presentation image, before being put into an AtalaImage.

In most cases, client code will use a modality transformed image then repeatedly apply a window and leveling to produce a series of AtalaImage objects. By operating this way, the original image will only be decoded once and the the modality transform will only be applied once.

The two main objects used for this process are DicomDataset and DicomImage. A DicomDataset object models the dataset structure from within a DICOM file. Constructed from a Stream object,

the DicomDataset object implements IDisposable. It is very important for client code to call the Dispose() method when it is done with a DicomDataset. If this is not done, an exception may be thrown when the application exits.

A DicomDataset is a factory for DicomImage objects. Client code calls one of the Get...Image methods to get a new DicomImage object of the desired type. Each Get...Image method requires an index to the desired frame. Note that multiple calls with the same frame index will return unique, newly-allocated images.

- GetRawImage(int frameIndex) - returns a new raw dicom image with minimal processing of the image data
- GetModalityTransformedImage(int frameIndex) - returns a new raw dicom with the device modality transform applied
- GetPresentationImage(int frameIndex) - returns a new raw dicom image with the modality transform and a window and leveling transform applied.

DicomImage

The DicomImage object is a model of the underlying raw data format. Using the DicomImage object, client code can transform it into a new or existing AtalaImage with an optional window and leveling transform. In addition, the DicomImage object has information about how the image may have been transformed.

For example, DICOM images are often represented by sample values that are signed rather than unsigned. Since most imaging systems use signed sample values, it is necessary to shift the signed values into an unsigned range. If an image was shifted out of the signed range, the property ImageCameFromSignedSamples will be true and the amount of this shifting is available via the ImageDataShiftedBy property.

DicomImage implements IDisposable and like the DicomDataset object, it is very important for client code to call the Dispose() method when it is done with a DicomImage.

Use DicomDataset and DicomImage

Using DicomDataset and DicomImage it is possible to implement the functionality of the DicomDecoder method:

Implement DicomDecoder.Read

```
public AtalaImage Read(Stream stm, int frameIndex)
 {

    using (DicomDataset dataset = new DicomDataset(stm)) {

        if (frameIndex < 0 || frameIndex >= dataset.FrameCount)

            throw new ArgumentOutOfRangeException("frameIndex");

        using (DicomImage image = dataset.GetPresentationImage(frameIndex)) {

            return image.GetAtalaImage();

        }

    }
}
```

This code creates a DicomDataset object, range checks the frameIndex, then retrieves the presentation DicomImage and translates it to an AtalaImage. Note the use of the "using" syntax to ensure that the DicomDataset and DicomImage objects are disposed.

A typical application will want to open a modality transformed image then repeatedly perform window and leveling operations on the image. Rather than allocate a new AtalaImage for each operation, it is desirable to apply the window and leveling operation directly into an existing image. An appropriate image can be constructed via the DicomImage method AllocateAtalaImage.

Apply window and leveling

The window value is typically stored in a signed sample range and must therefore be shifted by the same amount as the samples. If this is not done, then it is likely that the resulting image will be all white or all black.

```
private void OpenDicomImage(Stream stm)
{
    _dataSet = new DicomDataset(stm);
    _dImage = _dataSet.GetModalityTransformedImage(0);
    _atalaImage = _dImage.AllocateAtalaImage();
    _window = _dImage.DefaultWindow;
    _leveling = _dImage.DefaultLeveling;
    UpdateImage();
}
private void UpdateImage()
{
    int window = _window;
    if (_dImage.ImageCameFromSignedSamples)
        window += _dImage.ImageDataShiftedBy;
    _dImage.GetAtalaImage(_image, window, _leveling); // transforms into _image
}
```

JPEG2000 - encoding images

Encoding JP2 images involves creating an instance of Jp2Encoder, which derives from ImageEncoder, and then invoking the Save() method. The Standard edition, allows adjustment of just one property, the Compression property. The Professional edition grants low level access to the Codec. This covers all encoder settings.

Encode tiled images

Among the important features of JPEG2000 is capability of encoding a large image - up to $(2^{32} - 1) \times (2^{32} - 1)$ pixels - without breaking into tiles. JPEG2000 also has a facility of compressing an image

into tiles in which data may be compressed independently in each tile or in each color component. Encoding with (or without) tiles using Atalasoft DotImage Jpeg2000 is explained here.

To encode without tiles	Set the TileSize property to of Jp2Encoder to (0,0) or Size.Empty which implies no tiles.
To encode with tiles	Set the TileSize property of Jp2Encoder to any non-zero size.

Encode region of interest

The JPEG2000 specification includes the ability to encode user specific areas of the image at higher quality. This gives more detail to certain areas of an image, without compromising file size. The Jp2Encoder has an EnhancedRegions property, which points to a collection of Jp2EnhancedRegion objects. Setting the EnhancementFactor in the Jp2EnhancedRegions collection boosts the quality of all regions by a factor over the Compression property.

Baseline encoder properties

The properties listed in the table below can be set for the entire image only (cannot be independently set to tiles or components).

Compression

Gets or sets the compressed size of the image as a percentage of an uncompressed image.

EnhancedRegions

Gets a collection of Enhanced Regions that can be set at a higher quality level than the rest of the image during compression.

FileFormat

Gets or sets the file format to generate (JP2, JPEG2000 codestream, or JPX).

IPData

Gets or sets intellectual property rights data to be encoded with the image.

IptcTags

Gets or sets IPTC data that will be stored in the encoded image.

Precision

Gets or sets a value indicating the precision of the wavelet coefficients.

QualityStyle

Gets or sets a value indicating the quality mode during lossy compression.

SpeedMode

Gets or sets the speed mode (Fast or Accurate) to use during lossy compression.

TileSize

Gets or sets the size in pixels of each individual tile.

UuidBoxes

Gets or sets UuidBox metadata to be stored in the encoded image.

UuidInfoBoxes
Gets or sets UuidInfoBox metadata to be stored in the encoded image.

WriteTileLengthMarker
Gets or sets a value indicating if a tile length marker is written to the encoded image.

XmlBoxes
Gets or sets XML metadata to be stored with the image.

Per tile encoder properties

These properties can be set for the entire image or independently for each tile. These properties exist in EncoderOptions, and by default are applied to the entire image unless overriding GetEncoderOptions.

QualityLayers
Gets or sets the number of quality layers in the code stream for use with progressive decoding.

ProgressionOrder
Gets or sets the organization of the coded data.

PacketMarkers
Gets or sets a value that creates special markers at the beginning and/or at the end of each block of a coded area.

Per tile and per component properties

These properties can be set for the entire image, independently for each tile, and independently for each component. These properties exist in EncoderOptions, and by default are applied to the entire image unless overriding GetEncoderOptions.

WaveletFilterMethod
Gets or sets a value selecting reversible (WaveletFiveThree) or irreversible (WaveletNineFive) wavelet filters.

WaveletLevels
Gets or sets the number of wavelet transformation levels.

QuantizationStyle
Gets or sets the quantization steps.

CodeBlockSize
Gets or sets the size of the blocks of coded data.

CoderOptions
Gets or sets the coder options for faster compression / decompression.

Getting Started with JPEG2000

This section gives you information to get started with JPEG2000.

Register the JPEG2000 codec in Atalasoft DotImage

To set up Atalasoft DotImage to decode JP2 images, add an instance of the Jp2Decoder to the Atalasoft.Imaging.Codec.RegisteredDecoders.Decoders collection.

Use the JPEG2000 imaging codec

C#

```
...
Jp2Decoder jp2 = new Jp2Decoder();
RegisteredDecoders.Decoders.Add(jp2);
```

Link to the license file

To compile Atalasoft DotImage JPEG2000 in a Windows Forms application such that the royalty free license is installed into the application resource, a file called `licenses.licx` must be added to the project, with the following line in that file:

```
Atalasoft.Imaging.Codec.Jpeg2000.Jp2Decoder, Atalasoft.dotImage.Jpeg2000
```

This instructs the Visual Studio .NET compiler to embed the license file to the resources in the exe. For information about compiling and embedding the license, see Generating licenses.

Decode JPEG2000 images

Once the decoder is registered, all methods that decode an image, such as Workspace.Open or new AtalaImage(filename) recognizes JP2 images as valid supported images. The Read() method of Jp2Decoder can also be called directly to bypass the image format check determining the codec to use.

Get information from JPEG2000 images

It is possible to retrieve information from a JPEG2000 image without decoding it by using the GetImageInfo() method of the Jp2Decoder class, or the GetImageInfo() method of the RegisteredDecoders class. Information, including width, height, bitdepth, is available. To access Intellectual Property Rights data and the type of code stream, cast the returned ImageInfo class to a Jp2ImageInfo.

Get image information

To encode JPEG2000 images, create an instance of the Jp2Encoder class, and pass it into the Workspace.Save() method, the AtalaImage.Save() method, or the Save() method in the Jp2Encoder class. The Compression property in the Jp2Encoder can be set to compress the resulting image to the desired amount. For example, setting the compression to 5 results in an image that is approximately five per cent of the size of the original uncompressed image.

C#

```
using Atalasoft.Imaging.Codec.Jpeg2000;
...
Jp2ImageInfo info = (Jp2ImageInfo)RegisteredDecoders.GetImageInfo("myimage.jp2");
```

```
Console.WriteLine(info.FileFormat);
```

Deploy Atalasoft DotImage JPEG2000

When deploying Atalasoft DotImage JPEG2000 to a client machine or server, the following must be copied to the same folder as the exe which references the Atalasoft DotImage assemblies:

- `Atalasoft.dotImage.dll`
- `Atalasoft.dotImage.Jpeg2000.dll`
- `Atalasoft.dotImage.Lib.dll`

When distributing client desktop applications, the license file is embedded into the resource, and there is no need to distribute or activate any additional licenses.

When installing on a production server, a server license must be acquired and activated for the server.

Introduction to JBIG2

Atalasoft DotImage JBIG2 codec can be used to decode and encode JBIG2 images using the Microsoft .NET Framework. JBIG2 compression is an open standard and can compress bi-tonal images 2 - 5 times more than the same image compressed with the industry standard TIFF CCIT Group4 compression. The codec is available as a plug-in that integrates with Atalasoft DotImage seamlessly. Atalasoft DotImage JBIG2 is based off of Luratech's Lurawave.jb2 compression technology. Licensing is runtime royalty free for desktop applications.

The JBIG2 standard has been developed by the Joint Bi-level Experts Group (JBIG) for the efficient lossless and lossy compression of bilevel (black and white) images. It is capable of compressing black and white documents considerably more than the more commonly used CCIT Group 4 TIFF compression.

The use of symbol dictionaries and symbol matching in JBIG2 enables very effective encoding of documents containing recurring symbols, making JBIG2 ideal for compressing documents. JBIG2 has been made popular by Adobe PDF, which incorporates JBIG2 in the PDF 1.5 specification. PDF Reader includes JBIG2 support for reading these types of PDF documents.

Product Features

- Ability to decode any page from a 1-bit JBIG2 image.
- Encode a single or multipage document as a JBIG2 image, an image only PDF document.
- Supports Lossless or Lossy compression
- Supports encoding or decoding from any stream
- Read a specified region from an existing JBIG2 image stream.
- Generate image only PDF documents with embedded JBIG2 images.
- Integrated with the Atalasoft DotImage PDFEncoder to encode PDF images with other compression formats.
- Runtime Royalty Free Desktop Licensing

JBIG2 Compression

Jb2Encoder, derived from MultiFramedImageEncoder, can compress a bi-level image (i.e. PixelFormat.Pixel1bppIndexed) to JBIG2 or PDF.

ℹ An exception is thrown if the source image is not 1-bit image. To avoid the exception, convert the image to 1-bit prior to compression.

The JBIG2 Codec can compress images as JB2 format, or PDF with embedded JBIG2 images, or a JBIG2 stream that can be directly embedded inside a PDF. See the ExportFormat property for specifying the export format.

The EncodingMode property indicates if an image is compressed Lossy or Lossless.

JBIG2 Decompression

The Jb2Decoder. derived from MultiFramedImageDecoder, can access an existing JBIG2 file or JBIG2 embedded stream. To decode a stream, see the Read() method.

The decoder supports decoding a single frame, specifying the frame index, or directly to an ImageCollection. To register the JBIG2 decoder in the list of known codecs, add an instance of the Jb2Decoder into the Atalasoft.Imaging.Codec.RegisteredDecoders.Codecs collection.

Compression Examples

	Uncompressed	CCIT Group IV	JBIG2 Lossless	JBIG2 Lossy	PDF w/ Lossless JBIG2
3-page FAX, 1728 x 2293	1,454 Kb	121 Kb	24 Kb	23 Kb	26 Kb
strip chart grid scan, 9760x6976	8,321 Kb	1,088 Kb	675 Kb	674 Kb	676 Kb
Mechanical CAD drawing, 9259x6816	7,716 Kb	209 Kb	104 Kb	100 Kb	105 Kb

Custom codecs - image codec

An image codec is a program that can encode and decode an image.

Atalasoft DotImage can read and write most common image formats. Images are read with ImageDecoders and written with ImageEncoders. The extensibility model of Atalasoft DotImage allows custom encoders and decoders to be created easily by Atalasoft DotImage or third parties. Plug-ins for Jpeg2000 and other codecs are available separately.

Supported formats

Format	ImageDecoder	ImageEncoder	Assembly location
Jpeg	JpegDecoder	JpegEncoder	Atalasoft.dotImage

Png	PngDecoder	PngEncoder	Atalasoft.dotImage
Bmp	BmpDecoder	BmpEncoder	Atalasoft.dotImage
Tiff	TiffDecoder	TiffEncoder	Atalasoft.dotImage
Gif	GifDecoder	GifEncoder	Atalasoft.dotImage
Pcx	PcxDecoder	PcxEncoder	Atalasoft.dotImage
Tga (Targa)	TgaDecoder	TgaEncoder	Atalasoft.dotImage
Psd	PsdDecoder	PsdEncoder	Atalasoft.dotImage
Wbmp	WbmpDecoder	WbmpEncoder	Atalasoft.dotImage
Emf	EmfDecoder	EmfEncoder	Atalasoft.dotImage
Wmf	WmfDecoder	WmfEncoder	Atalasoft.dotImage
Tla	TlaDecoder	TlaEncoder	Atalasoft.dotImage
Pcd	PcdDecoder	-	Atalasoft.dotImage
Pnm	PnmDecoder	PnmEncoder	Atalasoft.dotImage
Pdf†	-	PdfEncoder	Atalasoft.dotImage.Pdf
Pdf**	PdfDecoder	-	Atalasoft.dotImage.PdfReader
Raw*	RawDecoder	-	Atalasoft.dotImage.Raw
JPEG 2000**	Jp2Decoder	Jp2Encoder	Atalasoft.dotImage.Jpeg2000
JBIG2**	Jb2Decoder	Jb2Encoder	Atalasoft.dotImage.Jbig2
DWG**	DwgDecoder	-	Atalasoft.dotImage.Dwg
DXF**	DwgDecoder	-	Atalasoft.dotImage.Dwg
DICOM**	DicomDecoder	-	Atalasoft.dotImage.Dicom
XPS	XpsDecoder	-	Atalasoft.dotImage.Wpf
JpegXR	JpegXrDecoder	JpegXrEncoder	Atalasoft.dotImage.Wpf
Heif	HeifDecoder	-	Atalasoft.DotImage.Heif

* Included with Atalasoft DotImage Photo Pro and Atalasoft DotImage Document Imaging

**Available as a separate plug-in module

† Included with Atalasoft DotImage Document Imaging

Create a decoder

The Decoder class reads image data and converts it into an AtalaImage object. This class inherits from the Atalasoft.Imaging.Codec.ImageDecoder object.

Required methods

The following methods are required to qualify as a compatible Atalasoft DotImage plug-in.

C#

```
public override ImageInfo GetImageInfo(Stream stream)
```

This method is used to gather information about this image, including ImageType, width, height, bit depth, PixelFormat and DPI. You can also create your own custom ImageInfo object if the image format contains additional information. See Create Custom Image Information.

C#

```
public override bool IsValidFormat(Stream stream)
```

Atalasoft DotImage uses this method to find out if the image being opened can be read by your plug-in. The method returns *true* if it can be read.

C#

```
public override AtalaImage Read(Stream stream, ProgressEventHandler progress)
```

This method is called to ask the plug-in to read the image from a stream and convert it into an AtalaImage object. If possible, you should also raise the progress event for each scan line. If you are unable to read the image in the stream, return *null.*

In addition to the preceding methods, you must also include the following property.

C#

```
public override ImageType SupportedImageType
{
 get { return ImageType.Unknown; }
}
```

This property indicates what image type your plug-in can read. If the image type is unknown to Atalasoft DotImage, you should return ImageType.Unknown.

Additional methods and properties

If the image format your decoder reads can make use of additional methods and/or properties, go ahead and add them. While Atalasoft DotImage does not use them, you can use your class to access those features directly.

AtalaImage data format

It is very important that you know how the image data should be formatted in order for Atalasoft DotImage to display and manipulate it properly.

Internally, the image data is stored as a top-down, 32-bit aligned buffer. This makes it very likely that you will have to add padding to the end of each row. In addition, the colors of 24- and 32-bit images are stored in BGR and BGRA order.

Palettes

If the image format you are reading requires a palette, you will have to add the palette entries to the AtalaImage.Palette object. You can do this by using the Palette.SetEntry() method.

If the image is grayscale, a standard grayscale palette is automatically be added when you create the AtalaImage object. This palette is ordered from black (0) to white (255).

If the image is 1-bit, a black and white palette is automatically created in which white is 0 and black is 1. If the image requires different colors, you can change them with the SetEntry() method.

ImageData

The AtalaImage.ImageData property is a pointer to the beginning of the image data. You can set the image data using Marshal.Copy, or you can use unsafe code to set the image data directly to the pointer.

There is also an AtalaImage constructor which lets you to set the value of the ImageData property. You may prefer this if you want to control the memory allocation.

Create an encoder

The image encoder is used to save an AtalaImage object into a specific image format. Your encoder must inherit the Atalasoft.Imaging.Codec.ImageEncoder class.

There is only one required method for the encoder as shown in the example.

Required method for image encoder

This method is used to save the AtalaImage into the specified stream. If possible, you should raise the progress event for each scan line. The return value is the number of bytes written to the stream.

C#

```
public override int Save(Stream stream, AtalaImage image, ProgressEventHandler
 progress);
```

Additional methods and properties

You are free to add any additional methods and properties to your encoder. If you review the source code for PortableImageEncoder, you see that Atalasoft DotImage has added multiple constructors and three properties that provide easy access to the features of this image format.

AtalaImage data format

Image data is stored as a top-down, 32-bit aligned buffer. You must be sure to watch for row padding in the image data, and remember that 24- and 32-bit images are stored in BGR and BGRA order.

Extend a codec

Atalasoft DotImage was designed with extensibility in mind. Advanced .NET programmers can extend or override existing functionality. Many methods are marked as virtual, meaning that they can be overridden when the class is inherited. This allows native functionality to be overridden and enhanced.

The Atalasoft DotImage ImageCommands and Codecs provide good examples of how a developer can extend and customize the Atalasoft DotImage application. By inheriting the ImageCommands and Codecs base classes, you can add your own plug-ins to the product.

For example, you can create an ImageCommand that uses your own special algorithm. Alternatively, you can add an ImageDecoder for your favorite, but unsupported and obscure, image format. You have the option to sell your plug-ins commercially.

To create a custom decoder or encoder, you must inherit the abstract base class %ImageDecoder:Atalasoft.DotImage~Atalasoft.Imaging.Codec.ImageDecoder% or %ImageEncoder:Atalasoft.DotImage~Atalasoft.Imaging.Codec.ImageEncoder% respectively. The next section explains how to create a custom decoder and encoder.

Decoder tutorial

One of the great features of Atalasoft DotImage is the ability to create your own plug-in to support extra file types. You can even create your own image format and take advantage of the image processing features of Atalasoft DotImage.

This tutorial covers they key concepts you need to create your own plug-in. The code examples are taken from the PortableImage component. The tutorial includes the following:

- The Decoder
  - Requirements
  - Additional Methods and Properties
  - The AtalaImage Data Format
  - Palettes
  - Image Data
  - Example Code
- Custom ImageInfo
  - The Constructor
- The Encoder
  - Requirements
  - Additional Methods and Properties
  - The AtalaImage Data Format
  - Example Code

Adjust decoder properties

Some Image Decoders have properties that can be set to change the decoding behavior. For example, the JpegDecoder can be set to read at scaled down levels.

The following example shows how to change the image loading scale for a JPEG image.

Reset decoder properties

This example demonstrates how to change the properties of the default JpegDecoder so that the image is loaded at one-half scale.

C#

```
JpegDecoder jpeg =
 (JpegDecoder)RegisteredDecoders.GetDecoderFromType(typeof(JpegDecoder));
jpeg.ScaleFactor = JpegScaleFactor.Half;
```

Read a PPM image

The following example shows how to write code to read a PPM Image into an AtalaImage object.

Reset decoder properties

The following code is used to read a PPM (24-bit) image into an AtalaImage object.

The method in the following example loops through each pixel, reads the three bytes which make up the RGB data and sets the memory values to this data in BGR order. At the end of each row, it moves to the beginning of the next row by advancing the *pointer* by *offset* amount.

To calculate the offset (padding), take the number of bytes required for one row (width * 3 in this case) and subtract it from the number of bytes in a 32-bit aligned row (image.RowStride).

C#

```
private AtalaImage _Read24BitBinary(Stream stream, int width, int height, int
 maxColor,
    ProgressEventHandler progress)
{
    AtalaImage image = new AtalaImage(width, height, PixelFormat.Pixel24bppBgr);
    ProgressEventArgs e = new ProgressEventArgs(0, height,
        System.Reflection.MethodInfo.GetCurrentMethod().Name);

    unsafe
    {
        int offset = (image.RowStride - (width * 3));
        byte[] bytes = new byte[3];
        byte* pointer = (byte*)image.ImageData;

        // Read each line into the image buffer with padding.
        for (int y = 0; y < height; y++)
        {
            // Raise the progress event.
          if (progress != null)
            {
                e.Current = y;
                progress(this, e);
            }

            for (int x = 0; x < width; x++)
            {
                stream.Read(bytes, 0, 3);
                Utilities.GetReadColorValue(ref bytes, this.maxColor);

                // The color is stored in BGR order.
                pointer[0] = bytes[2];
                pointer[1] = bytes[1];
                pointer[2] = bytes[0];

                // Move to the next pixel.
                pointer += 3;
            }

            // Advance to the next row.
            pointer += offset;
        }
    }
```

```
    // Send final progress event.
    if (progress != null)
    {
        e.Current = height;
        progress(this, e);
    }

    return image;
}
```

Save an AtalaImage to a PPM file

The following is the code used to save an AtalaImage object into a PPM (24-bit) file. Take a look at the PortableImagePlugin source code for 8 and 1-bit saving.

C#

```
private int _Save24Bit(Stream stream, AtalaImage image, ProgressEventHandler
progress)
{
    // Save the header information.
    int count = _WriteTheHeader(stream, image);

    int height = image.Height;
    int widthBytes = image.Width * 3;
    byte[] bytes = new byte[widthBytes];
    int rs = image.RowStride;
    int padding = rs - widthBytes;
    byte[] ascii;
    IntPtr p = image.ImageData;

    ProgressEventArgs progressArgs = new ProgressEventArgs();

    for (int y = 0; y < height; y++)
    {

        // Raise the progress event.
        if (progress != null)
        {
            progressArgs.Current = y;
            progress(this, progressArgs);
        }

        // Read one line of the image.
        Marshal.Copy(new IntPtr(p.ToInt32() + y * rs), bytes, 0, widthBytes);

        // Remember, internally the image is a DIB in BGR format.
        Utilities.SwapRedAndBlue(ref bytes, widthBytes);

        if (this.binaryEncoding)
        {
            // Adjust the color values if needed.
           if (this.maxColor != 255)
                Utilities.GetSavedColorValue(ref bytes, this.maxColor);

            stream.Write(bytes, 0, widthBytes);
            count += widthBytes;
        }
        else
        {
            // Convert the data into its ascii version and do the color correction.
```

```
            ascii = Utilities.BinaryToAscii(bytes, this.maxColor);
            stream.Write(ascii, 0, ascii.Length);
            count += ascii.Length;
        }

    }

    // Raise the progress event.
    if (progress != null)
    {
        progressArgs.Current = height;
        progress(this, progressArgs);
    }

    return count;
}
```

Create custom image information

When creating a decoder for an image, there may be times when the image format contains unique information beyond what the ImageInfo object provides. Create a class that inherits ImageInfo to allow users access to this extra information.

The image formats supported by the PortableImagePlugin contain a parameter called MaxColor, which specifies the maximum value of a single color component in the image data. The ImageInfo object does not contain this property, so the custom class, PortableImageInfo provides that information upon return from the GetImageInfo() method in the decoder.

This allows Atalasoft DotImage to use the known properties of ImageInfo and also allows users of the to get the MaxColor value from your plug-in.

C#

```
public class PortableImageInfo : ImageInfo
```

The constructor

The constructor of your custom class must also create the ImageInfo object to set the properties. As these properties are read-only, use the following syntax to accomplish this.

C#

```
public PortableImageInfo(Size size, int colorDepth, PixelFormat pixelFormat, int
 maxColor)
    : base(size, colorDepth, ImageType.Unknown, new Dpi(0, 0,
 ResolutionUnit.Undefined),
    pixelFormat)
```

This code creates the base (ImageInfo) constructor, using the parameters of the PortableImageInfo constructor. Because the image formats supported by this plug-in always return ImageType.Unknown, we removed this parameter from our constructor, and placed it in the base constructor.

ℹ The addition of the maxColor parameter in our constructor. The parameter sets the value returned my the MaxColor property of the custom ImageInfo class.

Barcode reading

Atalasoft DotImage BarcodeReader was designed to be very easy to use. An application needs just a few lines of code to read all supported barcodes located within an image.

Before working with barcodes, you may also wish to review the Barcode Reader Demo supplied with the Atalasoft DotImage Toolkit. It is a working Windows Forms application that demonstrates the abilities of the component. Full source code is supplied. For more information on this demo, see Atalasoft DotImage Demos.

As of Atalasoft DotImage 7.0b, there is a new Barcode Reader. It was designed to be as compatible as possible with the old Barcode Reader, but there are a few differences:

1. The new object for reading is call BarCodeReader.
2. BarCodeReader implements the IDisposable interface and should be disposed as soon as you are done with it. We recommend putting BarCodeReadings in a C# using block.
3. BarCodeReader is highly optimized for 1-bit images. To make this easier, there are two new constructors:
   - BarCodeReader(AtalaImage image, bool autoThreshold) - if autoThreshold is true, this constructor will convert images that are not 1-bit to 1-bit. This process is done with the DynamicThresholdCommand if it is licensed or with the ThresholdCommand otherwise.
   - BarCodeReader(AtalaImage image, ImageCommand thresholdCommand) - if the image is not 1-bit, thresholdCommand will be applied to the image to convert it. If the processed image is not 1-bit, it will be disposed and ignored.
4. BarCodeReader now includes the property SupportedSymbologies which returns a Symbologies enumeration that includes all bar codesymbologies recognized and licensed for use by the BarCodeReader. In addition, there is the method IsSymbologySupported(), which returns true if a passed in Symbologies enumeration is recognized and licensed by the BarCodeReader.
5. The new BarCodeReader no longer supports the MicroQr, Planet, AustraliaPost, Code11, IntelligentMail, Itf14, Rss14, RssLimited, Telepen or Rm4scc symbologies.
6. The XOptions property of the ReadOpts class is no longer used and has been marked obsolete
7. The Directions enumeration now includes diagonal compass points (Northeast, Southeast, etc.). Directions.Northeast is not equivalent to Directions.East | Directions.North.
8. The ScanInterval property of the ReadOpts class is not currently used.
9. ReadOpts now includes a property called ReadingQuality, which controls the balance of speed vs. accuracy in the BarCodeReader. The default value, ReadingQuality.MostAccurate, reads with the greatest accuracy. ReadingQuality.Fastest reads as fast as possible, but with less accuracy.
10. The Symbologies enumeration now includes convenience values All1D, All2D, and All, which represent all one-dimensional symbologies, all two-dimensional symbologies, and all symbologies, respectively.
11. The BarCode object now includes a property, Data, which contains the data for a bar codeas an array of bytes.

Upgrade tips

In most cases, all that needs to be done is to reference the new assembly Atalasoft.dotImage.BarCoding and change the name of the Barcode Reader class from BarReader to BarCodeReader. It is strongly recommended that you dispose BarCodeReader objects as soon as possible after you are done with them. This can be done in a C# using block. In addition, we recommend that you read 1-bit images. You can convert them 1-bit yourself, or construct the BarCodeReader with autoThreshold set to true:

```
using (BarCodeReader reader = new BarCodeReader(image, true))

{

    BarCode[] bars = reader.ReadBars(myReadOpts);

    ProcessBars(bars);

}
```

Use the BarcodeReader

The BarcodeReader was designed to be very easy to use. An application needs just a few lines of code to read all supported barcodes located within an image.

The following examples demonstrate how to read barcodes from an Atalasoft.Imaging.AtalaImage object.

The steps involved in reading a bar codeare as follows:

1. Create an instance of BarCodeReader by passing in an AtalaImage object.
2. Create an instance of the ReadOpts class and set the symbology(s) and directions you wish to read.
3. Invoke the ReadBars() method in the BarCodeReader class. This returns an array of Barcode instances. Each element of the array corresponds to a bar coderead from the image:

Reading a barcode

You can use a single BarCodeReader instance to read the same image a number of times, each time with different options as shown in the example that follows.

C#

```
// 1: Load the image containing barcodes
AtalaImage myImage = new AtalaImage("barcodes.tif");
// 2: Create BarCodeReader for specified image.
using (BarCodeReader br = new BarCodeReader(myImage))
{
    // 3: Create a ReadOptions.
    ReadOpts options = new ReadOpts();
    // 4: Read left to right.
    options.Direction = Directions.East;
    // 5: Symbology to read.
    options.Symbology = Symbologies.code128;
    // 6: Read the barcodes contained in the image.
```

```
    Barcode[] bars = br.ReadBars(options);
    // 7: Process the results.
 for (int i = 0; i < bars.Length; i++)
       System.Console.WriteLine(bars[i].ToString());
}
```

Read a bar code with options set

C#

```
// 1: Load the image containing barcodes
AtalaImage myImage = new AtalaImage("barcodes.tif");
// 2: Create BarCodeReader for specified image.
using (BarCodeReader br = new BarCodeReader (myImage))
{
    // 3: Create a ReadOptions.    ReadOpts options = new ReadOpts();

    // 4: Read left to right.
    options.Direction = Directions.East;
    // 5: Symbology to read.
    options.Symbology = Symbologies.code128;
    // 6: Read the barcodes contained in the image.
    Barcode[] bars = br.ReadBars(options);

    if (bars.Length == 0)
    {
       // No barcodes read. Maybe the image was scanned upside down. Let's see
       options.Direction = Directions.West;
       bars = br.ReadBars(options);
    }
}
```

Render a bar code into an AtalaImage

Even though the Barcode Writing assembly has no direct Atalasoft DotImage dependencies, it's easy to write barcodes into an AtalaImage. The process is simple, first create a 24-bit color AtalaImage, then use the GetGraphics method to get a Graphics object, then use the bar codewriters Render method.

This process is shown in the following code sample.

C# - Code 39 Barcode Writing

```
AtalaImage CreateBarcodeImage(string text, int width, int height)
{
    AtalaImage image = new AtalaImage(width, height, PixelFormat.Pixel24bppBgr,
 Color.White);
    BarcodeWriter bc = new BarcodeWriter(BarcodeStyle.Code39);
    Graphics g = image.GetGraphics();
    bc.Render(text, g, new Rectangle(0, 0, width, height));
    return image;
}
```

Verify a bar code can represent a string

Since some 1-D barcodes are limited in what they can represent, it is useful to be able to ask the BarcodeWriter if it can correctly render a given string. This is done by calling the BarcodeWriter's

Validate method. Validate takes a string and a boolean and will return true if the string can be rendered in the BarcodeWriter's current BarcodeStyle. If the boolean passed in is true, the BarcodeWriter will throw an ArgumentException if given an invalid string. If the boolean passed in is false, the Validate will return false if given an invalid string.

This process is shown in the following code sample.

C# - Validating Code 39

```
public void CheckCode39String(string s)
{
    BarcodeWriter bc = new BarcodeWriter(BarcodeStyle.Code39);
    if (!bc.Validate(s, false))
    {
        Console.WriteLine("Invalid bar codestring: " + s);
    }
    else
    {
        Console.WriteLine("Barcode string accepted.");
    }
}
```

Barcode writing

The DotImage Barcode Writing assembly is designed to be a simple set of classes that make it easy to create barcodes within a .NET application. There are objects that are designed for writing directly into Graphics objects and corresponding Win Forms Controls that allow barcodes to appear in window-based applications.

The Barcode Writing classes can be used for many common 1-D bar codetypes as well as PDF417 and DataMatrix barcodes.

Deployment

When using Barcode Writing, the assemblies that need to be copied with your application include:

Assembly	Description
`Atalasoft.Shared.dll`	Shared classes such as licensing management
`Atalasoft.dotImage.Barcoding.Reading.dll`	Barcode Writing Engine

Use the Barcode writer

The simplest way to use the Atalasoft DotImage Barcode Writer is to drag a bar codecontrol onto a window form. There are three main controls:

1. BarcodeControl - for all 1-D barcodes such as Code 39, Codabar, UPCA etc.
2. DataMatrixBarcodeControl - for DataMatrix 2-D barcodes
3. Pdf417BarcodeControl - for PDF417 2-D barcodes.

Once placed on a form, you can change the size and placement in the designer and adjust specific settings in the Properties window.

Each control has a property called Text which is used to set the text encoded in the barcode. Note that many 1-D barcodes have restrictions as to what characters can be encoded in the bar code- for example, many 1-D barcodes only represent numbers and letters.

The bar codewriters can also be used to write into a Graphics object directly. For each bar codecontrol, there is a corresponding class that will write the barcode:

1. BarcodeWriter - writes all 1-D barcodes.
2. Pdf417BarcodeWriter - writes PDF417 2-D barcodes.
3. DataMatrixBarcodeWriter - writes DataMatrix 2-D barcodes.

Each of these classes contains a method called Render which draws the requested barcode. All bar codewriters have a flavor of Render that handles strings. Pdf417BarcodeWriter and DataMatrixBarcodeWriter have methods for handling raw data in the form of a byte array.

Writing a raw data PDF417 barcode

```
public void WriteRawPDF417Data(Graphics g, Rectangle bounds, byte[] data)
{
    Pdf417BarcodeWriter writer = new Pdf417BarcodeWriter();
    writer.Render(data, g, bounds);
}
```

OCR document design considerations

Since some 1-D barcodes are limited in what they can represent, it is useful to be able to ask the BarcodeWriter if it can correctly render a given string. This is done by calling the BarcodeWriter's Validate method. Validate takes a string and a boolean and will return true if the string can be rendered in the BarcodeWriter's current BarcodeStyle. If the boolean passed in is true, the BarcodeWriter will throw an ArgumentException if given an invalid string. If the boolean passed in is false, the Validate will return false if given an invalid string.

The design of the OcrDocument hierarchy reflects the conflicting needs of OCR engines and OCR clients.

It is highly likely that a client of an OcrEngine will modify the OcrDocument and OcrPage classes. For example, a client may add keywords to the OcrDocument object or thumbnail images to the pages. Therefore, concrete engine implementations should never construct an OcrDocument or OcrPage class directly. It is the responsibility of the client to supply code to construct the objects within the hierarchy.

On the other hand, the engine is likely to supply implementations of OcrLine, OcrWord, and OcrGlyph that can be tightly coupled to data supplied by the engine. This means the engine needs to be able to make very specific versions of these classes.

To manage these conflicting goals, the Ocr namespace uses two patterns. The first is a factory used to construct document elements. Never access constructors for document elements directly. Use the engine's Factory property instead as shown in the example below.

Use the factory property to construct a document

```
OcrPage page;

//Do not do this!
page = new OcrPage(width, height, resolution);

//This is much better
page = engine.Factory.OcrPage(width, height, resolution);
```

The page element object represents an extensible interface to the element without dictating the implementation. The data accessor defines how that data is stored and retrieved.

For example, the implementation of the Baseline property in OcrLine does not use a member variable to store the baseline. Instead, it uses an object called an OcrLineAccessor to get the baseline. In this way, a client can supply definitions for OcrLine, OcrWord, and OcrGlyph.The engine supplies the accessors that define how the object's information is retrieved. A client could then override OcrLine to include extra information about a line of text without affecting how any given engine is required to provide information that meets the core definition of an OcrLine.

Furthermore, once an OcrDocument has been constructed, its elements can be edited, augmented, deleted, or merged without affecting any of the existing components. For example, OcrPage objects taken from an OcrDocument recognized by one engine can be inserted into an OcrDocument that was produced by another engine, or words can be changed (spell corrected, moved, replaced) without affecting any of the other elements, *even if they are still tied to a specific engine.*

To get an OcrDocument or an OcrPage from an image, use the OcrEngine's Recognize() method. There are versions that operate on single images and ImageSource objects.

Load OCR resources

In order to operate, each of our OCR engines require some external resources. The particular type and content of those resources as well as how they are structured vary greatly from one engine to another. Thankfully, we have taken care to streamline the management of these resources for you. There are only two potential issues you need to keep in mind when developing an OCR product with Atalasoft DotImage: Initialization and Deployment.

Initializing resources

Most supported OCR Engines are external programs you can interface with. These programs may need to be run or otherwise loaded into memory before they can be used. To simplify this we have provided specialized loaders which take care of this for you.

GlypthReaderLoader

C#

```
static Form1()
{
  //Preload the GlyphReader resources from a default location.
  GlyphReaderLoader loader = new GlyphReaderLoader();
}
```

To make it easy to jump in and start developing an OCR application, our packaged OCR engines know where your OCR resources are installed and will access them automatically. For similarly simple deployment, you can copy the `OcrResources` directory located within the `bin` directory of your Atalasoft DotImage installation directly into the directory which holds your application's assemblies. All of our engine's wrappers are designed with this case in mind and should require no further configuration. However, if you wish to place these resources elsewhere, some care must be taken to ensure they are loaded properly. Exactly what must be done can vary from engine to engine.

GlyphReader

All of the resources required by GlyphReader are contained within it's own executable. Therefore, there is no need to worry about them once the GlyphReaderLoader has been called. However, you will need to ensure that the loader can find the executable, the configuration file, and the dll found within the GlyphReader subdirectory of OcrResources.

GlyphReaderLoader will look in the following locations for its resources, in order:

If the resourceDir constructor parameter is supplied:

1. The "OcrResources\GlyphReader\v5.0" subdirectory of the specified directory is checked.
2. The "GlyphReader\v5.0" subdirectory of the specified directory is checked

If the resourceDir constructor parameter is not supplied:

1. The `bin\OcrResources` subdirectory of your Atalasoft DotImage installation directory is checked.
2. The `OcrResources\GlyphReader\v5.0` subdirectory of the directory where the GlyphReader Assembly is located is checked.
3. The `GlyphReader\v5.0` subdirectory of the directory where the GlyphReader Assembly is located is checked.

To place the resources in a different directory you will need to specify that location inside the GlyphReaderLoader constructor.

C#

```
static Form1()
{
  //Preload the GlyphReader resources from a subfolder of the
  //global ApplicationData directory.
 string appdata =
 Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData);
  string subfolder = @"GlyphReader Resources";
  string resourcePath = Path.Combine(appdata, subfolder);
  GlyphReaderLoader loader = new GlyphReaderLoader(resourcePath);
}
```

Tesseract

Tesseract has a great number of external files it needs in order to operate. The vast majority of these files are the datasets for each of the languages Tesseract supports. All of these files are contained within the Tesseract subdirectory in OcrResources.

Tesseract's resources are loaded at construction time. First, a directory list is built. Second, each directory in that list is checked to see if it contains the "Tesseract\v3.04\tessdata" directory structure. When a directory that matches is found, that directory is checked to ensure that it contains all of the correct resource files. If any of the resources are missing an OcrException is thrown.

Each of the following is checked for the "Tesseract\v3.04\tessdata" subpath in order:

1. If the constructor's resources argument is supplied, it is checked.
2. If the constructor's resources argument is supplied, the "OcrResources" subdirectory of the argument is checked.
3. The location of the Tesseract Assembly.
4. The location of the Tesseract Assembly's "OcrResources" subdirectory.
5. The Atalasoft DotImage installation's "bin\OcrResources" directory is checked.

Finally, if all of the previous checks fail, the TESSDATA_PREFIX environment variable is checked. TESSDATA_PREFIX is used internally by Tesseract to determine where it's resources are located. This environment variable is checked for the "tessdata" subdirectory only.

To place the resources in a different directory you will need to specify that location inside of the TesseractEngine constructor:

C#

```
private OcrEngine MakeTesseractEngine()
{
  //Load the Tesseract resources from a subfolder of the
  //global ApplicationData directory.
 string appdata =
 Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData);
  string subfolder = @"Tesseract Resources";
  string resourcePath = Path.Combine(appdata, subfolder);
  Tesseract3Engine engine = new Tesseract3Engine(resourcePath);
  return engine;
}
```

OmniPageLoader

C#

```
static Form1()
{
  //Preload the OmniPage resources from the directory to which they
  // were downloaded into.
  OmniPageLoader loader = new OmniPageLoader(@"c:\OmniPageEngineOcrResources ");
}
```

There is the second OmniPageLoader constructor without parameters that is used when resources are located in the predefined locations.

```
OmniPageLoader loader = new OmniPageLoader();
```

OmniPage

The resources required for the initialization of the OmniPage Engine are not included in the main Atalasoft DotImage installation. They must be downloaded separately, and a link and details are issued to the customers of this add-on. The resources are delivered in a single compressed file that must be decompressed into a folder for Atalasoft DotImage to read them.

The location of the resource folder, whether absolute or relative, can then be passed in as a parameter to the loader. This resource folder will have to be included into any distribution of your application.

Stages in OCR translation

Broadly speaking, the stages in OCR translation are:

- Preparing for the translation
- Doing the work of translation
- Cleaning up after the translation

Work of translation

The work of translation can, for most OCR engines, be broken down into several stages. Typically there are three stages as shown below.

- During preprocessing, an image is treated to make it more likely that it is correctly recognized.
- During location, an image is broken up into regions that contain specific content such as text, images, barcodes, and so forth.
- During recognition, these regions are analyzed and scanned for content.

Preprocessing

Image preprocessing options allow you to make changes to an image before it is processed by the OcrEngine. These options serve to clean up the image so that the results of optical character recognition are improved.

Engines that support built-in document preprocessing provide a number of options. The OcrPreprocessingOptions class maintains options settings as a 32-bit integer with each bit representing an option.

- To select an option, set the property corresponding to the option you want on to *true*.
- To deselect an option, set the property corresponding to the option you want off to *false*.

The pre-processing options are listed in the following table.

Option	**Description**
Deskew	Image is straightened
Autorotate	Image is rotated so that it is oriented correctly for reading
Despeckle	Noise in the image is stripped out

ToBilevel	Image is down-sampled to 1 bit per pixel
Invert	Image's sense of black and white is reversed
FlipLeftRight	Image is mirrored left/right

Not all engines support these options. Use the OcrEngine property AvailablePreprocessingOptions to determine if an option is supported by a particular engine.

Atalasoft DotImage performs all Deskew operations. This is done via the AutoDeskewCommand by default. This behavior can be overridden in two ways. In both cases, clients start by creating a subclass of the specific OcrEngine object. To change how the AutoDeskewCommand operates, override the factory method MakeDeskewCommand and return a new AutoDeskewCommand object with properties set to your needs. The second way is to override the virtual method HandleDeskew. HandleDeskew takes an input image and returns a deskewed version of the image as well as the angle detected, or null if there was no need to deskew.

All other operations are performed by the engine itself. The order of operations is entirely engine dependent.

A client could add any or all of these methods to an engine by attaching to the ImageSendOff event and performing the operations there.

Recognition / Translation

OcrEngine supports the events listed in the table below.

When a page is recognized, the following events are fired and are in this order listed in the table below.

Event	**When fired**	**Notes**
DocumentProgress	When document processing starts.	Indicates in broad terms what an engine is in the process of doing and how far along it is.
PageProgress	When page processing starts.	Indicates in broad terms what an engine is in the process of doing and how far along it is
ImageIntroduction	When image is first presented to the engine.	This is the first opportunity that a client has to manipulate the image.
ImageTransformation	When an image is processed in such a way that it might affect the overall page coordinate system.	This is where deskewing is done
PageConstructing	OcrPage object for this page has been constructed and its size fixed.	This is an opportunity to hang client specific data onto the page object to be accessed later (ie, metadata, page numbers, etc).
ImageSendOff	Image is in its final state before being sent to the engine.	This is the last opportunity to access or modify the image before it is otherwise changed by the engine (turned to black and white or processed)

PageLocation	Bounding region have been identified.	If the engine supports this capability, provides a set of bounding regions that represent areas identified for recognition. Engines that do not support this capability return a single text region encompassing the entire page. A client may choose to modify this set of regions to limit OCR to particular areas.
PageConstructed	After a page had been fully populated with all page elements.	A client may opt to spell-check or otherwise edit the page contents at this point.

The DocumentProgress and events can be used to cancel an OCR in progress.

If an engine has Native Translators, it may not fire all of these events during translation. In particular, PageConstructing and PageConstructed do not make sense in a NativeTranslator context.

Translator types

Broadly speaking, here are two different types of translators: native and foreign. Native translators are built into a particular engine, foreign are not.

Native translators

Native translators never create an OcrDocument object. It may be difficult or impossible to adjust their feature sets and if they do not provide an interface for streaming. Foreign translators are those that are supplied from outside of an engine. A foreign translator is likely to be flexible in its configuration and output styles, but can be less efficient, when used with some engines, since it operates at a significantly higher level than native translators.

Typically, native translation never incurs the generation of an OcrDocument and foreign translations always incur the generation of an OcrDocument. This distinction allows us to publish low-level engine's translators and to provide a means of adding other translators that are treated as first class objects.

Text translator

A TextTranslator class implements is a foreign translator that is used to generate text files from an OCR engine. TextTranslator properties are used primarily to control how much the TextTranslator attempts to mimic the layout and format of the original document.

PDF translator

The Atalasoft DotImage PdfTranslator class allows client applications to generate high quality PDF documents from scanned documents. PdfTranslator provides the following features:

- Ability to set PDF Metadata fields
- High quality thumbnail images

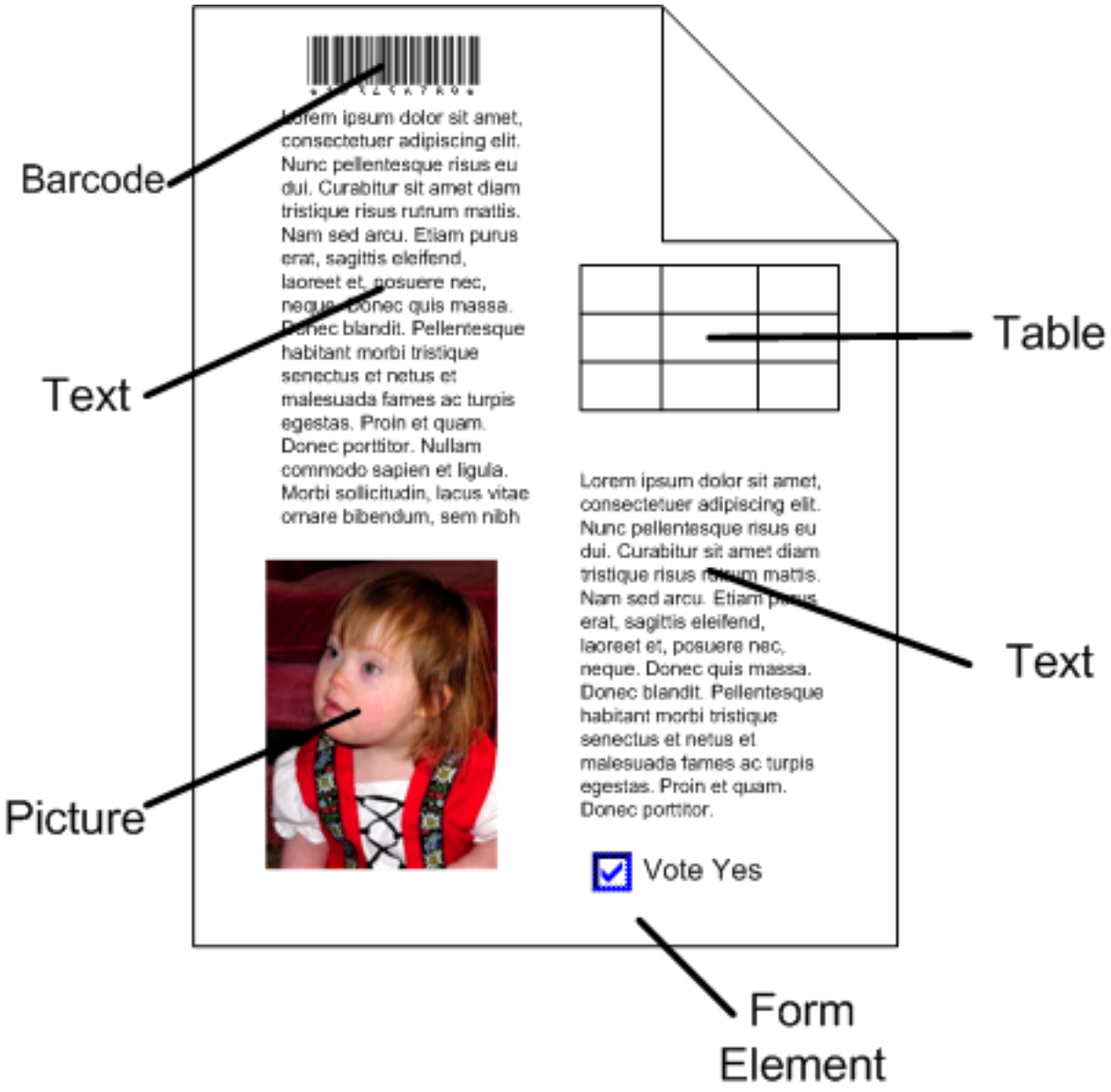
- Accurate text placement
- Text-Under-Image placement
- Optional placement of picture regions
- Automatic or client-controlled image compression
- Advanced codec support (JBIG2, JPEG 2000)
- Insertion of client synthesized pages
- Generation of PDF/A-1b and PDF/A-2b compliant documents with embedded fonts and color profiles

Page region types

Once the images that make up a page are located or recognized, an OcrPage is created. The OcrEngine also creates an OcrRegionCollection that contains OcrRegion objects for every located or recognized area on the page.

Atalasoft DotImage OCR defines the set of OcrRegion types listed in the following table.

Region type	**Area type**
OcrTextRegion	Contains text
OcrImageRegion	Contains an image
OcrBarcodeRegion	Contains a bar code
OcrTableRegion	Contains a table
OcrFormElementRegion	Contains form elements such as check boxes or bubbles
Picture	shown in illustration



Bounding boxes

Every OcrRegion has a property called Bounds which is the bounding box for that element on the page. The bounding box is always measured in pixels using the source image resolution for units.

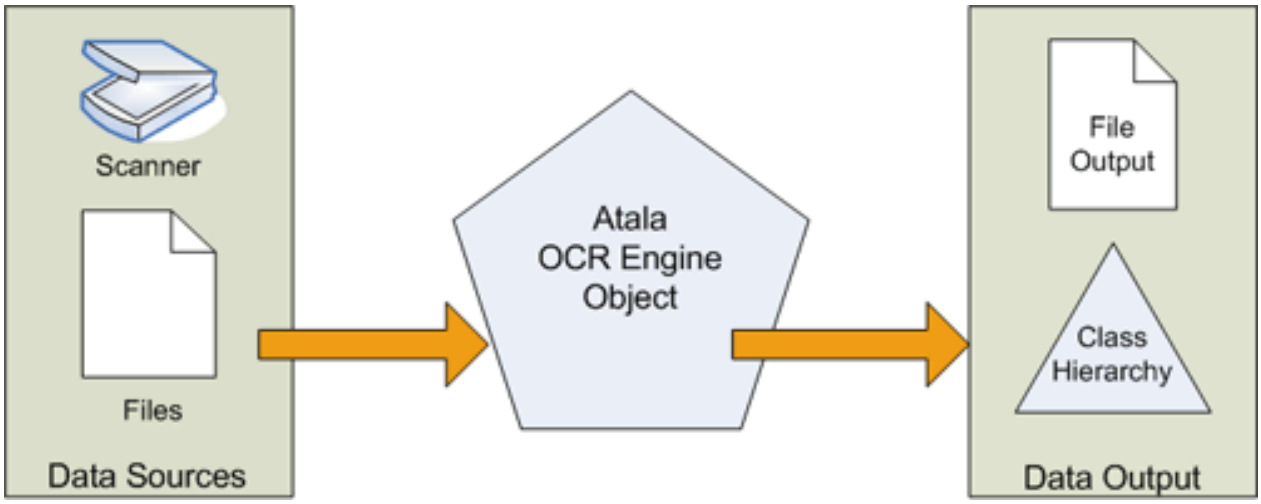
The GlyphReader engine only supports OcrTextRegion.

OCR engine

Atalasoft DotImage OCR is designed to easily interface with other aspects of your application. It is extensible with an event driven object-oriented object model. In just a few lines of code, a

developer can recognize an image and output that image to a file, or enumerate its lines, words, and characters with confidence.

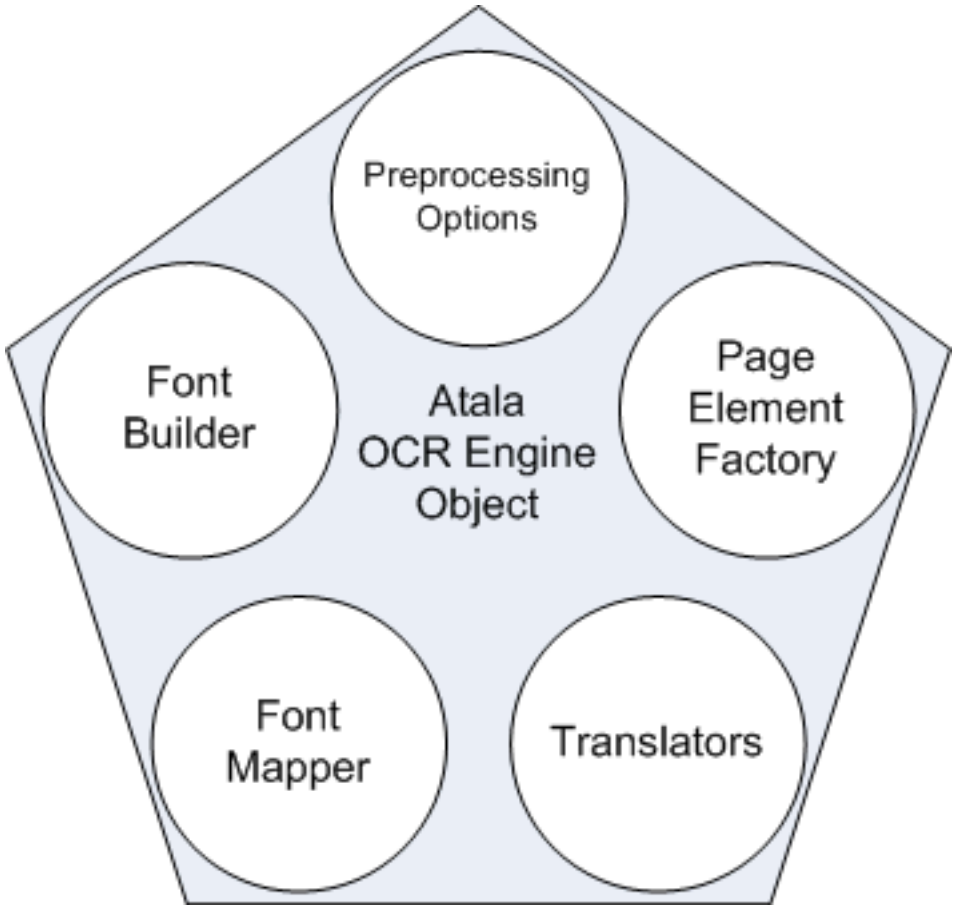
Data sources for the engine can be scanned images or files. The engine output consists of either a file or a class hierarchy. This model is illustrated below.



As the OcrEngine object is abstract, you cannot create an instance of this object. Nevertheless, the object definition contains most of the necessary functionality needed for a concrete subclass to function with a minimum of extra code.

The OcrEngine object has five primary components as illustrated below:

- Preprocessing options
- Document translators
- Page element factory
- Font mapping
- Font building



Glyph reader engine

The glyph reader engine is a highly accurate OCR engine built for Atalasoft DotImage. The engine has been tested with the ISRI OCR Performance Toolkit and has been found to be more accurate, with a 99.5% accuracy rate, than other expensive industry leading OCR engines.

GlyphReader is a lexicon OCR engine requiring no dictionary. It supports European characters only. The following ASCII characters are supported.

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
€						†	‡			Š		Œ		Ž							▪					š		œ		ž	Ÿ
	¡	¢	£	¤	¥		§		©		«			®	¯	°	±				µ	¶	·				»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

The GlyphReader engine *does not* support font name or family determination. This engine does support font size, baseline, glyph bounds, and confidence.

Features

GlyphReader supports the following features:

- European Character Set.
- Reports individual character position and size.
- Reports character confidence.
- OCR's of rotated pages, reports the rotation angle.
- Automatically breaks merged characters, or merges broken characters.
- Optionally rejects low confidence characters.
- Optionally reject low confidence lines.
- Disabling recognition of specific characters.
- Full Page color OCR can be generated when combined with the Searchable PDF Module.

Features that are found in some engines but not in GlyphReader include zoning, and determining font characteristics.

Output formats

As with any OCR engine using the Atalasoft DotImage OCR interface, all foreign translators are supported. Text translation is supported out of the box. Searchable PDF is available with the PDF Translator add-on. Therefore, the following mime types are supported for output:

- text/plain
- application/pdf (requires PDF Translator add-on)

Licensing

The Atalasoft DotImage OCR GlyphReader Engine is licensed per concurrent use. Two GlyphReader licenses are required for two applications to use GlyphReader simultaneously. If the application will only be residing on the server, you have the option of purchasing a server license granting an unlimited number of users connected to the server running the Atalasoft DotImage OCR GlyphReader Engine enabled application with up to 20 concurrent processes/threads running at once.

Deployment

- `Atalasoft.Shared.dll`
- `Atalasoft.dotImage.Lib.dll`
- `Atalasoft.dotImage.dll`
- `Atalasoft.dotImage.Ocr.dll`
- `Atalasoft.dotImage.GlyphReader.dll`

GlyphReader also requires the following unmanaged assemblies and support files, located in: `C:\Program Files\Atalasoft\DotImage 11.4\Bin\OcrResources\GlyphReader\v5.0`:

- TOCR50.qnp

- TOCR50.teh
- TOCR50de.gar
- TOCR50el.gar
- TOCR50en.gar
- TOCR50es.gar
- TOCR50fr.gar
- TOCR50it.gar
- TOCR50nl.gar
- TOCR50no.gar
- TOCR50ru.gar
- TOCR50sk.gar
- TOCR50tr.gar
- `x86`
  - GlyphReader.dll
  - GlyphReaderEngine.exe
  - GlyphReader.ini
- `x64`
  - GlyphReader.dll
  - GlyphReaderEngine.exe
  - GlyphReader.ini

Please leave the directory structure intact when deploying the engine. Code inside `Atalasoft.dotImage.GlyphReader.dll` will determine which `GlyphReaderEngine.exe` will be executed. If 64-bit dlls are used the 64-bit engine will be executed, if 32-bit dlls are used then the 32-bit engine will be executed.

These assemblies can be installed along side the managed assemblies only if the OcrResourceLoader or GlyphReaderLoader class is instantiated in a static constructor of the class that invokes GlyphReader.

By default the unmanaged assemblies are found in `<SDK folder>\OcrResources\GlyphReader\v5.0\`.

Due to the architecture of the GlyphReader engine, to specify a location other than a default search path you need to create an instance of the OcrResourceLoader or GlyphReaderLoader in a static constructor before any OCR code is loaded. This is the case even if the resources are in the assembly folder. There you can specify an alternate location of the resources if desired.

Tesseract engine

The Tesseract Engine, class name TesseractEngine, is an open source engine that we provide without charge to those who purchase the OCR Package. It is a commercial quality OCR engine originally developed at HP between 1985 and 1995. HP and UNLV open-sourced this engine in 2005.

Features

The Tesseract engine is fast and runtime royalty free although it is not quite as powerful as the other engines supported by Atalasoft DotImage. In particular, it lacks segmentation and it is not very good at recognizing low quality documents.

Supported languages

The TesseractEngine supports the following languages:

- Dutch
- English
- French
- German
- Italian
- Portuguese
- Spanish

Supported output formatters

The TesseractEngine supports the following output formatters and provides a structure that allows you to build your own.

- Text
- PDF

Deployment

- The assemblies listed below are required for deployment.
  - Atalasoft.dotImate.Ocr.Tesseract3
  - Atalasoft.dotImage
  - Atalasoft.dotImage.Ocr
  - Atalasoft.dotImage.Lib
  - System
  - System.Data
  - System.Drawing

Additionally, the Tesseract language files must be accessible. These are automatically placed in the Atalasoft DotImage directory during toolkit installation. When deploying, you must either copy the OcrResources to your application directory or tell the engine their location explicitly by passing it into the TesseractEngine constructor.

Please see the Tesseract3Engine class documentation for additional information.

The Tesseract Engine is used in exactly the same way as the other OCR engines, all of which inherit from the same base class, Atalasoft.dotImage.OCR.

Special considerations

Once the Tesseract Engine is used and recognize is called with a language, you cannot change to an alternate language. The initialization happens the first time a document is recognized. Attempting to change the language at any time beyond that point results in an exception being thrown.

OmniPage Engine

The OmniPage Capture SDK lets developers access a broad range of algorithms and workflows that can recognize machine printed text as well as hand printed text in a variety of languages. Atalasoft DotImage includes the OmniPage Engine as an optional, purchasable add-on.

Features

The OmniPage Engine supports nearly all of the standard functionality defined by the Atalasoft.dotImage.Ocr.OcrEngine base class. This includes recognition and translation, whether using foreign translators (TextTranslator and PdfTranslator), or the engine's own built-in translation functions.

In addition to the standard functionality, the OnmiPage engine supports some additional features.

ICR (Intelligent Character Recognition) is used for recognizing printed handwritten text. In order to provide support for ICR, OmniPageIcrTextRegion class is used to represent an area of hand-printed text on a page with optional parameters specific to the OmniPage OCR engine.

This additional Recognize method is added:

```
OcrPage Recognize(AtalaImage image, List<OmniPageOcrTextRegion> regionList)
```

This method takes an AtalaImage as a parameter and a List consisting of OmniPageOcrTextRegion objects which extends OcrTextRegion. There are several implementations which specifies certain regions for recognition:

- OmniPageOcrMicrTextRegion
- OmniPageOcrCMC7TextRegion
- OmniPageIcrTextRegion

This feature enables you to create a list consisting of a mix of these objects or a list consisting of just one type or the other type and pass them into a Recognition process along with the image (AtalaImage) they pertain to. An object of one of these types holds information on a location in the image (x and y coordinates for width and height). Optionally, the object can contain the orientation of the text marked by its relation to the page. This info is used by the OmniPage Engine for configuration of the recognition process. The type of object itself instructs the OmniPage Engine which operation to perform on the corresponding region in the image.

Construction of an OmniPageOcrTextRegion (also as any of the implementations) can be performed through one of these constructors:

- OmniPageOcrTextRegion (Rectangle bounds)
- OmniPageOcrTextRegion (PolygonF bounds)
- OmniPageOcrTextRegion (Rectangle bounds, OcrTextRotation rotation)

The bounds parameter referenced in all of these constructors designates the location of the region in the image. The OcrTextRotation rotation parameter designates the orientation of the text in relation to the top of the document. If this parameter is not given, the text referred to by this OmniPageOcrTextRegion is assumed to be at 0 degrees in relation to top.

An alternative way to set specific recognition regions is to create a custom OcrPageLocationEventHandler by the client and register it to the PageLocation event of the OmniPageEngine instance. It should be registered before Recognition or Translation is launched.

The handler should retrieve the collection (OcrRegionCollection) of recognized regions output by the engine from the RegionsIn property of the OcrPageLocationEventArgs object returned from the event or create a new collection if this property is null. After retrieving this collection, the client can add new regions, remove or change located regions.

The result regions collection should be assigned to the RegionsIn property of the OcrPageLocationEventArgs object as in the following example code:

```
engine.PageLocation += (sender, e) =>
{
  OcrRegionCollection regionsColl = e.RegionsIn ?? new OcrRegionCollection();
  var region1 = new OmniPageOcrTextRegion(new Rectangle(150, 150, 350, 187));
  var region2 = new OmniPageOcrTextRegion(new Rectangle(538, 216, 319, 67));
  regionsColl.Add(region1);
  regionsColl.Add(region2);
  e.RegionsOut = coll;
};
```

Supported languages

Use the GetSupportedRecognitionCultures method of the Atalasoft.dotImage.Ocr.OcrEngine base class to obtain a full list of supported languages.

The following table shows all supported languages:

English	Moldavian	Interlingua	Shona
German	Bulgarian	Kasub	Sioux
French	Byelorussian	Kawa	Sami
Dutch	Ukrainian	Kikuyu	Sami(Lule)
Norwegian	Russian	Kongo	Sami(Northern)
Swedish	Chechen	Kpelle	Sami(Southern)
Finnish	Kabardian	Kurdish	Somali
Danish	Afrikaans	Latin	Sotho
Icelandic	Aymara	Luba	Sundanese
Portuguese	Basque	Luxembourgish	Swahili
Spanish	Bemba	Malagasy	Swazi
Catalan	Blackfoot	Malay	Tagalog
Galician	Breton	Malinke	Tahitian
Italian	Brazilian	Maori	Pirez

Maltese	Bugotu	Mayan	Tongan
Greek	Chamorro	Miao	Visayan
Polish	Tswana(Chuana)	Minangkabau	Welsh
Czech	Corsican	Mohawk	Sorbian(Wend)
Slovak	Crow	Nahuatl	Wolof
Hungarian	Eskimo	Nyanja	Xhosa
Slovenian	Faroese	Occidental	Zapotec
Croatian	Fijian	Ojibway	Zulu
Romanian	Frisian	Papiamento	Japanese
Albanian	Friulian	PidginEnglish	Chinese(S)
Turkish	Gaelic(Irish)	Provencal	Chinese(T)
Estonian	Gaelic(Scottish)	Quechua	Korean
Latvian	Ganda(Luganda)	Rhaetic	Thai
Lithuanian	Guarani	Romany	Arabic
Esperanto	Hani	Rwanda	Hebrew
Serbian(Latin)	Hawaiian	Rundi	Vietnamese
Serbian	Ido	Samoan	
Macedonian	Indonesian	Sardinian	

The following table shows all supported ICR languages:

Catalan	Finnish	Latvian	Spanish
Croatian*	French	Lithuanian	Swahili
Czech	Gaelic	Norwegian	Swedish
Danish	German	Polish	Tagalog
Dutch	Hungarian	Portuguese	Turkish*
English	Indonesian	Slovak	Welsh*
Estonian	Italian	Slovenian	

* Minor limitations

Note the following:

- Cyrillic languages and Greek are not supported for ICR.
- In Hungarian, lowercase i acute (í), o acute (ó), and u acute (ú) are not supported, which limits recognition to uppercase characters for ICR.
- There are limitations when combining Asian and Western language recognition on the same page. Asian languages are handled differently from Western languages. Only one Asian language should be set for recognition, and Western languages should not be set alongside an Asian

language. The exception is English because the Asian OCR Engine can recognize short English text embedded in any Asian language.

With CCJK OCR, English text is recognized by default without requiring English to be set. Some other languages with Latin alphabets might also be recognized, but accented characters might not always recognized correctly. For Thai, Vietnamese, and Hebrew, Western languages should not be set, except for English. Limitations have also been noted when attempting to recognize Arabic, Japanese, and English text on one page.

MICR E13B and CMC7 fonts support

The appropriate region (OmniPageOcrMicrTextRegion or OmniPageOcrCMC7TextRegion) should be added to RegionsOut property in PageLocation event. Only the necessary detection region can be set to recognize MICR.

MICR E13B and CMC7 fonts are supported by OmniPage. E13B contains digits and 4 special symbols. CMC7 has a 15-character set, comprising the 10 numeric digits and 5 control characters: internal, terminator, amount, routing, and an unused character. The control characters are set correspondingly to A, B, C, D, and E.

Output formats

The OmniPage Engine supports the following output formats, which are listed with their corresponding MIME types.

Output document type	Corresponding MIME type
Plain Text (.txt)	text/plain
Rich Text (.rtf)	text/richtext
HTML	text/html
XML	text/xml
XML Paper Specification (.xps)	application/vnd.ms-xpsdocument
PDF	application/pdf
EPUB	application/epub+zip
Microsoft Word 2007+ format (.docx)	application/vnd.openxmlformats-officedocument.wordprocessingml.document
Microsoft Excel format (.xls)	• application/vnd.ms-excel • application/excel
Microsoft Excel 2007+ format (.xlsx)	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
Microsoft PowerPoint format (.ppt)	application/vnd.ms-powerpoint

Output document type	Corresponding MIME type
Microsoft PowerPoint 2007+ format (.pptx)	application/vnd.openxmlformats-officedocument.presentationml.presentation
CSV format	text/csv

Deployment

The OmniPage Engine requires all assemblies and support files within the resource archive that is distributed to customers of the add-on.

The `x86/bin` and `x64/bin` folders are within the `resources` folder. These folders correspond to the resources for x86 and x64 processor architectures, respectively. The sets of resources are loaded upon initialization depending on which processor configuration of Atalasoft DotImage is installed and packaged with your application (x86 or x64).

Leave the folder and document structure unchanged. Any changes could adversely affect the correct initialization of the engine.

Special consideration

The OmniPage Engine enables to use the multiprocessing capabilities of the user's hardware configuration. The number of processes to run is detected automatically, depending on the number of available logical CPU cores and the number of images passed to the engine.

When in multiprocessing mode, the following limitations apply:

- PageProgress, ImageTransformation, and PageConstructed events are not fired during the OCR process.
- It is not possible to modify images in event handlers. In case of not using multiprocessing mode, OcrEngine allows to modify images in ImageIntroduction , ImageTransformation and ImageSendOff events handlers. With multiprocessing mode enabled, the engine discards image modifications.
- ImageSource objects have to be used in order to process images in parallel. Other methods that accept single images work in singleprocessing mode regardless of the OmniPageEngine.ParallelProcessing property value.
- You can only cancel in multiprocessing mode at the OcrDocumentStage.BeginPage stage.

To turn on the multiprocessing mode, do the following:

1. Set the OmniPageEngine.ParallelProcessing property to true.
2. Prepare the list of images to be recognized as ImageSources.
3. Use the OmniPageEngine.Translate() or OmniPageEngine.Recognize() methods, which accept ImageSource as a parameter.

ⓘ Even in the multiprocessing mode, recognition methods run synchronously and parallel OCR processing is performed internally.

Use an OCR engine

Using an OCR engine entails four steps:

1. Engine construction
2. Engine initialization
3. Translation of an image collection into a document
4. Engine shutdown

Engine usage example

Any number of Translate operations can be performed between Initialize and Shutdown. In most cases, you construct exactly one instance of a given engine, initialize that instance, use the object through the life of the application, and call shutdown at application close.

Although it is possible to shut down and reinitialize the engine any number of times, be aware that in some engines, initialization may be a costly operation

```
OcrEngine engine;
// Initialize your application here
 // Construct a new engine
engine = new SpecificEngine();

// Initialize the engine
engine.Initialize();

// Translate an ImageCollection into a document.
engine.Translate(imageCollection, "application/msword", outputPath);

// Terminate your application here
 // Shut the engine down
engine.Shutdown();
```

Get and set engine options

You can get or set engine options using the PreporcessingOptions property.

Get engine options

Use the PreprocessingOptions property to get the set of options currently employed by an engine. This property *only* contains options supported by the engine.

When an engine is initialized, PreprocessingOptions are set to reasonable defaults for that engine. Use the following code to reset the options to those defaults.

```
engine.PreprocessingOptions = engine.GetDefaultPreprocessingOptions();
```

Set engine options

The GetDefaultPreprocessingOptions() method is abstract and is defined by concrete OcrEngine objects. If you are unhappy with the particular set of defaults supplied by an engine, you can

change the PreprocessingOptions shortly after engine construction or, alternatively, you can subclass the engine in question and override the GetDefaultPreprocessingOptions() method.

Despite the fact that you can request that any of the options be given default values, the actual values are constrained by the options available for a given engine.

Determine if an engine supports a mime type

Translation of an image or set of images to another file format is managed through a collection of ITranslator objects contained within the engine. To translate a set of images, the engine is asked for a translator that supports a desired mime type and a specific output style. If the engine finds a translator that supports these conditions, that translator is applied to the set of images.

The set of supported mime types varies by engine. Use the engine's CanTranslate() method to determine if an engine can translate to a given mime type. The method returns true if there is an installed translator that supports a given mime type. To find out if it's possible to translate to a stream, use the engine's CanStream() method.

You can choose either of the two methods shown below to obtain this information.

Example

You can make the determination in one of the two ways shown below.

Both of the following code samples are exactly equivalent.

```
if (engine.CanTranslate(mime) && engine.CanStream(mime)) { ... }
```

```
if (engine.CanStream(mime)) { ... }
```

Alter the interpretation of page elements

A client can use the PageLocation event of type OcrPageLocationEventHandler to alter the interpretation of page elements.

For example, if a set of documents contain landmark text or form information in particular locations on a page, the locations to be searched can be reduced to just these locations. If a set of scanned pages with page numbers are out of order, they can be sorted by looking only at the bottoms and tops of the pages for page numbers.

The PageLocation event is fired after the engine has performed its location operation. A handler for this type is passed an object of type OcrPageLocationEventArgs which contains the properties listed in the table below

Property	Description
Image	Image that is currently being scanned for regions.
RegionsIn	Object of type OcrRegionCollection which contains the list of regions detected on the current page.
RegionsOut	Object of type OcrRegionCollection which contains a list of regions that should be used by the engine for this image.

RegionsIn is a read-only property. Do not modify the contents of RegionsIn. Changes to this object or its contents are ignored by the engine. Even though each of the OcrRegion objects within the collection are real objects, with the exception of the Bounds property, they have no valid content.

RegionsOut is initially set to **null**. If you wish to change the regions used for recognition on a page, make a new OcrRegionCollection and populate it with the regions you want to use.

Remove non-text regions

To export text and nothing else, use the following code shown below to remove everything but the text regions.

```
void TextFilterLocationHandler( object sender, OcrPageLocationEventArgs e)
 {
   foreach (OcrRegion region in e.RegionsIn) {
      if (region is OcrTextRegion) {
          if (e.RegionsOut == null) {
            e.RegionsOut = new OcrRegionCollection();
         }
         e.RegionsOut.Add(region);
      }
   }
 }
```

Not every engine understands every type of page element. For example, if an engine does not support the recognition of tables, then an OcrTableRegion object added to RegionsOut is ignored.

The OcrEngine that is currently locating is located in the sender object.

Determine translation type

For most OCR applications, it is important to know whether a translation is handled natively.

For an object that implements ITranslator, you can determine whether it is a native translator by checking to see if the class is an INativeTranslator. An example is provided below.

Determine if native translator is in use

The following code snippet can be used to determine if a native translator is in use.

```
if (myTranslator is INativeTranslator) { ... }
```

Distinguish between OCR region types

You can distinguish between OcrRegion object types by getting their class at runtime.

For example, the code that follows determines a region's type and draws each region type's bounding box in a different color.

Color region's bounding box ot reflect region type

The following code snippet can be used to determine if a native translator is in use.

```
Color GetRegionColor(OcrRegion region)
```

```
{
    if (region is OcrTextRegion) {
        return Color.Red;
    }
    else if (region is OcrImageRegion) {
        return Color.Blue;
    }
    else if (region is OcrBarcodeRegion) {
        return Color.Yellow;
    }
    else if (region is OcrTableRegion) {
        return Color.Green;
    }
    else if (region is OcrFormElementRegion) {
        return Color.Magenta;
    }
    return Color.black;
}
void PaintBounds(OcrRegionCollection regions, Graphics g)
{
    foreach (OcrRegion region in regions)
    {
        Rectangle destRect = region.Bounds;
        Color theColor = GetRegionColor(regions);
        Pen thePen = new Pen(theColor);
        g.DrawRectangle(thePen, region.Bounds);
        thePen.Dispose();
    }
}
```

Clean up after translation

The Finish() method allows the translator to clean up translation session specific data or whatever else is necessary after translation.

Example

The following code shows how to create the Finish() method.

```
public void Finish(OcrEngine engine, OcrDocument document, bool successful, object
 translationObject)
{
}
```

Traverse an OCR document

The Translate() method traverses the OcrDocument, writing its contents to the appropriate destination.

Translate a document into plain text file using a stream

The following example demonstrates how to translate a document into a plain text file using a stream.

```
public void Translate(OcrEngine engine, OcrDocument doc, string mimeType,
                      System.IO.Stream outStream, object translationObject)
```

```
{
    System.IO.StreamWriter writer = new System.IO.StreamWriter(outStream);
    foreach (OcrPage page in doc.Pages) {
        foreach (OcrRegion region in page.Regions) {
            if (region is OcrTextRegion){
                OcrTextRegion textRegion = (OcrTextRegion)region;
                foreach (OcrLine line in textRegion.Lines) {
                    for (int i=0; i < line.Words.Count; i++){
                        writer.Write(word.Text);
                        if (i < line.Words.Count-1){
                            writer.Write(" ");
                        }
                        else {
                            writer.Write("\n");
                        }
                    }
                }
            }
        }
    }
}
```

Cancel OCR in progress

To cancel an OCR in progress

1. Create an OcrPageProgressEventHandler.
2. Set the Cancel property of the EventArgs to true.
3. Hook into the event handler.

Create OCR page progress handler

The following code example shows how to create an OCR page progress handler.

```
private void PageProgressHandler(object sender, OcrPageProgressEventArgs e)
{
    //Boolean set elsewhere in your code
 if (CancelHasBeenRequested) {
        e.Cancel = true;
    }
}
```

Hook into an event handler

The following code example shows how to hook into the event handler.

```
// install the handler
engine.PageProgress += new OcrPageProgressEventHandler(PageProgressHandler);
```

Track page progress in a UI

To track page progress, create a PageProgressEventHandler and translate the information in the event arguments to your UI.

1. Translate OCR Page Stage Information to a string.
2. Create the event handler.

3. Install the event handler.

Translate the OcrPageStage information to a string

The first step in this task is to translate the OcrPageStage information to a String, which can be done with a method like the one shown in the code.

```
private string TranslatePageStageToString(OcrPageStage stage)
{
    string message = "";
    switch (stage) {
    case OcrPageStage.Analysis:
        message = "Analyzing";
        break;
    case OcrPageStage.Canceling:
        message = "Canceling";
        break;
    case OcrPageStage.Export:
        message = "Exporting";
        break;
    case OcrPageStage.Location:
        message = "Locating";
        break;
    case OcrPageStage.Postprocessing:
        message = "Post Processing";
        break;
    case OcrPageStage.Preprocessing:
        message = "Preprocessing";
        break;
    case OcrPageStage.Recognition:
        message = "Recognition";
        break;
    case OcrPageStage.Spellchecking:
        message = "Checking spelling";
        break;
    default:
        message = "Unknown stage";
        break;
    }
    return message;
}
```

Create the event handler

The next step is to create the event handler.

```
private void PageProgressHandler(object sender, OcrPageProgressEventArgs e)
{
    string message = TranslatePageStageToString(e.Stage);
    MyUITextDisplay.Text = message;
    MyPageProgressBar.Value = e.Progress;
}
```

The final step is to install the event handler.

Use page deskew events

Atalasoft DotImage OCR now includes a built-in facility for automatically deskewing images. In previous versions, this facility was provided by the OcrEngine.

In some cases, an OcrEngine will not provide this facility, as the quality might not be acceptable or because it is required that the image be bi-level, making antialising or creation of high quality thumbnails impractical. More importantly, coordinates of bounding boxes of page elements (lines, words, images, and so forth.) are accurate since any coordinate transform and image dimension changes happen before the actual OcrPage object has been constructed.

All engines report Deskew as an available preprocessing option and they also report that it is part of the default set of preprocessing options.

Out of necessity, deskew is performed before all other preprocessing operations and is not included in the preprocessing event. In other words, when the preprocessing event is fired, the OptionsIn flag will never have the deskew bit set.

In addition, the OcrEngine now adds an event for tracking the deskew operation specifically. The following example shows how to track a deskew event.

Deskew method

To track a deskew event, a client implements a method like that shown below.

```
void PageDeskewedHandler(object sender, OcrPageDeskewdEventArgs args)
{
}
```

The event arguments object includes the following members listed below.

Property	Description
Page	OcrPage that will be associated with this image
BeforeImage	AtalaImage that was deskewed
AfterImage	Image that was created. If AfterImage is null, there was no deskew operation
Angle	Detected angle of rotation
IsDeskewed	*true* if a deskew operation was performed

This event allows the client to perform additional processing, to inform a UI about the deskew amount and to cache the final image.

The deskewed image, if any, is owned and managed by the OcrEngine. It is not recommended that client code keep a reference to the deskewed image, as it is very likely that it will be disposed by the Engine.

Deploy an OCR engine

Each OCR engine has different deployment requirements. We have tried to formalize this process as much as possible by providing guidelines on the mechanism for deployment.

This shows how you can ensure that an OcrEngine can start and is able to find its resources.

Loading and locating folders

Before thinking about deploying an application it is helpful to know about the following key file folders.

Folder	Typical file name or location	Contents
SDK	`C:\Program Files\Atalasoft \DotImage 11.4\Bin`	All the Atalasoft DotImage assembly files as installed as part of the Atalasoft DotImage SDK.
OCR resources engine resources	`C:\Program Files\Atalasoft \DotImage 11.4\Bin \OcrResources` such as `C:\Program Files \Atalasoft\DotImage 11.4\Bin\OcrResources \GlyphReader\v5.0`	All OCR Engine resources. Individual OCR Engine's resource files.
32-bitengine module	`C:\Program Files \Atalasoft\DotImage 11.4\Bin\OcrResources \v5.0\x86\GlyphReader.dll`	Engine supplied 32-bit dll that provides engine functionality.
64-bitengine module	`C:\Program Files \Atalasoft\DotImage 11.4\Bin\OcrResources \v5.0\x64\GlyphReader.dll`	Engine supplied 64-bit dll that provides engine functionality.
assembly	`C:\Program Files\Atalasoft \DotImage 11.4\Bin\3.5`	Atalasoft DotImage assembly files (such as Atalasoft.dotImage.Ocr.dll), this may be the same as the application folder.
application	___Your choice	Install location for your application.

Most of the work of loading and locating resources is managed by Atalasoft DotImage or by the engine itself. In custom situations, however, the client may have work to do.

Before you can use most engines, the following must be available:

- An engine module is needed for some aspects of OCR functionality.
- Resource files used to configure the engine or otherwise provide necessary data or services. This may include such things as dictionaries, grammar rules, glyph shapes, neural networks, and so on.

An engine that requires engine modules typically needs to have those modules loaded before it attempts to construct a class. The assembly that uses the engine module should contain the knowledge of how to find the engine module but the engine module needs to be loaded before the module that needs be able to find it is loaded. Atalasoft DotImage OCR tries to handle this conundrum for you when possible but there are some cases, as outlined below, where you must handle the problem yourself.

Options for the developer

You can select from the options in the table.

Option	Consequences
Leave the engine module in the OcrResources folder as shipped.	You must put the OCR resources folder within the assembly folder.
Move the OcrResources folder in the location of your choice.	You must load the dll. If the OcrResources folder is not in the assembly folder, you are required to pass its location in to the OcrEngine's constructor or resource loader.
Move the engine module out of the OcrResources folder.	If the engine module is put into the application folder or the assembly folder, it should be located automatically. If the engine module is located somewhere else, you must locate the module and load it. If the OCR resources folder is within the assembly folder, you can pass null to the engine constructor for the path, otherwise you must pass the location in.

Access document information properties

The OcrDocument class provides the document information properties listed in the table below.

Except for Metadata, these are not true properties in that they are not contained within the OcrDocument object itself. The actual values are contained within an object in the Metadata hashtable.

To retrieve the document information properties, you need code similar to the following.

Retrieve document information properties

```
public static Hashtable GetDocumentInfo(OcrDocument document)
 {
    object infoObject = document.Metadata[OcrDocumentMetadataKey.DocumentInfo];
    if (infoObject == null)
        throw new Exception("Unable to find document info.");
    Hashtable infoTable = (Hashtable)infoObject;
    return infoTable;
 }
```

Access document information properties

```
public static object GetDocumentInfoValue(OcrDocumentInfoKey key)
 {
    Hashtable ht = GetDocumentInfo(document);
    if (ht == null)
        throw new Exception("Unable to get document info.");
    return ht[key];
 }
```

Color management

Atalasoft DotImage enables images to be correctly adjusted based on color profiles and the international standard established by the International Color Consortium. You need only minimal knowledge of color management or ICM 2.0 to add full color management support to your applications using Atalasoft DotImage.

The ImageViewer and WorkspaceViewer controls have built-in support for color management with color profiles. Atalasoft DotImage uses Little CMS Color Management to automatically transform an image for display based on the device and output profiles. To enable automatic color management in the control, set the ColorManage property to true. With color management enabled, the colors are adjusted based on the color profile associated with the image (See the AtalaImage.ColorProfile property) and the color profile associated with the display device. If no profile is found for either the image or the monitor, the default sRGB profile is used.

Color profiles

Color profiles are used to adjust colors so that they correctly match the desired visual or printed color when you display an image. You can also use color profiles to convert images from one color space to another (for example, from CMYK to RGB), store them on a disk as ICM or ICC files, or embed them in an image. Atalasoft DotImage supports reading color profiles from a file, or in TIFF, PNG, or JPEG images.

You can use the ColorProfile object in the ColorManagement namespace to set the ColorProfile property in an AtalaImage object. Normally, an AtalaImage object would not have a color profile; it's value would be set to null. However, if the image has an embedded profile, the ColorProfile value is set, and the specified color profile is used when converting between color spaces. The ColorProfile is most relevant when you display a CMYK image, and when you convert from the CMYK color space to the RGB color space.

CMYK images

CMYK images are commonly used in the pre-press industry. They contain color values for cyan, magenta, yellow, and black. These images occupy a different color space than RGB images, the most commonly used color space for image formats, because the colors are subtractive. For example, when you increase the value of one of the four color values, that color is actually subtracted from the image. For example, increasing the value of cyan in an image subtracts cyan from the color. In CMYK images, the color value 0,0,0,0 is white, while in RGB images, the absence of color is black.

Because CMYK displays are additive and no color is perceived as black, you must convert a CMYK image to an RGB image to display it. Because Atalasoft DotImage uses Color Profiles (commonly saved as ICC or ICM files), it can display CMYK images properly, without losing the integrity of the data. Atalasoft DotImage does this by converting the CMYK image to RGB for display, and then creating a cache of the image.

Most Atalasoft DotImage commands work on CMYK images.

Draw on the canvas

The Atalasoft DotImage Drawing namespace can be used to draw text, lines, rectangles, and other primitives onto images and graphic objects. The Drawing namespace is very similar to native GDI+ drawing in .NET, but while GDI+ can draw only on RGB and RGBA images, Atalasoft DotImage can draw on RGB, RGBA, CMYK, grayscale, grayscale-alpha, colormapped, and 1-bit images.

The canvas

Before drawing on an image, a Canvas object must be created. The constructor of the Canvas object requires either an AtalaImage object or a Graphics object.

The canvas is similar to .NET's graphics object in that it is where primitives and text are drawn onto other images. All drawing operations are drawn directly onto the AtalaImage or the Graphics object.

Atalasoft DotImage drawing versus GDI+

The Drawing namespace in Atalasoft DotImage is not meant entirely to replace the drawing features in .NET's GDI+ implementation. It is designed to allow drawing text and primitives on CMYK, grayscale, colormapped, and binary images where GDI+ requires RGB or RGBA.

GDI+ offers more features, and is a good choice when drawing onto RGB or RGBA images.

The following table explains the key differences between the Atalasoft DotImage Drawing namespace and GDI+.

Feature	**Atalasoft DotImage drawing (Canvas)**	**.NET GDI+ crawing (graphics)**
RGB and RGBA support	X	X
Grayscale support	X	
CMYK support	X	
Color-mapped and 1-bit black and white support	X	
Floating point positioning and accuracy		X
Antialiasing*	X	X
Draw borders and fills for solid entities	X	
Hatched fills	X	X
Set the transparency	X	X
Geometric transforms XOR inverted pens	X	
* GDI+ uses different antialiasing technology than Atalasoft DotImage.		

Draw text

Using Atalasoft DotImage to draw high quality text involves setting the FontQuality property in the Canvas class to a value of ClearType of Antialias. ClearType is only supported on XP Machines, while Antialias is supported on all machines.

The following illustrates the differences in FontQuality as it relates to drawing text with DrawText:

- NonAntialiased
- Antialiased
- ClearType

Draw primitives and text onto a workspace or WorkspaceViewer object

The following example shows how to draw primitives and text onto a Workspace or WorkspaceViewer object. It draws a blue-filled rectangle with a red border and then draws transparent text on top of the rectangle.

ℹ To update the display in a WorkspaceViewer after using the Canvas object, the Refresh() method must be invoked to cause a repaint and update the cached display image.

C#

```
Canvas myCanvas = new Canvas(myWorkspaceViewer.Image);
myCanvas.DrawRectangle(new Rectangle(10, 10, 100, 100),
   new AtalaPen(Color.Red, 5), new SolidFill(Color.Blue));
myCanvas.DrawText("Atalasoft DotImage", new Point(20, 20), new Font(new
 FontFamily("Arial"), 24),
   new SolidFill(Color.FromArgb(127, Color.Black)));
//Update the display
myWorkspaceViewer.Refresh();
```

Draw shapes

It is possible to draw text, lines, circles, rectangles, and other shapes, but they are not visible until they are "burned" into the image. You cannot edit them once they are drawn.

For example, suppose the user creates a rectangular selection (RubberBand) on the image. When the mouse is released, the event can automatically post back to the server, and programmatically draw a circle on the image, using the bounding box that the user just created.

1. Set the AutoPostBack property of the WebImageViewer 's Selection, to **true**.
2. Create a new event handler for the WebImageViewer' s SelectionChanged event similar to the method shown in the following example.

Draw a rectangle using the rubber band selection mousetool

C#

```
Canvas myCanvas = new Canvas(this.WebImageViewer1.Image);
Rectangle mySelection = this.WebImageViewer1.Selection.Rectangle;

// Draws a black rectangle, with a semi-transparent orange fill
```

```
myCanvas.DrawRectangle(mySelection, new AtalaPen(Color.Black), new
 SolidFill(Color.FromArgb(128, Color.DarkOrange)));

// Resets the Selection so it's no longer there
this.WebImageViewer1.Selection.Reset();

// Notifies the control that the Image was modified
this.WebImageViewer1.Update();
```

Draw with rubber bands

The Atalasoft DotImage Rubberband class provides a convenient way to allow a user draw onto an image.

When using a rubberband for WinForm drawing, be sure that the Persist property is set to false. This causes the Rubberband to disappear once the entity is drawn.

The RectangleSelection can be used to draw rectangles, but the Persist property is true by default. In addition, the pen in the RubberBand probably should be set to the same pen being used to draw on the image. In this case, it renders the entity onto the image, and only permanently marks it after the user depresses the mouse button.

1. Open Visual Studio and start a new WPF Windows Application project.
2. Drop a RectangleRubberband onto the form and set the Parent property to the WorkspaceViewer that was just added Drop a WorkspaceViewer or ImageViewer onto the form and set the Image property to any image.
3. In the RectangleRubberband , set the Active property to **true**.
4. Change the Pen properties to the desired color and style. Be sure to turn Inverted off so the rubber band draws with a particular color.
5. To add code to the changed event, in the property grid, click the Events button, and then double-click the Changed event in C#. Enter the following code.

   C#

   ```
   private void rectangleRubberband1_Changed(object sender,
       Atalasoft.Imaging.WinControls.RubberBandEventArgs e)
   {
       Atalasoft.Imaging.Drawing.Canvas myCanvas =
           new Atalasoft.Imaging.Drawing.Canvas(workspaceViewer1.Image);
       myCanvas.DrawRectangle(e.GetBounds(), rectangleRubberband1.Pen);
       workspaceViewer1.Refresh();
   }
   ```
6. Finally, run the project and draw away! Keep in mind that you can draw other shapes by using other Rubber band objects, such as the EllipseRubberband and LineRubberband.

Interoperability - Work with GDI+ images

You can use Atalasoft DotImage to:

- Directly process native .NET Bitmap images by creating a copy of the data.
- Pass the data directly into the AtalaImage constructor.

Create a copy of the data

To process a native .NET Bitmap image by creating a copy of the data:

1. Create an AtalaImage copy of the Bitmap.
2. Process the image.
3. Create a bitmap copy of the results.

ℹ Atalasoft DotImage has a built-in type converter that can be used to cast a bitmap to an AtalaImage and vice versa.

Pass the data directly to the AtalaImage constructor

Manipulate the Bitmap bits directly by creating an AtalaImage that points to the same block of memory as the Bitmap.

ℹ Directly manipulating the bits is more difficult than creating a copy, but does conserve memory.

Process a bitmap directly

You can create an AtalaImage directly from a Bitmap object using the static AtalaImage.FromBitmap() method.

ℹ When you use this method, AtalaImage takes ownership of the Bitmap and its associated memory and manages its disposal. Assume that when you create an AtalaImage from a Bitmap, you lose permission to access the Bitmap object.

The following example shows direct processing of a bitmap.

Create a bitmap object using AtalaImage.FromBitmap() method

C#

```
AtalaImage image = AtalaImage.FromBitmap(srcBitmap);
BlurGaussianCommand blur = new BlurGaussianCommand(2);
AtalaImage newImage = blur.Apply(image).Image;
// Dispose of srcBitmap
image.Dispose();

//Bitmap only points to the newImage.
//  The newImage still controls the memory and it must be disposed when done.
Bitmap bm = new Bitmap(newImage.Width, newImage.Height, newImage.RowStride,
   (System.Drawing.Imaging.PixelFormat)((int)newImage.PixelFormat),
   PixelMemory.PixelDataFromPixelMemory(newImage.PixelMemory));
```

Interoperability - Work with WPF images

This topic explains how to use the AtalaImageViewer in a WPF application.

The example code that follows is written in C#; however any CLS compliant language can be used.

Add the AtalaImageViewer control to a WPF windows application

You have several options for creating a WPF application. The following example uses Visual Studio and its XAML source editor to create the project.

1. Open Visual Studio and start a new WPF Windows Application project.
2. Add the following references:
   - Atalasoft.dotImage
   - Atalasoft.dotImage.Lib
   - Atalasoft.dotImage.Wpf
   - Atalasoft.Shared
3. Double-click Window1.xaml to launch the editor.
4. Use the form designer to set the window size, title and other common options.
5. Switch to XAML source view.
6. Add the following XML namespace to the **Window** tag:

   ```
   xmlns:atala="clr-
   namespace:Atalasoft.Imaging.Wpf;assembly=Atalasoft.dotImage.Wpf"
   ```

7. Inside the **Grid** tag, add the following code:

   ```
   <Grid.ColumnDefinitions>
       <ColumnDefinition/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
       <RowDefinitionHeight="22"/>
       <RowDefinition/>
   </ Grid.RowDefinitions>
   <MenuGrid.Column="0"Grid.Row="0">
       <MenuItemHeader="_File">
           <MenuItemHeader="_Open"Click="OnOpenFile"/>
           <MenuItemHeader="_Save"Click="OnSaveFile"/ >
               <Separator/>
               <MenuItemHeader="E_xit"Click="OnExit"/>
           </MenuItem>
       </Menu>
       <atala:AtalaImageViewer Name="Viewer" Grid.Column="0" Grid.Row="1"/>
   ```

8. Open the Window1.xaml.cs file and add the file menu event handlers.
9. Build and run the application.

Use mouse tools

AtalaImageViewer has a MouseTool property that takes any class deriving from the MouseTool class. This allows you to create custom mouse tools for the viewer. The Atalasoft DotImage WPF component provides several commonly used tools including selection, panning, magnifier and zoom.

The following code tells the viewer to use the panning tool:

```
this .Viewer.MouseTool = new PanningMouseTool();
```

All the viewer mouse tools have common default values making it easy to switch between the tools. You may want to modify the look or behavior of a tool to better fit your application or preference.

For instance, the PanningMouseTool can have two cursors: one for the normal cursor and another for a mouse down (grab) cursor.

Forms processing

Forms processing brings the ability to analyze and process documents containing user-filled forms to the Atalasoft DotImage Document Imaging toolkit.

Key features:

- Document alignment: With the DocumentAligner class you can align one document to a predefined template. This allows for the reliable extraction of individual barcodes, small sections of text via OCR, and marks via OMR.
- Alignment rejection: After performing alignment, use the AlignmentRejector class to test if the alignment was successful.
- OMR (Optical Mark Recognition): Using the OmrEngine class you can check for expected marks on a document.

Align an image to a template

Aligning an image to a template is a three-step process:

1. Generate PageFingerprints with the PageFingerprintGenerator.
2. Generate an AlignmentResult by passing those fingerprints into the DocumentAligner.
3. Validate that AlignmentResult with the AlignmentRejector.

C#

```
//Load Images
AtalaImage sourceImage = new AtalaImage(@"C:\temp\sourceImage.tif");
AtalaImage targetImage = new AtalaImage(@"C:\temp\targetImage.tif");

//Generate PageFingerprints
PageFingerprintGenerator generator = new PageFingerprintGenerator();
PageFingerprint sourceFingerprint = generator.GenerateFingerprint(sourceImage);
PageFingerprint targetFingerprint = generator.GenerateFingerprint(targetImage);

//Align with PageFingerprints
DocumentAligner aligner = new DocumentAligner();
AlignmentResult result = aligner.Align(sourceFingerprint, targetFingerprint);

//Validate Alignment
AlignmentRejector rejector = new AlignmentRejector();
rejector.TestAlignmentResult(result);
```

Disable alignment rejection heuristics

In some cases you might find that one of the AlignmentRejector's alignment rejection heuristics is incorrectly reporting your document as misaligned. In this case, you can disable this particular heuristic by removing it from the AlignmentRejectionHeuristics enumeration property with the exclusive-or operator.

C#

```
AlignmentRejector rejector = new AlignmentRejector();
rejector.AlignmentRejectionHeuristics ^=
 AlignmentRejectionHeuristics.AlignmentConfidence;
```

Use the OMR engine to recognize marks on a page

Given an existing OmrTemplateDocument and an FileSystemImageSource, basic usage of the OmrEngine class is a three step process:

1. Create an OmrEngine instance.
2. Pass your document and template in to the OmrEngine.
3. Parse the OmrEngine results.

C#

```
//Create engine instance
OmrEngine engine = new OmrEngine();

//Load OMR Template
OmrTemplateDocument docTemplate = OmrTemplateDocument.Load(@"C:\temp
\document.template");

//Load Document to perform OMR on
FileSystemImageSource imageSource = new FileSystemImageSource(@"C:\temp\omrimage.tif",
 true);

//Perform OMR
OmrDocument results = engine.RecognizeDocument(imageSource, docTemplate);

//Parse Results
foreach (OmrPage page in results.Pages)
{
    foreach (OmrGroup group in page.Groups)
    {
        foreach (OmrMark mark in group.Marks)
        {
            string markString = String.Format("Mark {0}: {1}",
                                              mark.Template.Name,
                                              mark.IsMarked);
            System.Console.WriteLine(markString);
        }
    }
}
```

Create and save an OMR template

Creating OMR templates is simply a matter of stacking together the various OmrTemplate classes. Each represents a different template concept:

- An OmrTemplateDocument represents an entire document.
- An OmrTemplatePage represents a single document page.
- An OmrTemplateGroup represents a group of marks.
- An OmrTemplateMark represents a single location to check for a mark.

1. Create an OmrTemplateDocument to hold your template pages:

```
OmrTemplateDocument templateDoc = new OmrTemplateDocument();
```

2. Whenever you open a new image as a page for this template, you'll also want to create a new OmrTemplatePage to go with it.

```
AtalaImage templateImage = new AtalaImage(@"C:\Temp\templateImage.tif");
OmrTemplatePage templatePage = new OmrTemplatePage(templateImage);
templateDoc.Pages.Add(templatePage);
```

3. Create a group to hold your marks, which are currently for organizational purposes but can be used for more complex tasks. When creating a group, assign it a name that can be used to find it after processing. For this reason, each group's name should be unique within the page.

```
String groupName = "Group One";
OmrTemplateGroup templateGroup = new OmrTemplateGroup(groupName);
templatePage.Groups.Add(templateGroup);
```

Perform additional preprocessing in the OMR engine

Sometimes you may wish to do additional image preprocessing before entering the rest of the OMR process.

1. Create an EventHandler<OmrImagePreprocessingEventArgs> which:
   - Performs the desired preprocessing.
   - Assigns the result to the InMarkedImage property of the OmrImagePreprocessingEventArgs.
2. Install the event handler.

Create an OmrImagePreprocessingEventArgs event handler

As images are passed in to the OmrEngine do not have the AutoDeskewCommand applied by default, this is one example of when you may want to perform additional preprocessing.

1. Perform the desired preprocessing.
2. Assign the result to OmrImagePreprocessingEventArg's InMarkedImage property.

```
void PreprocessingHandler(object sender, OmrImagePreprocessingEventArgs e)
{
    AutoDeskewCommand command = new AutoDeskewCommand();
    AtalaImage newImage = command.Apply(e.InMarkedImage).Image;
    e.InMarkedImage = newImage;
}
```

3. Install the handler into the OmrEngine instance's ImagePreprocessing event.

```
omrEngine.ImagePreprocessing += new
 EventHandler<OmrImagePreprocessingEventArgs>(PreprocessingHandler);
```

Cancel OMR in progress

Use this procedure to cancel an OMR session that is in progress.

1. Create a PageProgress event handler which sets the Cancel property of the OmrPageProgress EventArgs to **true**.
2. Install the event handler.

Create an OMR page progress handler

The following code example shows how to create an OMR page progress handler that cancels processing when the class-wide boolean CancelHasBeenRequested is set to **true**.

1. When the desired conditions for cancellation are met, set the Cancel property to true.

```
private void CancelPageProgressHandler(object sender, OmrPageProgressEventArgs e)
{
    //Boolean set elsewhere in your code
    if (CancelHasBeenRequested)
    {
        e.Cancel = true;
    }
}
```

2. Install the handler into the OmrEngine instance's PageProgress event.

```
// Install the handler

omrEngine.PageProgress += new
 EventHandler<OmrPageProgressEventArgs>(CancelPageProgressHandler);
```

Track OMR engine page progress in a UI

To track page progress, create an EventHandler<OmrPageProgressEventArgs> and translate the information in the event arguments to your UI.

1. Create an event handler that:
   - Translates OMR Page Stage Information to a String.
   - Passes that string and current progress to your UI.
2. Install the event handler.

Translate the OmrPageStage information to a string

1. Translate the OmrPageStage information to a String.

```
private string TranslatePageStageToString(OmrPageStage stage)
{
    string message = "Unknown Stage";
    switch (stage)
    {
        case OmrPageStage.Preprocessing:
            message = "Preprocessing Image and Template";
            break;
        case OmrPageStage.Aligning:
            message = "Aligning Template";
            break;
        case OmrPageStage.Recognizing:
            message = "Recognizing Marks";
            break;
        case OmrPageStage.Finishing:
            message = "Finishing";
            break;
    }
    return message;
}
```

2. Create the event handler.

```
private void TrackingPageProgressHandler(object sender, OmrPageProgressEventArgs
 e)
{
    string message = TranslatePageStageToString(e.Stage);
    TextToShow = message;
    if (e.Progress.HasValue)
        ProgressBarValue = e.Progress.Value;
}
```

3. The handler must be installed into your OmrEngine instance's PageProgress event.

```
omrEngine.PageProgress += new EventHandler
<OmrPageProgressEventArgs>(TrackingPageProgressHandler);
```

Web Document Viewer

The WebDocumentViewer is JavaScript based image viewing control that can be created on the client side without the need for a traditional WebServerControl back end. It communicates directly with a WebDocumentRequestHandler on the server side, so there are no page lifecycle problems to deal with.

A WebDocumentViewer only requires a few snippets of HTML and JavaScript on your page, and a separate bare-bones handler.

The WebDocumentViewer doesn't have a Toolbox item to drag onto a form, so you can create the control on any page that you need to use it, without forms. See our Web Document Viewer Guide for a step-by-step tutorial of setting up a WebDocumentViewer in a new project and deploying it to an IIS server. A complete example of the WebDocumentViewer is also included in the DotImageWebForms demo projects that are installed with Atalasoft DotImage.

The Web Document Viewer online documentation is available at `https://atalasoft.github.io/web-document-viewer`. The offline verison can be downloaded from the public GitHub repository at `https://github.com/Atalasoft/web-document-viewer/tree/master/docs`.

Chapter 4

Deploy Atalasoft DotImage

Atalasoft DotImage does not contain COM components to register, and no Registry modifications are required to use the SDK. To deploy the SDK, copy Atalasoft DotImage assemblies alongside your EXE.

Visual C++ Runtime dependencies

Atalasoft DotImage is distributed in several configurations, which are listed in the *Atalasoft DotImage Technical Specifications*.

Deploy Atalasoft DotImage in ASP.NET

When deploying Atalasoft DotImage in an ASP.NET application, the Atalasoft DotImage license file must be located in the bin directory of the application.

Dependencies using Atalasoft DotImage class library

The following files must be included on the server that usesAtalasoft DotImage. This is all that is required when using the class library only:

- `Atalasoft.dotImage.dll`
- `Atalasoft.dotImage.Lib.dll`
- `Atalasoft.Shared.dll`
- `Atalasoft.dotImage.lic`

All of these files must be placed in the application's `bin` folder.

Dependencies using Atalasoft DotImage with WebControls

The following files must be included on the server that uses Atalasoft DotImage with WebControls:

- `Atalasoft.dotImage.dll`
- `Atalasoft.dotImage.WebControls.dll`
- `Atalasoft.dotImage.Lib.dll`
- `Atalasoft.Shared.dll`
- `Atalasoft.dotImage.lic`
- `Atalasoft.dotImage.Pdf.dll`
- `Atalasoft.dotImage.PdfReader.dll`

- `Atalasoft.dotImage.PdfDoc.Bridge.dll`
- `Atalasoft.dotImage.PdfDoc.dll`
- `Atalasoft.dotImage.Ocr.dll`
- `Atalasoft.dotImage.AdvancedDocClean.dll`

All of these files must be placed in the application's `bin` folder.

Generating licenses

To license application components, a license file is generated or updated and compiled into the project output.

The licenses.licx file is generated or updated automatically by Windows Form Designer when a licensed control is added to a form. For console application, this file is added manually as shown in HOWTO: License an EXE for Deployment on the Atalasoft website. During compilation, the project system transforms licenses.licx into a .licenses binary resource that provides support for .NET control licensing. The binary resource is embedded in the project output.

For .NET Framework, use the License Compiler (lc.exe) to compile and embed the license binary resource. (See the Microsoft website for instructions.) For .NET 5 or later, the License Compiler is not supported. Instead, use the Atalasoft License Compiler (AtalasoftLicenseCompiler.exe) provided with Atalasoft DotImage to transform and embed the license binary resource. Just like the License Compiler, the Atalasoft License Compiler takes the licenses.licx file that was generated or updated by Windows Form Designer or added manually, transforms the file into a .licenses binary resource, and embeds it into the project output.

The Atalasoft License Compiler can be run separately, and it uses the same command-line arguments as the License Compiler, as in this example:

```
AtalasoftLicenseCompiler.exe
/complist:<licenses.licx_path>
/outdir:<result_folder_path> /target:<application_name>
/i:"<refassembly1>;<refassembly2>;<refassembly3>;..;<refassemblyN>"
```

But to embed licensing, you need to install the Atalasoft.dotImage.AtalasoftLicenseCompiler.x86 or Atalasoft.dotImage.AtalasoftLicenseCompiler.x64 NuGet package for .NET 5 project. The NuGet package includes AtalasoftLicenseComplier.exe and the appropriate targets and instructions for *.licenses generation. Targets are added to the .csproj file during compilation.

To use the Atalasoft License Compiler, follow these steps:

1. Install the NuGet package, either Atalasoft.dotImage.AtalasoftLicenseCompiler.x86 or Atalasoft.dotImage.AtalasoftLicenseCompiler.x64.
2. Create or add the licenses.licx file.

   If you create the file, make sure it is in `<project folder>/Properties`. If you add it, follow the instructions in HOWTO: License an EXE for Deployment on the Atalasoft website.

3. Build the project.
   During compilation, the following takes place:
   a. The AtalasoftLicenseComplier.exe utility and necessary assemblies are copied to `<destination folder>/lib`.
   b. The <application name>.licenses file is generated and embedded into the resulting application file.
4. Check the build log file for any errors.

   If the license is not found for the assembly, an error message is added to the build log, but the build does not fail.

Chapter 5

Program with DotPdf

DotPdf is a set of tools used for creating or manipulating PDF documents. PDF is a file format created by Adobe Systems that is used to represent the content and structure of a document in a way that the appearance of the document will maintain its quality independent of the device on which it is displayed. For example, TIFF documents are scanned images that only look as good as the resolution of the scan, whereas PDF documents can contain text and graphic content that do not have a fixed resolution and render well on low or high resolution devices.

In addition, PDF can contain a number of interactive features including hyperlinks, annotations, bookmarks.

DotPdf includes two main tools for operating on PDF files:

- PdfDocument - Object for performing efficient, document-level manipulation of PDF documents, including rearranging or deleting existing pages, adding pages from another document, creating or editing the bookmark tree, creating or editing document metadata, or combining multiple documents into one.
- PdfGeneratedDocument - Object capable of doing everything PdfDocument can do, but requires reading in the full content of the document. In addition, PdfGeneratedDocument can be used for adding content to existing pages and creating new content from scratch.

Both PdfDocument and PdfGeneratedDocument have the ability to detect and repair many types of broken or non-compliant PDF documents.

The PDF document format is a standard format that describes the appearance layout, and to a certain extent the behavior of a collection of pages. PDF documents are designed to look consistently good on whatever device is used to display them, whether the device is a computer screen, a desktop printer, a phototypesetter, or a cell phone. Unlike most image formats, PDF has no sense of resolution. This means that a document can viewed at arbitrary magnification with little or no loss of information.

The Atalasoft PDF Generating library provides a mechanism for creating PDF documents that is simple, consistent, and extensible. Since the underlying document format is complicated, the library is built to separate the document format from the means used to create the document. Client code needs to concern itself with the content and the mathematical modeling. The actual production of PDF from this is handled behind the scenes.

In addition to basic shapes, images and text, the Atalasoft library has tools for creating your own shapes from primitive shapes, composites of basic shapes, as well the ability to stitch all of these together into high-level tools for creating documents from very little code.

To create a PDF document, one needs to make a document object, add pages to the document, put content onto the pages and save the document. The following example demonstrates how to make a basic PDF:

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();

PdfGeneratedPage page = PdfDefaultPages.Letter;

doc.Pages.Add(page);

string font = doc.Resources.Fonts.AddFromFontName("Times New Roman");

PdfTextLine line = new PdfTextLine(font, 12, "Hello, PDF", new PdfPoint(72, 400));

page.DrawingList.Add(line);

doc.Save("hello.pdf");
```

The authoring library has seven main components: resources, pages, drawing primitives, shapes, forms, annotations and rendering. Resources are collections of large objects that may be used multiple times on a page or a document such as fonts or images. Resource objects are named and are always referred to by name. Pages are objects that contain dimensions as well as a list of drawings that make the visible contents of the page. Pages may be moved freely from one document to another, cloned and serialized. Drawing primitives are objects that can directly generate PDF page content. Primitives include paths, rectangles, primitive text, and images. Shapes are higher level objects that are more easily described and controlled and may include transforms to apply to the shape like scale and rotation. Shapes can be built in terms of primitives or in terms of other shapes. Rendering is the process of turning a collection of pages and their content into PDF or some other format. Although most applications concerned with making PDF documents will only need to concern themselves with resources, pages and shapes, the Atalasoft library is designed to be open and extensible. Advanced applications can work with primitives directly, create their own higher level shapes or create their own renderers. And while the rendering process is typically invisible to client code, the mechanism is open so that documents can be created that are limited only by the PDF specification.

Mathematical model

In PDF, a page is based on a formal Cartesian coordinate system. In this model, the origin is in the lower left corner of the page with the positive X axis stretching to the right and the positive Y axis extending up. Units are in PDF standard units which are 1/72 of an inch. Coordinates are expressed in floating point numbers. Every page includes an Affine transformation matrix through which all coordinates are pushed before being placed on the page.

ℹ This differs from conventional image coordinates where the origin is in the upper left corner of the image and the positive Y axis extends down.

For drawing, there are five main primitives: paths, rectangles, images, text, and templates. A path is a collection of lines and Bezier curves. Paths may be disjoint or non-disjoint. In non-disjoint paths, all elements are connected. A non-disjoint path may be closed or open. In a closed path, there is an explicit step to connect from the first element in the path to the last element in the path. A disjoint path may consist of any number of sub paths which may be open or closed.

Paths and rectangles are placed on the page. After a shape has been placed on the page, it may be stroked, filled or clipped. Outlines in the path may be stroked with solid or dashed lines. Line ends may rounded, square projecting, or square flat. Line joints may be beveled or mitered. Paths may

filled with solid colors. Clipping and filling are done based on one of two different filling rules, the even-odd rule and the non-zero winding rule.

Images in PDF are considered to be 1 by 1 in PDF units. To place an image on the page, one sets a transform to set the location and size of the image on the page.

Templates are encapsulated collections of other PDF primitives. In PDF Generating they are intended for two main purposes: creating reusable page content like letterhead, backgrounds or watermarks. Templates can also be used for building transparency or blending layers.

Transformations

The PDF imaging model includes the notion of a current transformation. All objects that are rendered get pushed through the transformation before being rendered.

Transformations are represented by an Affine transformation matrix which is a 3x3 matrix of the form:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

When a point (x, y) is transformed by the matrix, the output of the transformation will be (x', y'), where $x' = ax + cy + e$ and $y' = bx + dy + f$. In the Atalasoft Pdf Generating library, transformations are represented by the class PdfTransform. Within that class there are some factory methods for making common transformations.

PdfTransform.Identity() returns a new identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

PdfTransform.Translate(double x, double y) returns new matrix that will perform a translation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{bmatrix}$$

PdfTransform.Scale(double s) returns a new matrix that will perform a uniform scale:

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

PdfTransform.Scale(double x, double y) returns a new matrix that scales in x and y directions, possibly by different amounts:

$$\begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

PdfTransform.Rotate(double theta) returns a new matrix that will perform a counter clockwise rotation by theta radians:

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

PdfTransform.Skew(double x, double y) performs a two dimensional skew operation by x and y radians:

$$\begin{bmatrix} 1 & \tan x & 0 \\ \tan y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

PdfTransform includes a property, TransformType that attempts to determine if the transform is one of the primary transformation types. If the transform type can't be determined, the property will be set to PdfTransformType.Other.

To transform a point, use the Transform methods. For example, to rotate a point counterclockwise around the origin, you can do this:

```
PdfPoint p = new  PdfPoint(x, y);
PdfTransform transform = PdfTransform.Rotate(angle);
p = transform.Transform(p);
```

PdfTransform can also combine transformation by using the Concat() method:

```
PdfTransform combined = PdfTransform.Rotate(angle);
PdfTransform translate = PdfTransform.Translate(x, y);
combined.Concat(translate);
```

Note that the Concat operation is not reflexive - a.Concat(b) is not necessarily the same as b.Concat(a).

In PdfDrawingSurface, there is a method called ApplyTransformation() which takes a PdfTransform object and Concats it onto the drawing surface's current transformation. In this way, transforms are cumulative. Applying a transformation will accumulate changes into the drawing surface. To undo a transform, there are two approaches. The first is to apply the inverse transformation:

```
PdfTransform transform = GetTransform();
if (!transform.IsInvertable())
 return;
PdfTransform itransform = transform.GetInverse();

Renderer.DrawingSurface.ApplyTransformation(transform);
...perform drawing operations
Renderer.DrawingSurface.ApplyTransformation(itransform);
```

In order to do this, the specific transform to be applied must have an inverse. In all but degenerate transformations (scale by 0 or a skew that creates a flat line), there will be an inverse that can be applied. Using the IsInvertable() method will tell you if an inverse exists.

The second way to undo a transform is to use the GSave() and GRestore() methods that are part of the PdfPageRenderer objects. GSave() takes the entire drawing state of the PdfPageRenderer and saves it on a stack. GRestore() pops the most recently saved drawing state and restores it. GSave()/GRestore() performs a great deal more work than saving and restoring the current matrix. It will also save line style, clipping, and more. Generally speaking, for working with transformations, it's best to always avoid degenerate transformations and to apply the transform, perform operations and then apply the inverse.

The power of the cumulative approach to transformation is that it is straight forward to encapsulate drawing within another transformation. For example, the entire DrawingList of an existing PdfGeneratedPage could be rendered as a the contents "thumbnail" shape with a dog-eared page by applying a scale transform, doing a GSave(), clipping to the dog eared page boundary, calling the DrawingList's Render() method, doing a GRestore(), stroking the dog-eared page boundary and then undoing the transform.

PdfBaseShape provides indirect access to the transforms by breaking out Translation, Scale, and Rotation into separate properties and concatenating them together before drawing the shape.

When any of the Add or Place methods are used in PdfDrawingSurface, an implicit transform will be applied before the operation and the inverse afterwards. For example, AddRect(PdfBounds r) is implemented in terms of AddRect(r, PdfTransform.Identity()).

PdfGeneratedDocument

For creating or modifying exist PDF documents, use the PdfGeneratedDocument object. Unlike the PdfDocument object, the PdfGeneratedDocument object allows you to directly manipulate the content and details of PDF documents to a much greater depth (and is also more resource intensive). Strictly speaking, PdfGeneratedDocument offers a superset of the features in PdfDocument.

With both PdfGeneratedDocument and PdfDocument, you can rearrange or delete pages, add pages from other documents, rotate pages, set document permissions, create or modify bookmarks, encrypt or decrypt documents, set automatic printing, or create or edit document metadata. With PdfGeneratedDocument, you can replace images in a document, add new pages with new content, add content to existing pages, create or edit annotations, create and edit data collection forms, import SVG artwork, and define high level shapes.

PdfGeneratedDocument can be the cornerstone of a report generation system, a document format converter, a document review system, or a print driver. Since content created within a PdfGeneratedDocument can be serialized and embedded within the output PDF itself, it is easy to create content and read it back for editing.

Pages

The main page class PdfGeneratedPage is a container class that represents a page in a PDF document. It contains a set of PdfBounds objects that are used to describe the page's dimensions

as well as PdfDrawingList object that represents the page's contents. The main dimensions of the page are described with the following:

- Media Box - this is the size of the physical media on which the page is to be printed.
- Crop Box - this is the area to which all content on the page will be cropped when being displayed or printed.
- Bleed Box - this is an area that defines the area that will be used for cropping in a production environment, which may include extra area to accommodate cutting folding and trimming equipment.
- Trim Box - this is the area of that page to be trimmed to in a production environment. It may be smaller than the Media Box to allow for printing instructions, cut marks, color bars or other printer's marks.
- Art Box - this is the area of the page that contains meaningful content intended by the creator.

Each of these areas are measured in PDF units and are subject to PDF's size limitations (3 units (1/24 inch) minimum and 14400 units (200 inches) maximum).

When a new PdfGeneratedPage is constructed only the MediaBox property is set to an area. All other boxes are set to null PdfBounds objects, indicating "not used". In addition, all boxes must be either the same size or within the MediaBox.

Standard page sizes

The object PdfDefaultPages contains a number of static properties that create new PdfGeneratedPages initialized to standard sizes. While it is straightforward enough to create a page with the PdfGeneratedPage constructor and pass in the desired width and height in PDF units, the factory properties in PdfDefaultPages make it easy to work with common standard page sizes such as letter, legal, ledger, A4-A6, B4-B6, and C4-C6. For each default size in portrait layout (the page is thinner than it is tall), there is also a landscape version of the same.

Create stationery

There are a number of ways to create the effect of stationery in the PDF Generating API. Since each PdfGeneratedPage object contains a list of things that are drawn on the page, it can be as simple as prepopulating that list with a few items. Here is a simple example that creates a page that will appear to be a note card.

In this sample, we first make a page that is wide x high in inches. Next we make a background rectangle the same size as the page and add it to the drawing list. Then we make a path that is a single red line a half inch (36 PDF units) down from the top and add it to the page. Finally, we make a disjoint path of blue lines that are evenly spaced by quarter inches down from the red line. Since each line in the path is defined with a separate MoveTo/LineTo pair, the path is disjoint. When the page is returned from this method, there will be three items in the page's drawing list: a rectangle, a red path and a blue path.

```
public PdfGeneratedPage Notecard(double wide, double high, IPdfColor backGroundColor)
{
```

```
 PdfGeneratedPage page = new PdfGeneratedPage(wide * 72, high * 72);
 double top = page.MediaBox.Top;
 double right = page.MediaBox.Right;
 PdfRectangle backGround = new PdfRectangle(page.MediaBox, backGroundColor);
 page.DrawingList.Add(backGround);
 PdfPath redLine = new PdfPath(PdfColorFactory.FromRgb(.75, .16, .45), 0.5);
 redLine.MoveTo(new PdfPoint(0, top - 36));
 redLine.LineTo(new PdfPoint(right, top - 36));
 page.DrawingList.Add(redLine);
 PdfPath blueLines = new PdfPath(PdfColorFactory.FromRgb(.08, .64, .89), 0.5);
 for (double y = top - 36 - 18; y >= 0; y -= 18)
 {
  blueLines.MoveTo(new PdfPoint(0, y));
  blueLines.LineTo(new PdfPoint(right, y));
 }
 page.DrawingList.Add(blueLines);
 return page;
}
```

If you wanted to structurally organize your drawing so that the background of the page was a single layer, you could use a separate layer for background. Although the PDF file format doesn't have strong support for this kind of structural organization, the Atalasoft Generating library gives you the ability to generate with structure if you choose via the PdfDrawingList object. In this way, we could rewrite the note card sample to use a PdfDrawingList for the background:

```

public PdfGeneratedPage Notecard1(double wide, double high, IPdfColor backGroundColor)
{
 PdfGeneratedPage page = new PdfGeneratedPage(wide * 72, high * 72);
 double top = page.MediaBox.Top;
 double right = page.MediaBox.Right;
 PdfDrawingList backLayer = new PdfDrawingList();
 backLayer.Name = "background";
 page.DrawingList.Add(backLayer);
 PdfRectangle backGround = new PdfRectangle(page.MediaBox, backGroundColor);
 backLayer.Add(backGround);
 PdfPath redLine = new PdfPath(PdfColorFactory.FromRgb(.75, .16, .45), 0.5);
 redLine.MoveTo(new PdfPoint(0, top - 36));
 redLine.LineTo(new PdfPoint(right, top - 36));
 backLayer.Add(redLine);
 PdfPath blueLines = new PdfPath(PdfColorFactory.FromRgb(.08, .64, .89), 0.5);
 for (double y = top - 36 - 18; y >= 0; y -= 18)
 {
  blueLines.MoveTo(new PdfPoint(0, y));
  blueLines.LineTo(new PdfPoint(right, y));
 }
 backLayer.Add(blueLines);
 return page;
}
```

Every object that can be in a PdfDrawingList implements the interface IPdfRenderable. One element of that interface is the property "Name" which is a string that names that item. This property is never used by the PDF Generating library. It is intended for client code. In this example, the Name property is used to make the backLayer object easy to identify in later code. For example, if you wanted to create a sense of back-, mid- and foreground layers you could add three PdfDrawingList objects to the page and name them appropriately.

Clipping

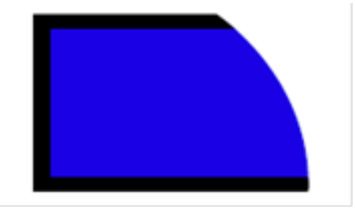
In every PDF page there is always an area that clips drawing to a reduced area. The initial clipping region for any page is the rectangle that defines the page itself. When creating PDF content, it is possible to change that clipping region. Clipping in PDF is different than clipping in GDI. In GDI, any region can be set as the current clipping region. In PDF when you request a new clipping region, the result is the intersection of the current clipping region and the requested one. The net result is that in PDF, it is only possible to reduce the current clipping region or keep it the same. It is, however, possible to save and restore the current clipping region through calls to PdfPageRenderer.GSave() and PdfPageRenderer.GRestore().

In this example, a circle is added to the page as a clipping shape and the rectangle added afterwards will be clipped to the circle.

C# code

```
PdfCircle circle = new  PdfCircle(new  PdfPoint(72, 600),
 100,PdfColorFactory.FromGray(1));
circle.Clip = true;
page.DrawingList.Add(circle);
PdfRectangle rect = new  PdfRectangle(new  PdfBounds(72, 600, 288, 72),
PdfColorFactory.FromGray(0), 6, PdfColorFactory.FromRgb(0.1, 0, .9));
page.DrawingList.Add(rect);
```

This code produces this output.



Since clipping is permanent outside of calls to PdfPageRenderer.GSave() and PdfPageRenderer.GRestore(), there are two IPdfRenderable objects named GSave() and GRestore() which make those calls for you. By modifying the previous sample, the clipping region can be saved and restored:

C# code

```

page.DrawingList.Add(new GSave());
PdfCircle circle = new PdfCircle(new PdfPoint(72, 600), 100,
PdfColorFactory.FromGray(1));
circle.Clip = true;
page.DrawingList.Add(circle);
PdfRectangle rect = new PdfRectangle(
 new PdfBounds(72, 600, 288, 72),
 PdfColorFactory.FromGray(0),
 6,
 PdfColorFactory.FromRgb(0.1, 0, .9));
page.DrawingList.Add(rect);
page.DrawingList.Add(new GRestore());
rect = new PdfRectangle(
```

```
 new PdfBounds(36, 636, 400, 18),
 PdfColorFactory.FromRgb(1, 0, 0));
page.DrawingList.Add(rect);
```

This code produces this output.

As with any filled shape, clipping to a path or shape is done via either the non-zero winding rule or the even odd rule.

Colors

The color model in PDF is very flexible. Colors are associated with a notion of a current color space. Color spaces can include RGB, Gray, CMYK, Lab, and others. Color spaces may also be calibrated or uncalibrated. The Atalasoft PDF Generating library gives you access to colors through a color factory which hides the complexity of the PDF color model. To make a color, use the PdfColorFactory static methods FromRgb, FromColor, FromGray, or FromCmyk. Each of these methods will return a new IPdfColor object that represents the requested color. Color channel values go from 0.0, representing the minimum value, to 1.0, representing the maximum value. Colors may be associated with the name of a PdfColorSpaceResource object. If a color has a resource name, then the color will be a calibrated color, possibly with an associated ICC color profile.

To use RGB colors with an ICC color profile, you can use the resource name "sRgb" as the resource name for your colors. This uses the "standard" RGB ICC color profile which is always available in the color space resources. While there will always be a profile named "sRgb", it is better to use the property DefaultRgbColorSpace as the default resource name. This allows you code to change the name of the default RGB color space resource without changing the calibration of any colors already selected with the previous default.

To add additional color profiles to the resources, you only need a stream, path or the raw data itself. For example, you could use the following C# code to add in a new ICC profile:

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
String csname = doc.Resources.ColorSpaces.AddFromFile("mycolorprofile.icm");
IPdfColor color = PdfColorFactory.FromRgb(1.0, .8, .8, csname);
```

Note that it is up to client code to create colors that are in the appropriate color space for a given resource. In the previous example, if the color profile had been for a CMYK color space, the code requesting an RGB color would be in error and may result in an invalid PDF. In addition to a standard RGB color space, there is also a calibrated gray color space preinstalled. The calibrated gray color space has the resource name "CalGray" and is also accessible using the string property

DefaultGrayColorSpace. While there is a property for a default CMYK color space, there is no default installed. A standard CMYK color profile can be downloaded from Microsoft from the link http://msdn.microsoft.com/en-us/windows/hardware/gg487391.

All color space resources include a property called ColorSpaceType which can be used to find the type of color space represented by the resource.

Rendering

The PdfGeneratedDocument and the PdfGeneratedPage classes are representations of PDF documents and PDF pages, but they are not actual documents or pages. No PDF is created until the document is saved. The process of saving a document to PDF is part of a more general rendering process and in this case, the output of rendering is a PDF document.

The rendering process involves creating an object that is a subclass of the abstract DocumentRenderer class. DocumentRenderer defines the overall process that is used to render a document including firing events, error handling and page rendering. The overall process follows this outline:

1. Notify that the document has begun.
2. Render each page.
    a. Notify that a page has begun.
    b. Construct a PdfPageRenderer object for the page.
    c. Generate the page.
    d. Notify that the page has finished.
3. Notify that the document has finished.

Behind the scenes, the PdfGeneratedDocument.Save() method creates a PdfPageRenderer object and uses it to create the PDF. In most cases, it will not be necessary to use any other means to save a PDF document. The PDF Generating library is robust for creating documents that may have a thousand pages or more without having to worry about memory use. However, in some cases client code may wish to use another mechanism to produce documents. In this case, the client code can construct the PdfPageRenderer directly and use the Render method that takes a PdfGeneratedDocument and an ICollection<BasePage>. In this way client code can use their own collections of pages instead.

Resources

PDF has the notion of document resources. These are objects or chunks of data that may be shared within a page or several pages to reduce the memory needed for the document. There are several classes of resources within PDF. Of them, the Atalasoft PDF library exposes four types: fonts, images, templates and color spaces. In the Atalasoft PDF library, to use a resource, you create it and assign it a name. From then on the resource is referred to by name.

The PdfGeneratedDocument class contains a property, Resources, of type GlobalResources. This object contains properties which represent "managers" for each type of resource. While each

resource manager shares a common base class which contains methods for adding, getting, and querying resources, each manager also contains convenience factory methods specific to each resource type so that making resources is easier.

For example, it might be easier to work with a font by its font name, so The PdfFontManager has a method that will search through installed fonts and attempt to create a font resource based on that.

Font resources

The PDF Generating library supports fonts in PDF via True Type font files. Fonts resources can be created from a font's name (ie, Goudy Old Style Bold), a path to a .ttf or .otf file or a Stream containing the True Type font. Note that .otf files may contain either True Type or Type 1 fonts, but only True Type fonts are accepted.

When creating a font resource, client code assigns the font a name (or accepts an auto-generated one). The actual name is inconsequential and is only used as a unique identifier for the font. Client code should feel free to use any name it wishes. All references to that font will be made through that name and not the resource object.

In version 10.4 and above, there is support for PDF standard Type 1 fonts. In the original version of Acrobat, there were a set of standard fonts that did not need to be embedded within a PDF file and were guaranteed to render accurately. These fonts will be pre-installed in any new GlobalResources object.

The fonts are referred to by their PostScript names:

- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- Helvetica
- Helvetica-Bold
- Helvetica-Oblique
- Helvetica-BoldOblique
- Courier
- Courier-Bold
- Courier-Oblique
- Courier-BoldOblique
- Symbol
- ZapfDingbats

ⓘ Type 1 fonts do not typically have support for more than 255 simultaneously encoded characters. The standard Roman fonts use PDF Standard Encoding, but Symbol and Zapf Dingbats use an Identity encoding scheme where the character value corresponds to the Adobe index of a particular glyph name for the font.

Type 1 symbol font encoding

Unicode character	Character code	Glyph	Unicode character	Character code	Glyph
space	32	space	!	33	!
"	34	#	#	35	#
$	36	#	%	37	%
&	38	&	'	39	#
(	40	(	)	41	)
*	42	#	+	43	+
,	44	,	-	45	#
.	46	.	/	47	/
0	48	0	1	49	1
2	50	2	3	51	3
4	52	4	5	53	5
6	54	6	7	55	7
8	56	8	9	57	9
:	58	:	;	59	;
<	60	<	=	61	=
>	62	>	?	63	?
@	64	#	A	65	A
B	66	B	C	67	X
D	68	#	E	69	E
F	70	Φ	G	71	Γ
H	72	H	I	73	I
J	74	ϑ	K	75	K
L	76	Λ	M	77	M
N	78	N	O	79	O
P	80	Π	Q	81	Θ
R	82	P	S	83	Σ
T	84	T	U	85	Y
V	86	ς	W	87	Ω
X	88	Ξ	Y	89	Ψ
Z	90	Z	[	91	[
\	92	#	]	93	]

Unicode character	Character code	Glyph	Unicode character	Character code	Glyph
^	94	#	_	95	_
`	96		a	97	α
b	98	β	c	99	χ
d	100	δ	e	101	ε
f	102	φ	g	103	γ
h	104	η	i	105	ι
j	106	ϕ	k	107	κ
l	108	λ	m	109	μ
n	110	ν	o	111	o
p	112	π	q	113	θ
r	114	ρ	s	115	σ
t	116	τ	u	117	υ
v	118	ϖ	w	119	ω
x	120	ξ	y	121	ψ
z	122	ζ	{	123	{
|	124	|	}	125	}
~	126	_		127	
	128	Ä		129	Å
	130	Ç		131	É
	132	Ñ		133	Ö
	134	Ü		135	á
	136	à		137	â
	138	ä		139	ã
	140	å		141	ç
	142	é		143	è
	144	ê		145	ë
	146	í		147	ì
	148	î		149	ï
	150	ñ		151	ó
	152	ò		153	ô
	154	ö		155	õ
	156	ú		157	ù
	158	û	DŸD	159	ü

Unicode character	Character code	Glyph	Unicode character	Character code	Glyph
	160	€	¡	161	Y
¢	162	′	£	163	#
¤	164	⁄	¥	165	#
¦	166	ƒ	§	167	#
¨	168	#	©	169	#
ª	170	#	«	171	↔
¬	172	←	•	173	↑
®	174	→	¯	175	↓
°	176	°	±	177	±
²	178	″	³	179	#
´	180	×	µ	181	#
¶	182	#	·	183	•
¸	184	÷	¹	185	#
º	186	#	»	187	#
¼	188	…	½	189	|
¾	190	——	¿	191	↵
À	192	ℵ	Á	193	ℑ
Â	194	ℜ	Ã	195	℘
Ä	196	#	Å	197	#
Æ	198	#	Ç	199	#
È	200	#	É	201	#
Ê	202	#	Ë	203	#
Ì	204	#	Í	205	#
Î	206	#	Ï	207	#
Ð	208	#	Ñ	209	#
Ò	210	®	Ó	211	©
Ô	212	™	Õ	213	#
Ö	214	#	×	215	#
Ø	216	¬	Ù	217	#
Ú	218	#	Û	219	⇔
Ü	220	⇐	Ý	221	⇑
Þ	222	⇒	ß	223	⇓
à	224	#	á	225	〈

Unicode character	Character code	Glyph	Unicode character	Character code	Glyph
â	226	®	ã	227	©
ä	228	™	å	229	#
æ	230	æ	ç	231	ç
è	232	è	é	233	é
ê	234	ê	ë	235	ë
ì	236	ì	í	237	í
î	238	î	ï	239	ï
ð	240		ñ	241	ñ
ò	242	#	ó	243	#
ô	244	ô	õ	245	#
ö	246	ö	÷	247	÷
ø	248	ø	ù	249	ù
ú	250	ú	û	251	û
ü	252	ü	ý	253	ý
þ	254	þ	ÿ	255	

Embed fonts

Standard Type 1 Fonts are not embedded. Allowed True Type fonts are embedded within created PDFs by default. True Type fonts contain information about the contexts in which embedding is permissible.

To embed a font, the PdfFontManager provides the embedding policy for the font. The policy provided looks at the embedding permissions and returns a PdfFontEmbeddingPolicy object containing an action to take.These actions include embed, don't embed, or throw an exception. The default policy provider will embed where allowed and throw an exception when not allowed.

You can also replace the policy provider with a provider that embeds all fonts. Policy providers may also exclude a set of common fonts that are typically on all systems or are known to Acrobat. In this case, when a common font is not present, Acrobat will create a matching "faux font".

Color space resources

PDF allows the use of calibrated colors within documents. This can be done through specific calibrated color spaces or through an ICC Color profile. To handle this the PdfColorSpaceManger object holds a set of color space resources which can be embedded in PDF documents. See the section on Colors for more information.

Image resources

In PDF images resources are stored as a resolution free stream of two dimensional samples. The stream is typically compressed in some manner within the file. The Atalasoft DotImage model for image resource handling to allow the resource manager to accept any object type as an image and then use a set of installed image compressors to determine how to handle that object type. When an image resource is created, all handlers are iterated until one determines that it can handle the object type. That handler then reports a list of possible ways that it can compress the object into a stream suitable for PDF. A compression method is then selected and subsequently applied to the object. For example, if presented with a .NET Bitmap object that is 24 bit RGB, the default handler will report that the image can be compressed using either DCT (JPEG), Flate, or no compression. A compression selector in the PdfImageManager then selects the most appropriate compression to use from that list and then the image is compressed to a stream suitable for PDF.

Image resource streams are kept in a "Stored Stream" object. This object is used to allow a chunk of data to be written out to an appropriate storage device for later retrieval. The default StoreStream type uses the systems temp folder for creating file streams that will be used for storing data. This mechanism can be replaced with other systems if needed by changing the StreamProvider property in the PdfImageManager object. In addition to the default TempStreamProvider, there is a MemoryStreamProvider which is equivalent, but keeps compressed streams in memory. This will be fast, but will clearly place a load on memory used and is therefore not recommended for anything but small images.

The PdfImageManager contains a collection of objects that implement the IPdfImageCompressor interface for compressing images. By default, this will be initialized to contain an instance of the GdiImageCompressor object for handling .NET Bitmap objects.

Compressors are selected by their ability to handle a particular object type. For any given object, a compressor is asked if it can handle the object at a particular "skill." Skills are an indication of the type of work needed to create the actual image data and includes:

- Perfect: The image is handled as is with no changes.
- IncreaseInformation: The image is handled, but the output image will have more information (for example, a compressor might not handle 1-bit perfectly, but instead converts it to 24 bit rgb color).
- DecreaseInformation: The image is handled, but the output image will have less information (for example, a compressor might not handle 48 bit rgb, but reduces it to 24 bit rgb).

For any given image format, there may be a number of different codecs that could be used to compress that image. When an IPdfImageCompressor has been selected, it will return a collection of PdfImageCodec enums that describe how the image will be compressed. Before compressing the image data, the PdfImageManager calls a CompressionSelector with the set of available PdfImageCodecs and returns back a PdfImageCompression object which fully describes all the parameters need to compress the image data. The default CompressionSelector always chooses the first compression in the list.

When an image is compressed and cached, the PdfImageManager uses a IStoredStreamProvider object to provide a way to get at the cache later. The default implementation is the TempFileStreamProvider, which creates a temporary file for the compressed stream for retrieving later. There is also a MemoryStreamProvider that keeps compressed image data in memory. In

most cases, it will not be necessary to change the default selections, but every step is the process is replaceable if need be.

In addition, there is an extra assembly for interacting with Atalasoft DotImage that contains an AtalaImageCompressor object for handling all AtalaImage types.

See Integrate with Atalasoft DotImage for more information.

Template resources

PDF defines a way to create page content that can be reused efficiently. In the PDF specification, these are called Form XObjects, but they are unrelated to the process of data input and collection (Acro Forms). In Atalasoft DotImage, these are called Templates or Drawing Templates. A template resource is a reference to a DrawingTemplate object. A DrawingTemplate object is very similar to a PdfGeneratedPage in that it contains a bounding rectangle which defines a clipping rectangle for the entire DrawingTemplate and a DrawingList which contains the shapes or operations that will mark the page. DrawingTemplate objects themselves can refer to all other resource types.

Shapes

The Atalasoft PDF Generating library includes a hierarchy of high-level shapes. Each shape is meant to fully encapsulate the shape's parameters and be able to draw itself. There are shape objects that represent paths, circles, arcs, rounded rectangles, images, and text. Each of these objects descends from a single class, PdfBaseShape. PdfBaseShape contains the definitions for the shape's color (fill and stroke), the line style used for stroking, and the location, scale and rotation of the object. Shapes that descend from PdfBaseShape typically only have to concern themselves with how they are drawn (how they are filled or stroked) and not with how they are placed on the page (location, scale, rotation). There is no requirement to use any of the PdfBaseShape-derived classes. Each shape is implements at least the PdfRenderable interface and optionally the PdfRenderableContainer and PdfResourceConsumer interfaces. All shapes must be serializable.

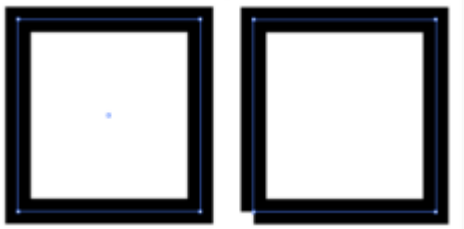
PdfPath

Path shapes are one of the fundamental components of PDF rendering. A path is a list of operations that are performed in sequence to draw the path. There are four operations that can be performed: move, line, curve and close. For example, you could create a square path with the following C# code:

```
private PdfPath Square(double wide, IPdfColor outlineColor, double lineWidth)
{
      PdfPath path = new PdfPath(outlineColor, lineWidth);
      path.MoveTo(0, 0);
      path.LineTo(wide, 0);
      path.LineTo(wide, wide);
      path.LineTo(0, wide);
      path.Close();
      return path;
}
```

The path starts with a move operation and traces the outline of the square. Notice that the square ends with a close operation and not another line. This is because PDF recognizes closed paths and treats them differently. When path is closed, the PDF viewer will automatically connects a straight

line from the last point to the first point and creates a joint to make a clean corner. If you connect the line directly yourself, the PDF viewer doesn't know that it should create a clean corner. The results may not be what you expect. For example, the square on the left was drawn with a close operation. The square on the right was drawn without a close operation.



All the path operations return the PdfPath object itself so you can use a "fluent" style if you choose. The previous path construction could have been written as:

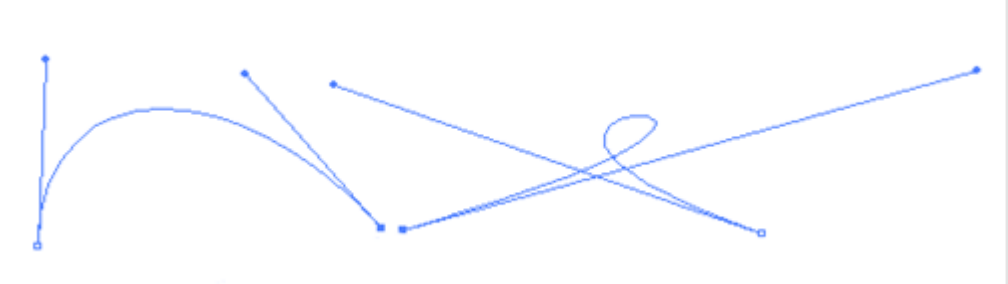
```
return path.MoveTo(0, 0).LineTo(wide, 0).LineTo(wide, wide).LineTo(0,
wide).Close();
```

Curves in PDF are represented by cubic Bézier functions. A Bézier is represented by four points, a start point and an end point (P 0 and P 3) and two control points (P 1 and P 2) and is represented by the following formula:

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 2P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

Where *t* represents time and ranges from 0.0 to 1.0. B(t) represents a point on the curve at time t.

Bézier curves have a number of desirable properties including: a small amount of information (4 points) can represent a wide variety of curves, they can be rendered efficiently, the entire curve will always be contained within a rectangle bounded by the minima and maxima of the four points and the segments P0P1 and P2P3 are tangent to curve at the start and end points respectively.

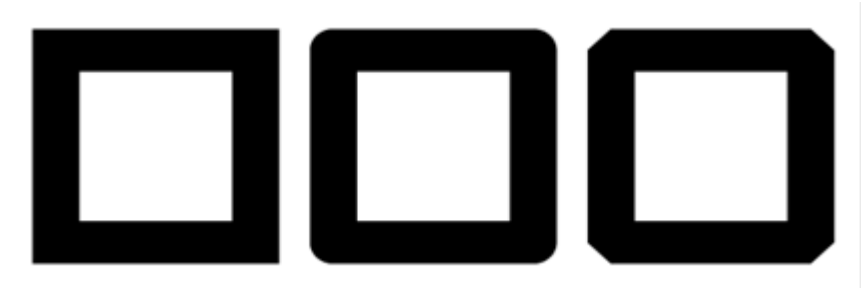


In the PdfPath shape, you can add a curve using the CurveTo method. This method takes three points which represent the two control points and the end point of the curve. The start point of the Bézier will be the last point in the path from any of MoveTo, LineTo or CurveTo methods.

Paths can be filled, stroked or clipped. When a line is stroked, there are a variety options that can be selected for the style of the line, including thickness, joint style, end caps and dashes. These are all available in the LineStyle property of PdfBaseShape.

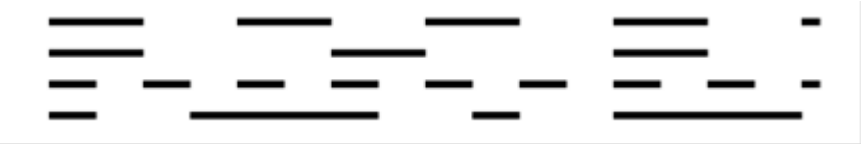
The thickness of a line is in PDF units and defaults to 1.0. When set to 0, the PDF viewer is instructed to render the line in the thinnest possible way. Since this is device-dependent, the final output will not be consistent from device to device and this should be avoided (consider the difference between the thinnest possible line on a 96 dpi monitor versus a 2400 dpi phototypesetter). If a client application wants to create a hairline, it should pick an appropriate thickness instead of 0.

The joint style for a path is how consecutive segments are merged together. There are three possible styles, square, rounded and beveled as shown in these squares.



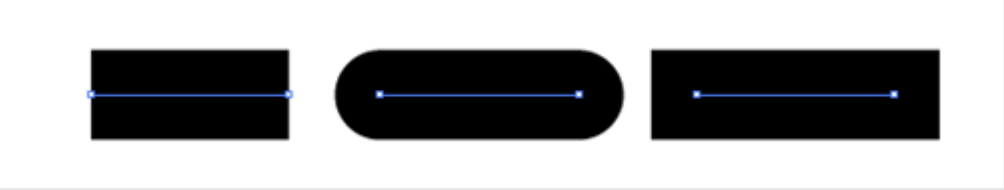
Paths may be stroked in an arbitrary dash pattern. The pattern is a phase number and a collection of alternating dash lengths and gap lengths. The dash and gap lengths are distances along the path in PDF units. The phase is how far into the pattern to start a line. The entire collection of dash and gaps is used until it is exhausted, then it is repeated until the complete path has been stroked.

The following figure shows dash patterns, from top to bottom: [1], [1 2], [0.5], [0.5, 1, 2, 1]

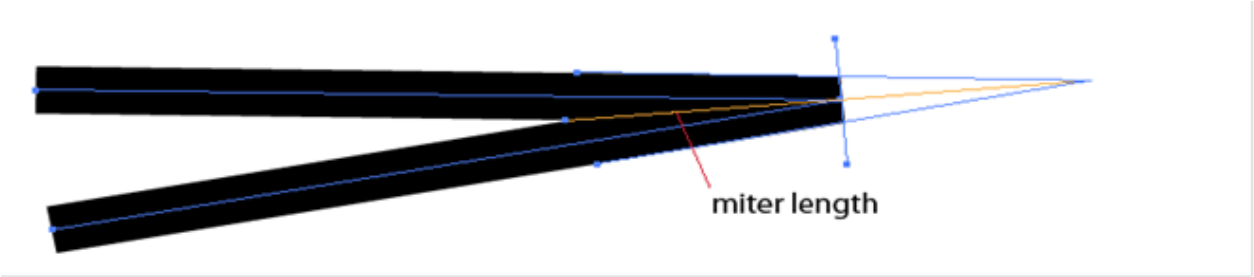


In the sample dash patterns, a single entry implies alternating dash and gaps of the same length. In the bottom example, you can see how complex dashes patterns can be made. Each pattern above has a phase of 0, meaning that the PDF viewer starts the pattern at the beginning. If the phase were 0.5, the first example would have started with a half dash then continued normally.

Paths may be stroked with three different types of ends: butt, round or projecting square.



The final line style is the miter limit. This is a parameter that is used to handle cases when a path with a highly acute angle will project in a reasonable way. In this picture the path is shown with an acute angle and the full miter is project from the line in blue. The miter limit prevents the miter from extending out this distance.



The miter limit is a point at which the mitering will be turned off. It is defined by the ratio of the miter length and the line thickness. When this ratio exceeds the miter limit, mitering will not be done on the line. Since the miter length is related to the angle between the two lines, there is also a relationship between miter limit and line join angle:

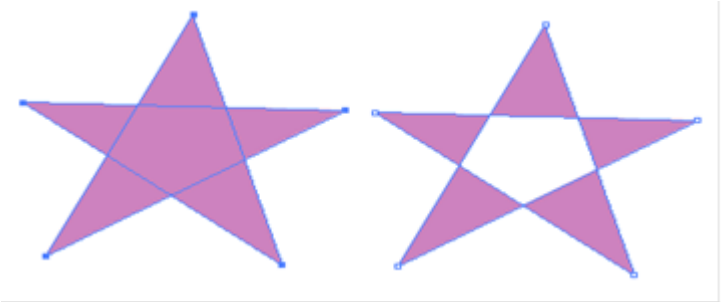
$$\frac{miter\ length}{line\ width} = \frac{1}{\sin\left(\frac{\theta}{2}\right)}$$

Where theta is the angle between the two lines.

A miter limit of 2.0 will cut off miters at angles less than 60 degrees. The default miter limit is 10.

In addition to stroking, paths may be filled with a color. A path may be filled using one of two techniques, either the non-zero winding rule or the even-odd rule. In the non-zero winding rule, horizontal rays are shot through the path. Whenever a path segment crosses the ray going up, one is added to a winding number. Whenever a path segment crosses the ray going down, one is subtracted from the winding number. Whenever the winding number is non-zero, areas along the ray will be filled. In the even odd rule, rays are shot through the path. Whenever the ray has crossed an odd number of path segments, areas along the ray will be filled. The choice of the rule will produce different filled areas in compound paths or paths that self-intersect.

The following figure shows the same shape with the non-zero winding rule (left) and the even-odd rule (right).



PdfRectangle

PdfRectangle is a shape that represents a rectangle. In addition to the properties of PdfBaseShape, it includes a property Bounds, which represents the area of the rectangle. The fill method doesn't affect how a rectangle is filled.

PdfRoundedRectangle

PdfRounded rectangle is a shape that represents a rectangle with rounded corners. In addition to the normal PdfBaseShape properties, PdfRoundedRectangle includes a property Bounds, which represents the area of the rectangle and a property CurveRadius that represents the radius of each corner.

PdfCircle

PdfCircle is a representation of a circle from a center and radius. The circle itself is drawn in PDF using a Bézier path approximation of the circle. By changing the Scale property to a non-uniform scale you can get an ellipse.

PdfArc

PdfArc represents a circular arc. It consists of a the center and radius of a circle as well as the start angle and end angle of the arc in degrees. If the property Clockwise is set to true, the arc will be drawn from the start angle to the end angle in a clockwise direction, otherwise the arc will be drawn counterclockwise. If the property IncludeWedge is set to true, the center will be added to the path drawn.

The following figure shows two PdfArc shapes stroked and filled with IncludeWedge set to false (left) and IncludeWedge set to true (right).

PdfImageShape

PdfImageShape represents an image placed in a rectangular area on the page. It includes a Bounds property representing the area that will be covered with the image and ImageName, the name of an image resource to use to fill the shape. The FillColor, OutlineColor and Clip properties of PdfBaseShape are ignored.

The following C# code creates an image shape from a bitmap.

```
PdfImageShape ConvertBitmapToShape(PdfGeneratedDocument doc, Bitmap bmp)
{
      string imageName = doc.Resources.Images.AddImage(bmp);
      PdfImageShape shape = new PdfImageShape(imageName, new PdfBounds(0,
 0, bmp.Width, bmp.Height));
      bmp.Dispose(); // if you don't need the Bitmap, dispose it
      return shape;
}
```

If you have also purchased the DotImage DocumentImaging toolkit, then you will have access to the classes AtalaImageCompressor and AtalaJpegStreamCompressor in the assembly Atalasoft.dotImage.PdfDoc.Bridge. The AtalaImageCompressor can be added to Images resource manager in a PdfGeneratedDocument's Resources and will handle compressing any object of type AtalaImage. Similarly the AtalaJpegStreamCompressor can be added to the Images resource manager and will handle streams that represent JPEG images. Any stream passed in will, if it is a JPEG image, be copied to the current StoredStreamProvider (default is a temporary file) without recompressing the JPEG data.

To install AtalaJpegStreamCompressor, use the following C# code:

```
doc.Resources.Images.Compressors.Insert(0, new
AtalaJpegStreamCompressor());
```

In addition to the above method to install a new compressor, the AtalaImageCompressor object contains a utility factory method which will construct a new PdfGeneratedDocument with both the AtalaImageCompressor and the AtalaJpegStreamCompressor pre-installed.

To create a document using the factory method, use the following C# code:

```
PdfGeneratedDocument doc = AtalaImageCompressor.CreateDocument();
```

ℹ The CreateDocument() method also has a flavor that accepts instances of the Jpeg2000Encoder and Jb2Encoder objects (or null for none). If you have a license for these objects, you can pass them in and they will automatically be used for color images and 1-bit images respectively.

When the AtalaImageCompressor is installed in a PdfGeneratedDocument, you can pass an AtalaImage directly into the resource manager.

In addition to the AtalaImageCompressor, the bridge assembly also contains a class, AtalaImageCoordinateConverter, which can be used to convert coodinates back and forth between image coordinates and image resolution to PDF coordinates and PDF units.

Remember that images can consume very large amounts of memory. Keeping images in memory will not scale well beyond a few dozen images. If you're working with hundreds of pages with hundreds of images, you should adopt an approach where you create image resources as early as possible and dispose the original images soon thereafter.

You can convert a folder of images to a PDF by using the following C# code:

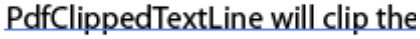
```
public void OneImagePerPage(string inputDirectory, string outputFile)
{
 PdfGeneratedDocument doc = AtalaImageCompressor.CreateDocument();
 FileSystemImageSource images = new FileSystemImageSource(inputDirectory, true);
 while (images.HasMoreImages()) {
  AtalaImage image = images.AcquireNext();
  PdfImageShape shape = AtalaImageCompressor.CreateImageShape(doc.Resources, image);
  PdfGeneratedPage page = doc.AddPage(new PdfGeneratedPage(shape.Bounds.Width,
 shape.Bounds.Height));
  page.DrawingList.Add(shape);
  images.Release(image);
 }
 if (doc.Pages.Count > 0)
  doc.Save(outputFile);
}
```

PDF text shapes

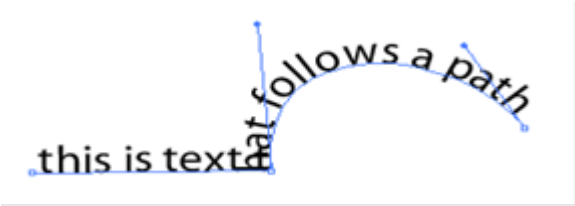
There are six main text shapes available, PdfTextLine, PdfClippedTextLine, PdfTextPath, PdfTextBox, PdfStyledTextBox and DynamicPdfTextBox. Each of the set have different uses and constraints.

PdfTextLine is the simplest of the set. It represents a horizontal line with text on top of it. Text is drawn along the line as people tend to hand write - the bottoms of most letters will be tangent to the line, except for letters with descenders (such as g, p, q, y etc.) which will appear with the descender below the line.

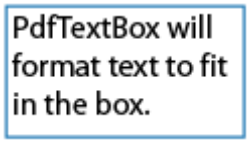
PdfClippedTextLine represents a line of text that will be clipped inside a bounding box on the page. It uses a PdfTextLine shape internally to draw the text.



PdfTextPath is similar to PdfTextLine except that instead of a horizontal line, text will follow any arbitrary set of path operations, including Bézier curves.



PdfTextBox is a shape that draws formatted text on a page. The text will be formatted to fit the bounds using the text properties.



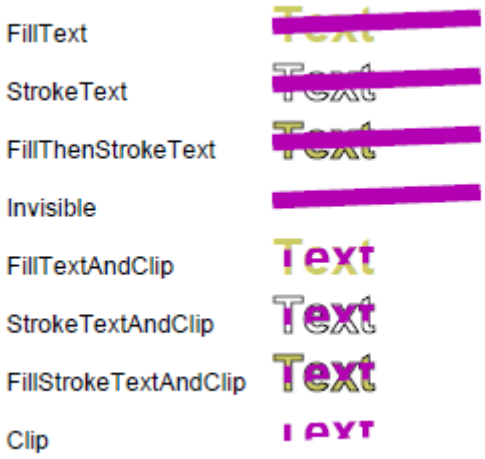
PdfStyledTextBox is similar to PdfTextBox except that it accepts a StyleTextInput object which can be used to add new styled text to the box. Typically this will be used for font changes or color changes.

DynamicPdfTextBox is similar to PdfTextBox except that instead of the text being limited to a fixed box, the DynamicPdfTextBox lets you set a fixed width and it will grow the box up to a maximum.

Each text shape that inherits from PdfBaseTextShape will include the RenderMode property. This is a flags enumeration that allows you to pick one of 8 possible modes of rendering the text which are a combination of filling, stroking, and clipping.

The following C# code provides a demonstration of the RenderMode property.

The code produces the following output.



PdfTable

PdfTable is a conceptual model of a table of text. The table is broken down into a collection of columns. Rows are added to the table to fill out the columns with data. Once the data has been added to the table, call the Fill() method to finalize the content.

Columns are defined by a few properties:

- A key or name for referring to the column
- Text to display as the column header
- The width of the column in PDF units
- The alignment of text in the column
- Left and right padding of the column

Rows can be represented by a Dictionary<string, string> where each key corresponds to a key in the columns. The value associated with that key in the dictionary will be displayed in the row under the column. In addition, rows can be represented by an enumeration of objects that have properties that correspond to the column names.

The following C# creates a simple table.

```
[Serializable]
public class Person
{
 public string Name { get; set; }
 public int Age { get; set; }
 public string Color { get; set; }
}

public void MakeSimpleTable()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
```

```
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
 PdfTable table = new PdfTable(new PdfBounds(72, 300, 400, 400), "Arial", 12);
 table.HeaderFontName = "Arial Bold Italic";
 table.BorderStyle = PdfTableBorderStyle.Grid;
 table.Columns.Add(new PdfTableColumn("Name", "Person", 120, PdfTextAlignment.Center,
 8, 8));
 table.Columns.Add(new PdfTableColumn("Age", "Age", 60, PdfTextAlignment.Center, 8,
 8));
 table.Columns.Add(new PdfTableColumn("Color", "Favorite Color", 0,
 PdfTextAlignment.Center, 8, 8));

 List<Person> people = new List<Person>() {
  new Person() { Name = "John", Age = 15, Color = "Orange" },
  new Person() { Name = "Emily", Age = 37, Color = "Blue" },
  new Person() { Name = "Philippe", Age = 19, Color = "Green" },
  new Person() { Name = "Jill", Age = 23, Color = "Ochre" }
 };

 table.AddRows(people.GetEnumerator());
 table.Fill(doc.Resources.Fonts);

 page.DrawingList.Add(table);
 doc.Save("basictable.pdf");
}
```

Person	*Age*	*Favorite Color*
John	15	Orange
Emily	37	Blue
Philippe	19	Green
Jill	23	Ochre

PdfTemplateShape

The PdfTemplateShape is a very simple shape that is used to place a DrawingTemplate (represented by a Template resource name) on a page. In order to work with a PdfTemplateShape, you need to first create a DrawingTemplate object and add it to your document's Template resources. Then construct a PdfTemplateShape using the resource's name and a desired Bounds on the page. The PdfTemplateShape will be drawn using the all the transformation information in PdfBaseShape (Location, Scale, and Rotation).

ℹ It is easier to make a template shape with coordinates that is based around the origin and Bounds that match the DrawingTemplate's bounds, then use the Location to place it where you want.

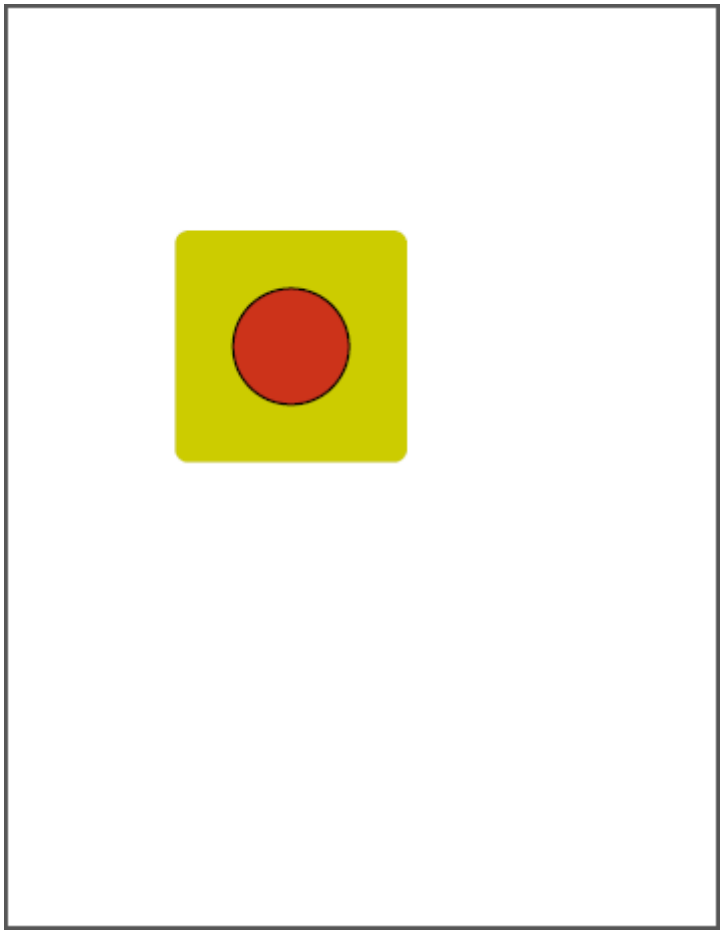
The following C# code makes a simple template.

```
public void SimpleTemplate()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 doc.EmbedGeneratedContent = false;

 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

 DrawingTemplate template = new DrawingTemplate(new PdfBounds(0, 0, 200, 200));
 template.DrawingList.Add(new PdfRoundedRectangle(template.Bounds, 12,
 PdfColorFactory.FromRgb(.8, .8, 0)));
```

```
 template.DrawingList.Add(new PdfCircle(new PdfPoint(template.Bounds.Width / 2,
 template.Bounds.Height / 2),
  template.Bounds.Height / 4, PdfColorFactory.FromRgb(0, 0, 0), 2,
 PdfColorFactory.FromRgb(.8, .2, .1)));

 string resourceName = doc.Resources.Templates.Add(template);

 page.DrawingList.Add(new PdfTemplateShape(resourceName, new PdfBounds(144, 400,
 template.Bounds.Width, template.Bounds.Height)));

 doc.Save("simpletemplate.pdf");
}
```

Note that the DrawingTemplate object has a DrawingList in it that is identical to a PdfGeneratedPage object. As such, you can put any PDF shape (and any IPdfRenderable) object into the your DrawingTemplate.

The output of this example is shown below:



When the DrawingList in a DrawingTemplate is rendered it will be clipped to the DrawingTemplate.Bounds property. Since lines in PDF are centered in width over the mathematical line that defines them, adding a PdfRectangle with a drawn outline that is coincident with the DrawingTemplate.Bounds will result in half of the rectangle's outline being clipped (since it extends beyond the DrawingTemplate.Bounds.

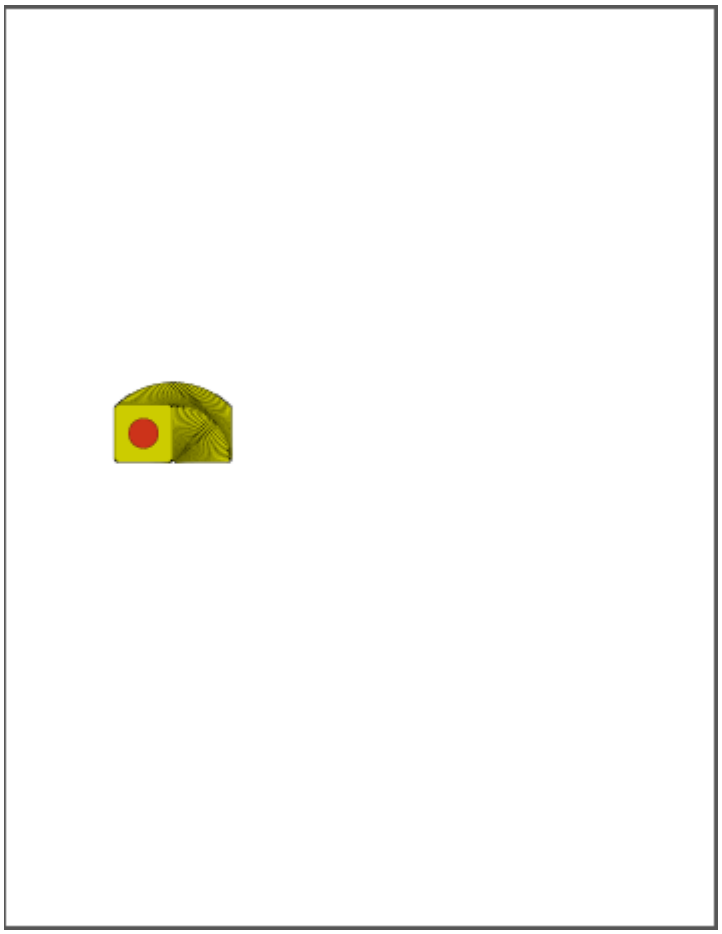
Although DrawingTemplates offer a great deal of flexibility, there are a few artifacts that may be undesirable. All graphic elements will be scaled to the PdfTemplateShape's bounds (and it's Scale). You might wish to make a background box to represent an underlay of a highlighted area and define a single unit-sized DrawingTemplate to represent it it. This will work as expected if the template only uses filled shapes, but if you add any lines, the line width will also be scaled, possibly non-uniformly, producing unpleasant results. In fact, anything with a typically fixed aspect ration (images, text, circles) will get scaled and may look off.

The original intent for DrawingTemplates in PDF was to create letterhead or logos that could be shared from page to page without appreciably increasing the document size.

By modifying the previous sample slightly, we can see how multiple PdfTemplateShapes can be used on a page without altering the original shape.

The following code uses multiple copies of the sample DrawingTemplate.

```
public void SimpleTemplate3()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 doc.EmbedGeneratedContent = false;

 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

 DrawingTemplate template = new DrawingTemplate(new PdfBounds(0, 0, 204, 204));
 template.DrawingList.Add(new PdfRoundedRectangle(new PdfBounds(2, 2,
 template.Bounds.Width - 4, template.Bounds.Height - 4),
  12, PdfColorFactory.FromRgb(.8, .8, 0), PdfColorFactory.FromRgb(0, 0, 0), 4));
 template.DrawingList.Add(new PdfCircle(new PdfPoint(template.Bounds.Width / 2,
 template.Bounds.Height / 2),
  template.Bounds.Height / 4, PdfColorFactory.FromRgb(0, 0, 0), 2,
 PdfColorFactory.FromRgb(.8, .2, .1)));

 string resourceName = doc.Resources.Templates.Add(template);

 PdfTemplateShape shape = page.DrawingList.Add<PdfTemplateShape>(new
 PdfTemplateShape(resourceName, new PdfBounds(0, 0, template.Bounds.Width / 4,
 template.Bounds.Height / 4)));
 shape.Location = new PdfPoint(144, 400);

 for (int i = 1; i <= 30; i++)
 {
  shape = page.DrawingList.Add<PdfTemplateShape>(new PdfTemplateShape(shape));
  shape.Rotation = i * 3;
 }

 doc.Save("simpletemplate3.pdf");
}
```



PostnetBarcodeShape

The PostnetBarcodeShape is an example shape that renders a zip code using a Postnet Barcode. A Posnet bar codeaccepts a text string with either 5, 9, or 11 digits. The bar codeis placed starting at the Location property and moving to the right. Full height bars will be 0.125 inches high and short bars will be 0.05 inches high.

GSave / GRestore

The GSave and GRestore objects are not strictly shapes – they are IPdfRenderable objects that perform graphics state save and restore operations in a DrawingList object.

In PDF (and historically in PostScript), many graphics operations make changes to the current graphic state that aren't changeable. For example, if the clipping area in a PDF page can only be made smaller by clipping operations, not larger. To work around this issue, there are operations in PDF to save and restore the current graphics state. Graphics state includes:

- Stroke Color
- Fill Color

- Transformation matrix
- Font name
- Font size
- Text rendering mode
- Font leading
- Word spacing
- Character horizontal scaling
- Line style (width, dash pattern, line caps, line join, miter limit)
- Clipping
- Current path

Normally, client code will not need these operations as PdfBaseShape is careful to save and restore the current transformation matrix and shapes that clip automatically generate GSave and GRestore operations.

There are cases, where it does make sense. For example, if you need to watermark or otherwise add content on top of existing content an existing PDF document created by software that is not so careful, it will be vital to ensure that the graphics state is predictable. This can be done either by inserting a GSave object in the beginning of the DrawingList and a GRestore object at the end of the list.

The following C# code ensures a clean graphics state in existing content.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument(sourceStream, true);
PdfGeneratedPage page = doc.Pages[0] as PdfGeneratedPage;
if (page == null) throw new Exception("unable to import page 0");
page.DrawingList.Insert(0, new GSave());
page.DrawingList.Add(new GRestore());
// add more content here
doc.Save("output.pdf");
```

Transform

The Transform object is not a shape. It is an object that implements IPdfRenderable. Transform encapsulates a PdfTransform object that will be applied to the PDF content that follows it. Note that transformations are cumulative not commutative. A scale transform applied after a translate transform is rarely the same as a translate transform followed by a scale transform.

Marked content

PDF allows content on a page to contain special markups that define special areas of interest with a name. The meaning of these names are highly specific to the task they represent. For example, the tag "Tx" is used to mark where text operations should fall for rendering an annotation with variable text; the tag "ReversedChars" is usually used for text in a right-to-left reading system that is being rendered by a font that follows left-to-right advancing.

The PdfMarkedContent object encapsulates the PDF marked content markups. It is not a shape itself, but instead contains a DrawingList that will contain content that will be surrounded by marked content markups.

Make custom shapes

To make custom shapes, the easiest approach is to subclass the PdfBaseShape object. Consider the task of making a shape that represents a regular polygon. To make a regular polygon, you need a center, a radius and the number of sides. One way to generate the points is to use get one starting point and rotate it around the center by the angle subtended each side. In creating a new descendant of PdfBaseShape, you need to write a constructor, a clone method and a means to draw the shape:

```
[Serializable]
public class RegularPolygon : PdfBaseShape
{
 public RegularPolygon(PdfPoint center, double radius, int sides) :
  base(PdfColorFactory.FromGray(0.0), 5.0)
 {
  if (sides < 3) throw new ArgumentException("Polygons must have at least 3 sides");
  GeneratePoints(center, radius, sides);
  Center = center;
  Radius = radius;
  Sides = sides;
 }
 public PdfPoint Center { get; private set; }
 public double Radius { get; private set; }
 public int Sides { get; private set; }
 private void GeneratePoints(PdfPoint center, double radius, int sides)
 {
  Points = new List<PdfPoint>();
  PdfPoint currPoint = new PdfPoint(0, radius);
  Points.Add(currPoint + center);
  PdfTransform transform = PdfTransform.Rotate(2 * Math.PI / (double)sides);
  for (int i = 1; i < sides; i++)
  {
   currPoint = transform.Transform(currPoint);
   Points.Add(currPoint + center);
  }
 }
 public List<PdfPoint> Points { get; private set; }
 protected override PdfBaseShape CloneInstance()
 {
  return new RegularPolygon(Center, Radius, Sides);
 }
 protected override void DrawShape(PdfPageRenderer pdfPageRenderer)
 {
  PdfPath path = new PdfPath(this);
  for (int i = 0; i < Points.Count - 1; i++)
  {
   PdfPoint p = Points[i];
   if (i == 0) { path.MoveTo(p); }
   else { path.LineTo(p); }
  }
  path.Close();
  path.Render(pdfPageRenderer);
 }
}
```

In this example, a private list of points is used to hold the points at the corners of the polygon. GeneratePoints() creates a start point at (0, radius) and adds successive rotations of the point to

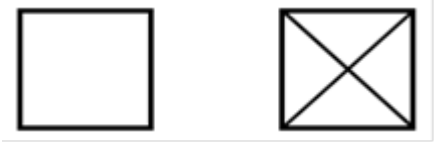
the list. DrawShape is an abstract method defined in PdfBaseShape. Overriding this method lets us draw the polygon as we see fit - in this case we use a PdfPath object to draw the shape for us.

Suppose that you want to create a check box shape. A check box could have a property for its size as well as a property for whether or not it is checked. We could implement this very simply with a PdfBaseShape.

Create a check box with a PdfBaseShape using C#.

```
[Serializable]
public class PdfCheckBoxShape : PdfBaseShape
{
 public PdfCheckBoxShape(double size, bool isChecked, IPdfColor outlineColor, double
 lineWidth)
   : base(outlineColor, lineWidth)
 {
  Size = size;
  IsChecked = isChecked;
 }

 public  double Size { get; set; }

 public  bool IsChecked { get; set; }

 protected  override  PdfBaseShape CloneInstance()
 {
  return  new  PdfCheckBoxShape(Size, IsChecked, OutlineColor, Style.Width);
 }

 protected  override  void DrawShape(PdfPageRenderer w)
 {
  PdfRectangle rect = new  PdfRectangle(new  PdfBounds(0, 0, Size, Size), OutlineColor,
 Style.Width, FillColor);
  rect.Render(w);
  if (IsChecked)
  {
   PdfPath path = new PdfPath(OutlineColor, Style.Width);
   path.MoveTo(new PdfPoint(0, 0));
   path.LineTo(new  PdfPoint(Size, Size));
   path.MoveTo(new  PdfPoint(0, Size));
   path.LineTo(new  PdfPoint(Size, 0));
   path.Render(w);
  }
 }
}
```

When adding these shapes to a PDF, we get something that looks like this:



Or like this when a fill color has been set:

This may be satisfactory for your needs, but what if you didn't want to have a fill color at all and maybe you feel that PdfBaseShape does too much work for you? In either case, you could define your own class from the ground up. All you would need to do is create a class that implements the interface IPdfRenderable, as in this C# code.

```
[Serializable]
public class  PdfSimplestCheckBoxShape : IPdfRenderable
{
 public PdfSimplestCheckBoxShape(double size, bool isChecked, PdfPoint location, double
 lineWidth)
 {
  Size = size;
  IsChecked = isChecked;
  Location = location;
  LineWidth = lineWidth;
 }

 public  double Size { get; set; }
 public  bool IsChecked { get; set; }
 public  PdfPoint Location { get; set; }
 public  double LineWidth { get; set; }

 public  string Name { get; set; }

 public  void Render(PdfPageRenderer w)
 {
  w.DrawingSurface.Begin();
  w.DrawingSurface.AddRect(new  PdfBounds(Location.X, Location.Y, Size, Size));

  if (IsChecked)
  {
   List<PdfPathOperation> path = new  List<PdfPathOperation>();
   path.Add(PdfPathOperation.MoveTo(Location));
   path.Add(PdfPathOperation.LineTo(Location.X + Size, Location.Y + Size));
   path.Add(PdfPathOperation.MoveTo(Location.X, Location.Y + Size));
   path.Add(PdfPathOperation.LineTo(Location.X + Size, Location.Y));
   w.DrawingSurface.AddPath(path);
  }

  PdfLineStyle style = PdfLineStyle.Default;
  style.Width = LineWidth;
  w.DrawingSurface.Stroke(style, PdfColorFactory.FromGray(0));
  w.DrawingSurface.End();
 }
}
```

In this case, the infrastructure of PdfBaseShape is gone, so we have to implement the method Render(). This method is give an object called PdfPageRenderer which is responsible for creating content that will go into the pages content. This object itself is an abstraction of the PDF rendering model and provides a number of operations that make is easy to create correct PDF content. Within the PdfPageRenderer object, there is a property called DrawingSurface. The DrawingSurface is a virtual canvas for performing drawing operations, including paths, rectangles, templates, and images. To draw shapes, you add path elements (paths or rectangles) then either stroke or fill them.

Before performing any drawing operations, you *must* call the Begin() method and after you are done, you *must* call the End() method. Begin() and End() calls may be nested to any depth.

Note the following:

- Whether you are subclassing PdfBaseShape or implementing IPdfRenderable, you should make your object serializable. When document content is embedded within a PDF document, the elements of drawing lists will be serialized into the final PDF. If any element is not serializable, this will cause a failure during a Save when the PdfGeneratedDocument property EmbedGeneratedContent is true.
- If you are implementing a shape that uses document resources (fonts, colorspaces, templates, images, etc.) or contains an object that implements IPdfResourceConsumer, you must implement the interface IPdfResourceConsumer. This interface allows an object to report the resources it uses as well as rename them if needed. In implementing ResourcesUsed and NotifyResourceRenamed, if you refer to Template resources or any other object that implements IPdfResourceConsumer, you must also find and return the resources consumed by them.
- If you are implementing a shape that contains text, consider implementing the interface IPdfTextContainer which will allow a standard way of setting and getting text from a shape.
- If you are implementing a shape that may contain sub-shapes, consider making a property of type PdfDrawingList and implementing IEnumerable<IPdfRenderable> and returning the PdfDrawingList's GetEnumerator(). This will ensure that child enumeration happens in a predictable manner.

Round trip documents

PDF documents can be created with a number of different tools and the process or toolset used in their creation determines the actual PDF data content, which in turn may bear little or no resemblance to the original data structures. As such, PDF is often considered to be a write-only or final format. The Atalasoft PDF Generating toolkit provides some means around this limitation. If you create a PDF from a PdfGeneratedDocument object and set the EmbedGeneratedContent property to *true*, then after the PDF content has been rendered, the DrawingList object in the PdfGeneratedPage will be serialized and embedded in the PDF so that it can be retrieved later and rebuilt.

In other words, you can get full round-trip editing of PDFs by embedding your Generated content within the PDF itself. This also means that shape objects like PdfCircle which generate Bezier curves in the final PDF will come back as PdfCircle objects and not as a PdfPath object.

Embedding the Generated content adds a moderate amount of overhead to the final PDF, but resource objects do not count in this overhead as these resources will get rebuilt from the PDF content itself.

The Atalasoft PDF Generating toolkit also includes the ability to import pages from the Atalasoft PdfDocument object. For example, you can dynamically insert a cover page into an existing document or easily pull in a page, say a legal disclaimer, from an existing PDF. PdfPage objects from the Pages property of PdfDocument also inherit from the BasePage object and can therefore go into the Pages collection of a PdfGeneratedDocument.

PdfPage objects from PdfDocument objects are very light-weight in comparison to PdfGeneratedPage objects as they only reference the original page instead of containing a representation of data within the page (size, rotation, annotations, scripts, etc.).

Integrate with DotImage

In addition to the main assembly, there is an additional assembly, Atalasoft.PdfDoc.Bridge. This assembly provides a bridge between Atalasoft DotImage classes and the PDF Generating classes. The main class is the AtalaImageCompressor. To use this class, make an instance of it and add it to the Compressors collection using the following code.

```
PdfGeneratedDocument doc = new  PdfGeneratedDocument();
doc.Resources.Images.Compressors.Insert(0, new  AtalaImageCompressor());
```

This will provide tools that will allow the PdfImageManager method FromImage to accept AtalaImage objects. All pixel formats are accepted by the AtalaImageCompressor. In addition, if the AtalaImageCompressor object is constructed with instances of the Atalasoft Jp2Encoder and Jb2Encoder objects, then images can be compressed using JPX and JBIG2 encoding.

There is also another image compressor, the AtalaJpegStreamCompressor. This compressor accepts a .NET stream object and if the stream contains a JPEG image, it will create an image resource with the already compressed stream and will not degrade the image by decoding and re-encoding it.

To make this process easier, AtalaImageCompressor has a static factory method called CreateDocument which will create a new, empty PdfGeneratedDocument object with the AtalaImageCompressor and AtalaJpegStreamCompressor preinstalled.

C#

```
PdfGeneratedDocument doc = AtalaImageCompressor.CreateDocument();
PdfGeneratedDocument  doc1 = AtalaImageCompressor.CreateDocument(new Jp2Encoder(),
 null);

string imName = doc.Resources.Images.AddImage(atalaImage);
string imName1 = doc1.Resources.Images.AddImage(atalaImage);
```

In this example, doc1 is created with the Atalasoft Jp2Encoder which will provide JPX compression, if it is available.

Since AtalaImage objects may contain calibrated color profiles through the ColorProfile property, it is advantageous to pass this on to the generated PDF. This can be done manually, by creating a PdfColorSpace resource through the PdfColorSpaceManager, but it can be done automatically via the static method AddImageResource in the AtalaImageCompressor:

C#

```
AtalaImage image = new AtalaImage(200, 200,
PixelFormat.Pixel24bppBgr);
image.ColorProfile = ColorProfile.FromSrgb();
string[] names = AtalaImageCompressor.AddImageResource(doc.Resources, image);
```

In this example, AddImageResource will first see if the image has a non-null ColorProfile and if so it will create a PdfColorSpaceResource for that ColorProfile and will then make a PdfImageResource

for the AtalaImage using the created PdfColorSpaceResource. The method returns an array of two strings. The first string is the name of the image resource and the second will be the name of the color space resource or null if there was no color profile.

When working with PdfImageShape objects, it is necessary to size the resulting object to PDF dimensions. This can be done automatically by using the static methods ImageSize and ImageSizeAt in AtalaImageCompressor. Given an AtalaImage object, these methods return a PdfBounds object that is sized in PDF units to match the image's real-world dimensions as specified by the Width, Height, and Resolution property of the image. If the units are not specified in the resolution, they will be treated as if they were pixels per inch.

Finally, there are a pair of utility methods in AtalaImageCompressor to make PdfImageShapes as automatically as possible. They are called CreateImageShape() and CreateImageShapeAt(). Both are passed the PdfGeneratedDocument Resources property and the source AtalaImage and return a new PdfImageShape object representing that image. CreateImageShapeAt() also takes an x and y in PDF coordinates specifying location of the lower left corner of the image. Note that once a PdfImageResource or PdfImageShape object has been created from an AtalaImage, the source image is no longer necessary and may be disposed freely. The PdfImageShape object and the PdfImageResource are themselves very lightweight when compared with the original AtalaImage as the actual image data will have been written out to a temporary stream on resource creation and is kept out of memory entirely - even at the point of calling PdfGeneratedDocument, the data is streamed across from the temporary stream to the final PDF and never stays in memory beyond buffering.

Actions

PDF defines a set of actions that can be performed in response to user interaction on a page or in response to other events that happen at a page or document level. In general, anything that cause or respond to an event usually has a suite of actions associated with it. For example, any PDF document may contain a list of bookmarks and instead of having each bookmark be simply associated with a location within the document, they are instead an action list of actions to take, one of which is likely to be a "go to view" action.

Actions may be put in a number of places within a PdfGeneratedDocument including:

- PdfGeneratedDocument.AdditionalActions: A set of actions that are triggered by document-level events.
- PdfGeneratedDocument.GlobalJavaScriptActions: A set of JavaScript-only actions that are performed when a document has been opened. This is intended to be used to define global functions to be shared across all JavaScript actions in the entire document.
- PdfGeneratedPage.AdditionalActions: A set of actions that are triggered by page-level events.
- BaseAnnotation.AdditionalActions: A set of actions that are triggered by annotation events. Even though the PDF spec allows for these to exist in all annotation types, they appear to only be honored by Adobe Acrobat with BaseWidgetAnnotation objects.
- BaseAnnotation.ClickActions: A set of actions that are triggered when an annotation has been clicked.

PdfAction

PdfAction is an abstract base class from which all actions inherit. It has a single property in it, ActionType, which is an enumeration that indicates the type of the action. These are the possible values of ActionType:

- GoToView - Go to a specific page and location in the document
- GoToRemote - Go to a page and location in a remote document
- GoToEmbedded - Go to a page and location in an embedded file
- LaunchApplication - Launch an application
- ReadThread - Start reading at a threaded point
- FollowURI - Resolve a uniform resource identifier
- PlaySound - Play a sound
- PlayMovie - Play a movie
- Hide - Set an annotation's hidden flag
- PerformNamedAction - Perform a set of actions associated with a name
- SubmitForm - Submit form data to a URI
- ResetForm - Reset form data to defaults
- ImportData - Import form data from a file
- JavaScript - Execute a JavaScript script
- SetOCGState - Set the state of optional content groups
- Rendition - Control how multimedia is played
- PerformTransition - Perform a transition
- GoTo3DView - For to a view in a 3D model.

Not all types are presently supported. Those that are not supported will have the correct ActionType, but will be represented as a PdfUnknownAction.

Go To View actions

The most common type of PdfAction is a PdfGoToView actions. A PdfGoToView action is very simple - it contains a Destination property that defines the location to where the viewer should navigate when the action is executed. The destination is an object of type Destination which contains information about which page will be visited and how to zoom on that page. While it is straight-forward to make a PdfDestionation object and construct a PdfGoToView action which contains it, there are factory methods within PdfDestination that make both PdfDestination objects or a PdfGoToViewAction containing the appropriate PdfDestination object with this C# code.

```
PdfAction action = PdfDestination.FitPageAction(targetPageIndex);
```

This will go to the 0-based page specified by targetPageIndex and display the page so that the entire page fits within the viewer window.

ⓘ If you reorder pages within a document, it will be necessary to modify actions within the document that point to that page.

PdfDestination has factory methods for making the following PdfGoToViewActions:

- PointZoomAction
- FitPageAction
- FitWidthAction
- FitHeightAction
- FitRectangleAction
- FitBoundsAction
- FitBoundsWidthAction

URI actions

The PdfURIAction object represents a URI with an optional Base URI that represents a target for a link. When activated, a typical viewer will request permission from the user to follow the URI specified. There is also an optional parameter to allow the area (if any) represented by a link to act as a mapped link. The coordinates of the click relative to the link area will be appended to the URI in the form ?<x-coordinate>,<y-coordinate>.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
doc.BookmarkTree = new PdfBookmarkTree();
doc.BookmarkTree.Bookmarks.Add(new PdfBookmark("Atalasoft", Color.Blue,
 FontStyle.Regular,
 new PdfURIAction(new Uri("http://www.atalasoft.com")), true));
doc.Save("uriaction.pdf");
```

JavaScript actions

PDF has the ability to define actions that execute JavaScript code when activated. The specifics for what can be done with JavaScript actions is extensive. Please refer to the Adobe documentation for the proper use of JavaScript action. It should be noted that the JavaScript within the actions is not checked for syntactic or semantic correctness.

The following C# code makes a document self-printing.

```
PdfJavaScriptAction selfPrint = new
 PdfJavaScriptAction("this.print({bUI:true,bSilent:false,bShrinkToFit:true});");
document.GlobalJavaScriptActions.Add("MySelfPrint", selfPring);
```

Sound actions

PDF has the ability to play sounds to actions. This can allow you to add audible feedback when buttons are pressed or links activated. Sounds to be played by PdfSoundAction objects can be specified using the Sound object. Within a PdfSoundAction, you can specify the volume of the sound, if it will be played synchronously, if it should repeat and if it should mixed with already playing sounds.

ⓘ Acrobat version 5.0 and earlier does not support the MixWithPlayingSounds property and Acrobat 6.0 does not correctly support the IsSynchronous property.

To make a sound action, the first step is to create a Sound object. That can be done with a WavReader, which determines the sound characteristics (sampling rate, bits per sample, etc) and

populates a Sound object. The PdfSoundAction object refers to the sound that will be played. This way multiple actions can refer to the same sound.

In this sample C# code, a document plays a sound when opened.

```
public void SoundActionOnOpened()
{
 using (FileStream stm = new FileStream(ImageUtilities.ImageDatabase + @"\PDF
\Multimedia\Sound\boing.wav", FileMode.Open, FileAccess.Read, FileShare.Read))
 {
  PdfGeneratedDocument doc = new PdfGeneratedDocument();
  PdfGeneratedPage page1 = doc.AddPage(PdfDefaultPages.Letter);

  WavReader reader = new WavReader(stm);
  Sound sound = Sound.FromWavReader(reader);
  PdfSoundAction soundAction = new PdfSoundAction(sound);
  doc.AdditionalActions.OnDocumentOpened.Add(soundAction);
  doc.Save("soundonopened.pdf");
 }
}
```

Show/Hide action

The PdfShowHideAction is used to make sets of annotations or form fields visible or invisible. It does this by setting the Hidden property within an annotation or field. The action can show or hide an arbitrary number of fields or annotations using a set of PdfAnnotationIdentifier objects. Each PdfAnnotationIdentifier either refers to an annotation by the index of the page and the index of the annotation within the page's collection or by FieldFullName (if the annotation is a form field).

ℹ Generally speaking, it is more convenient to use the FieldFullName for widget annotations instead of the page index/annotation index pair as it is immune to the annotation getting moved from page to page or having its order on the page changed. If the annotation is a widget annotation and is the child of a FormField, be sure to set the FieldName and ParentField properties of the widget annotation to ensure that FieldFullName is correct. If the ParentField is not properly set, DotPdf will set it for you on save, but this will cause the FieldFullName to change.

The following C# code shows and hides an annotation.

```
public void ShowHideAction()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 doc.Form = new PdfForm();
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

 PushButtonWidgetAnnotation toHide = new PushButtonWidgetAnnotation(new PdfBounds(72,
 600, 200, 36), "Hide Me", null, null);
 page.Annotations.Add(toHide);
 doc.Form.Fields.Add(toHide);

 PushButtonWidgetAnnotation willHide = new PushButtonWidgetAnnotation(new PdfBounds(72,
 650, 200, 36), "Hide", null, null);
 willHide.AdditionalActions.OnClickUp.Add(new PdfShowHideAction(true, new
 PdfAnnotationIdentifier(toHide.FieldFullName)));
 doc.Form.Fields.Add(willHide);
 page.Annotations.Add(willHide);

 PushButtonWidgetAnnotation willShow = new PushButtonWidgetAnnotation(new
 PdfBounds(300, 650, 200, 36), "Show", null, null);
```

```
 willShow.AdditionalActions.OnClickUp.Add(new PdfShowHideAction(false, new
 PdfAnnotationIdentifier(toHide.FieldFullName)));
 doc.Form.Fields.Add(willShow);
 page.Annotations.Add(willShow);

 doc.Save("annothideshow.pdf");
}
```

This creates a one-page document that has three button annotations. The first button is hidden when the button named "Hide" is pressed and is shown when the button named "Show" is pressed.

Named actions

PDF defines a type of action called a Named action which includes the name of a particular navigation action to take. These actions are ways for changing the current page being viewed. They are more convenient for coding than PdfGoToView actions in that PdfGoToView actions always need an absolute page number, whereas named actions are always relative to your current page.

Available names are:

- NextPage
- PrevPage
- FirstPage
- LastPage

ℹ The PDF specification allows nearly any arbitrary name for the action, but viewers are only responsible for responding to the four standard names. Viewers will typically ignore anything beyond the standard names. You can use the static method PdfNamedAction.IsStandardName to determine if a name is standard or not.

The following C# code adds navigation buttons to a page.

```
public void AddNavigationButtons(PdfGeneratedPage page, int pageIndex)
{
 string[] labels = new string[] { "|<", "<", ">", ">|" };
 string[] names = new string[] { "FirstPage", "PrevPage", "NextPage", "LastPage" };

 for (int i = 0; i < labels.Length; i++)
 {
  PdfBounds bounds = new PdfBounds(36 + 40 * i, page.MediaBox.Top - 40, 36, 36);
  PushButtonWidgetAnnotation button = new PushButtonWidgetAnnotation(bounds,
 String.Format("p{0}b{1}", pageIndex, i), null, null);
  // The FieldName must be unique, but the Name need not be.
  button.Name = labels[i];
  button.AdditionalActions.OnClickUp.Add(new PdfNamedAction(names[i]));
  page.Annotations.Add(button);
 }
}

public void NavigationButtons()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 string fontResName = doc.Resources.Fonts.AddFromFontName("Arial Black");
 for (int i = 0; i < 4; i++)
 {
  PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
```

```
  page.DrawingList.Add(new PdfTextLine(fontResName, 300, String.Format("{0}", i + 1),
 new PdfPoint(200, 400)));
  AddNavigationButtons(page, i);
 }
 doc.Save("navbuttons.pdf");
}
```

Submit Form Actions

The PdfSubmitFormAction is an action that will cause data within the form of the current PDF to be submitted to a remote client. The action has a number of flags that control what data will be submitted and the format of the submission. Formats include FDF, XFDF, HTML, and PDF. The action also has a property called Fields which can be used to exclude or include any particular field within the document.

ℹ Like PdfShowHideAction, the fields in PdfSubmitFormAction are referenced with a PdfAnnotationIdentifier. Using PdfAnnotationIdentifier with a form full name will be more reliable to changes than page/annoation indexes.

Reset Form Action

The PdfResetFormAction is an action that will cause fields within the form of the current PDF to be reset to their default value. Most fields have a DefaultValue property that will be used for the field. The action also has a property called Fields which can be used to exclude or include any particular field or fields within the document in the reset.

ℹ Like PdfShowHideAction, the fields in PdfResetFormAction are referenced with a PdfAnnotationIdentifier. Using PdfAnnotationIdentifier with a form full name will be more reliable to changes than page/annoation indexes.

Annotations

PDF comes with a rich set of annotations and the means of representing the annotation on the page and controlling the interactions with the user. Annotations can be anything from simple marks on the page to a complex set of appearances with attendant complex behaviors. Most annotations in PDF are supplied with a default appearance by the viewer, but using drawing template resources, it's easy to make annotations appear as you wish.

Each PdfGeneratedPage object contains a property called Annotations, which is a collection of all annotations on the page. Annotations are located on the page with a Bounds property that defines the location and dimensions of the annotation. The location and orientation by default follows the page orientation unless it is a sticky note/popup or if the NoRotate property is set to true.

Annotations fall into three broad categories:

General annotations

- LinkAnnotation
- OpaqueAnnotation
- PopupAnnotation

- SoundAnnotation

Mark up annotations

- CaretAnnotion
- CalloutAnnotation
- EllipseAnnotation
- LineAnnotation
- PolygonAnnotation
- PolylineAnnotation
- RectangleAnnotation
- RedactionProposalAnnotation
- RubberStampAnnotation
- StickyNoteAnnotation
- TextBoxAnnotation
- TextMarkupAnnotation
- TypeWriterAnnotation

Widget annotations

- CheckboxWidgetAnnotation
- ChoiceWidgetAnnotation
- PushButtonWidgetAnnotation
- RadioButtonWidgetAnnotation
- SignatureWidgetAnnotation
- TextWidgetAnnotation

Mark up annotations are annotation types that are used to describe annotations that are used for document mark up or review. Widget annotations are used to define form fields for data collection or user interaction. General annotations are all else.

All annotations will inherit from the abstract class BaseAnnotation. All markup annotations will inherit from BaseMarkupAnnotation. All widget annotations inherit from BaseWidgetAnnotation.

Properties common to all annotations

All annotation inherit from the class BaseAnnotation. BaseAnnotation defines a set of properties that are common to all annotation types. While all annotations have these properties, not all annotations use them, or use them in the same way.

Property name	Property type	Description
AdditionalActions	AnnotationAdditionalActions	A collection of annotation events by name with an associated collection of actions to take when that event happens. These are usually reserved for widget annotations, but the PDF specification demands that they are available in all annotation types whether or not they are meaningful.
AnnotationType	string	Gets the original type of the annotation if read from a PDF file, else empty string.
Appearance	AppearanceSet	A collection of appearances to be used for this annotation.
Border	AnnotationBorder	For simple annotation types (circle, rectangle, polygon), sets the corner radii (if applicable), line width, and line dash pattern. It is generally easier to control the actual appearance of a custom annotation by creating an appearance.
BorderStyle	BorderStyle	For any annotation with a border, define the line style of the annotation. It is generally easier to control the actual appearance of a custom annotation by creating an appearance.
Bounds (Required)	PdfBounds	Gets or sets the boundary rectangle for this annotations. This rectangle is in page coordinate and PDF standard units. The Bounds will be oriented relative to the page and its Rotation unless NoRotate is set to true. (Required)
ClickActions	PdfActionList	A set of actions performed when the annotation has been clicked.
Color	IPdfColor	Gets or sets the dominant color for the annotation. The interpretation of Color depends on the annotation. It may represent the color of the annotations icon (if any) or the border of the annotation.

Property name	Property type	Description
Contents	string	Represents the text of the annotation. Its interpretation depends on the annotation type. For sticky note annotations, it will be the contents of the note.
DefaultAppearanceState	string	Represents the initial/default state of an annotation. When an annotation is "Normal" (no interaction), the appearance that will be used for the annotation will be Appearance.Normal[DefaultAppearanceState].
Hidden	bool	If true, the annotation will neither be visible nor will it print.
Invisible	bool	If true, if the annotation type is not recognized by the viewer, it will not be displayed, otherwise the viewer will try to make a substitute appearance.
IsParentRequired (Required)	bool	If true, this annotation type requires the Parent property to be set. (Required)
IsTransparent	bool	If set to true, indicates that the Color property will be ignored. This does not indicate opacity.
Locked	bool	If set to true, indicates that the annotation may not be selected or moved (although its Contents may be editable).
ModificationDate	DateTime	Gets or sets the modification date of the annotation. DotPdf does not track or modify this property.
Name	string	Gets or sets the name of the annotation. This string is an identifier that is typically used for JavaScript actions to locate a particular annotation. It should be unique for annotations on a given page. If there are annotations with duplicate names, DotPdf will make the names unique if necessary on save.
NoRotate	bool	If set to true, the annotation will not be rotated with the page rotation.

Property name	Property type	Description
NoView	bool	If set to true, the annotation will not be visible and will not interact with the user, but it will be printed. This is one way of making a print-only watermark on a page.
NoZoom	bool	If set to true, the annotation will not zoom with the viewer but instead will be displayed in its native size.
ParentPage (Sometimes required)	PdfGeneratedPage	Gets or sets the page on which the annotation is attached. This property is encouraged but is only required on ScreenAnnotations.
Print	bool	If set to true, indicates that the annotation should be printed with the document.
ReadOnly	bool	If set to true, the annotation will not interact with the user.
ToggleNoView	bool	If set to true, indicates that when the mouse enters the annotation, the NoView property should be toggled.

Properties common to all mark up annotations

BaseMarkupAnnotation defines a set of properties that are common to all annotation types. While all mark up annotations have these properties, not all mark up annotations use them, or use them in the same way.

Property names	Property type	Description
AuthorName	string	Gets or sets the author of the annotation. Conventionally, this will be set to the current username or the full name of the user who is making the annotation.
CreationDate (Required, automatic)	DateTime	Gets the date and time when the annotation was created. This value is set automatically by the constructor of BaseMarkupAnnotation to the current time. (Required)
InReplyTo	BaseAnnotation	Null unless the annotation is meant to be a reply to another existing annotation.
InReplyToRelation	ReplyRelation	Describes the relationship of a reply annotation. Not required, but only meaningful if InReplyTo is set.

Property names	Property type	Description
Intent (Required, automatic)	AnnotationIntent	Describes the intent of the annotation. When required, this is set by individual classes.
Popup	PopupAnnotation	Gets or sets an annotation to be displayed as a Popup to a markup annotation. In the original version of Acrobat, a sticky note was the only annotation type with a pop-up text window and was a special case. In later versions, the ability to add pop-up information to an annotation was added to all mark up annotations.
RichTextContent	XmlDocument	RichTextContent is an XML representation of marked up text for display. It allows the body, p, i, b, and span tags. If you set the RichTextContent property, be sure to set the Content property to the plain text equivalent.
Transparency	double	Gets or sets the overall transparency of the annotation. A value of 1.0 means fully transparent and a value of 0.0 means fully opaque.

Properties common to all widget annotations

BaseWidgetAnnotation defines a set of properties that are common to all widget annotation types. While all widget annotations have these properties, not all annotations use them, or use them in the same way.

Property name	Property type	Description
BackgroundColor	IPdfColor	Gets or sets the color of the background.
BorderColor	IPdfColor	Gets or sets the color of the border.
ChildFields	IList<IFormElement>	Null
DefaultTextAppearance	PdfTextAppearance	Gets or sets the default appearance of text in the annotation.
DefaultValueAsString	string	Gets the default value of the annotation as a string.
FieldAlternateDescription	string	A string used to describe the field for use in display in a user interface. This typically gets displayed in a tooltip.

Property name	Property type	Description
FieldFullName	string	Returns the full name of the field. This is created by starting with the parent-most field's FieldNameForExport (or FieldName if FieldNameForExport is null), descending down to the annotation and separating them with '.' characters (ex: Address.Street.Number). It is the user's responsibility to ensure that if a widget annotation is a child of another field that its ParentField is set.
FieldName (Required)	string	Gets or sets the field's name. This name is used for submitting form information (unless FieldNameForExport is set) and display in the user interface. The FieldName should be selected so that the FieldFullName will be unique. (Required)
FieldNameForExport	string	Gets or set a field name that will be used for data export. The FieldNameForExport, if present, will be used instead of FieldName. It should therefore be chosen so that FieldFullName is unique.
HighlightAppearance	WidgetHighlightAppearance	Gets or sets how the widget will appear when it receives a mouse down event.
IsFieldNoExport	boolean	If set to true, this field will not be exported.
IsFieldReadOnly	boolean	If set to true, this field cannot be edited.
IsFieldRequired	boolean	If set to true, this field must be set by the user.
ParentField	IFormElement	This property should represent the parent field of this widget (if any). Widget annotations may not be the parent of any other form element.
ValueAsString	string	Returns the value of the form element as a string.

General annotations

General annotations are annotations that don't really fit into any other category. These include:

- LinkAnnotation

- OpaqueAnnotation
- PopupAnnotation
- SoundAnnotation

LinkAnnotation

In the original version of Acrobat, a link annotation was a set of regions bound to a destination within the document. When actions were added to the PDF specification, link annotations were changed to be a set of regions that included a ClickAction that described what should happen when the link was clicked.

The regions are defined by a set of PdfQuadrilateral objects. This intended so that you can delimit a set of words that are not axis aligned and they will highlight correctly. If the Regions is empty, the Bounds will be used as the link area. If the Regions is not empty, the Bounds will be automatically expanded to contain all the quadrilaterals.

The LinkAnnotation object comes with a number of convenience constructors for making simple URI links or single click actions.

Property name	Property type	Description
HighlightAppearance	LinkHighlightAppearance	Gets or sets how the link will appear when it is clicked. Can be one of None, Invert, Outline, and PushDown
Regions	PdfQuadrilateralCollection	A set of quadrilateral regions that define the annotation.

The following C# code creates a simple link annotation.

```
LinkAnnotation annot = new LinkAnnotation(new PdfBounds(72, 500, 72, 72),
new PdfURIAction(new Uri("http://www.atalasoft.com")));
```

OpaqueAnnotation

An OpaqueAnnotation represents an annotation type that is not currently supported by DotPdf. These can only be generated by reading in a PDF file that contains unknown annotations.

PopupAnnotation

A PopupAnnotation is a companion annotation to any kind of BaseMarkupAnnotation. As such it can never appear on its own. A PopupAnnotation may be open (in view) or closed (out of view). The PopupAnnotation is connected to the BaseMarkupAnnotation via the ParentAnnotation property and the BaseMarkupAnnotation is connected to the the PopupAnnotation via its Popup property. When in view, the PopupAnnotation will appear within its Bounds.

ⓘ Even though the PopupAnnotation expects a BaseMarkupAnnotation for its ParentAnnotation property, the property is a BaseAnnotation. The PDF specification allows this, even though it is not strictly correct. If the ParentAnnotation is not a BaseMarkupAnnotation, the properties will not reflect each other.

The PopupAnnotation has properties that represent the Contents, AuthorName, ModificationDate, and Color of the parent annotation. When the PopupAnnotation is connected to an appropriate parent BaseMarkupAnnotation, it these properties will reflect or modify the matching properties in the ParentAnnotation.

If you set the Contents, AuthorName, or ModificationDate before setting the ParentAnnotation, these property values will be lost.

Property Name	Property Type	Definition
AuthorName	string	Gets or sets the author of the annotation. Conventionally, this will be set to the current username or the full name of the user who is making the annotation.
Color	IPdfColor	Gets or sets the dominant color for the annotation. The interpretation of Color depends on the annotation. It may represent the color of the annotations icon (if any) or the border of the annotation.
Contents	string	Represents the text of the annotation. Its interpretation depends on the annotation type. For sticky note annotations, it will be the contents of the note.
IsOpen	bool	Gets or sets whether the PopupAnnotation should be in view when the document is opened.
ModificationDate	DateTime	Gets or sets the modification date of the annotation. DotPdf does not track or modify this property.
ParentAnnotation	BaseAnnotation	Gets or sets the parent annotation for the PopupAnnotation. The parent annotation should be a BaseMarkupAnnotation even though the PDF specification allows for any type of annotation.

The following C# code creates a RectangleAnnotation with an attached PopupAnnotation.

```
public void RectangleWithPopup()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 doc.EmbedGeneratedContent = false;

 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
 RectangleAnnotation rectAnnot = new RectangleAnnotation(new PdfBounds(36, 300, 200,
 200));
 rectAnnot.InternalColor = PdfColorFactory.FromRgb(1, 1, 0);
 rectAnnot.Color = PdfColorFactory.FromRgb(0, 0, 0);
 page.Annotations.Add(rectAnnot);
```

```
 PopupAnnotation popup = new PopupAnnotation(new PdfBounds(36, 400, 150, 350),
 rectAnnot);
 popup.Color = PdfColorFactory.FromRgb(.7, 0, 0);
 popup.IsOpen = true;
 page.Annotations.Add(popup);
 rectAnnot.Contents = "This space intentionally left blank.";
 rectAnnot.AuthorName = "Ignatius P. Reilly";

 doc.Save("rect_and_popup.pdf");
}
```

SoundAnnotation

A SoundAnnotation is a note on a page with an associated Sound object. A SoundAnnotation appears on the page with an icon specified by IconName. When the icon is double-clicked (or activated in some other way) by the user, it will play the sound. The PDF specification has two recommended icon names, Speaker and Mic. The specification alludes that other names may be supported, but there is no further information as to what those names might be.

If you want a specific icon, it's best to create a custom appearance for the annotation.

The following C# code creates a sound annotation.

```
using (FileStream stm = new FileStream(@"mysound.wav",
  FileMode.Open, FileAccess.Read, FileShare.Read))
{
 WavReader reader = new WavReader(stm);
 Sound sound = Sound.FromWavReader(reader);
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
 SoundAnnotation anno = new SoundAnnotation(new PdfBounds(72, 600, 72, 72));
 anno.Sound = sound;
 page.Annotations.Add(anno);
 doc.Save("soundannot.pdf");
}
```

Markup annotations

Markup annotations are intended for document editing and collaboration. The annotations include:

- CalloutAnnotation
- CaretAnnotation
- EllipseAnnotation
- LineAnnotation
- PolygonAnnotation and PolylineAnnotation
- RectangleAnnotation
- RedactionProposalAnnotation
- RubberStampAnnotation
- StickyNoteAnnotation
- TextBoxAnnotation
- TextMarkupAnnotation
- TypeWriterAnnotation

CalloutAnnotation

A CalloutAnnotation is a TextBoxAnnotation that also serves to point to content on the page. A CalloutAnnotation includes a Line that defines where the annotation points as well as a LineEnding that defines how the end of the line should appear. There are no guidelines as to how the Line should appear, but generally speaking, it should start from one edge of the Bounds nearest to the target and end at the point of interest. While the point of origin doesn't have to start at the annotation, if a user moves the annotation in Acrobat, the viewer will change the point of origin.

To make it easier to use there CalloutAnnotation constructor that includes a PdfPoint describing where the annotation will point and it will choose an appropriate set of points in order to make the call out line look least offensive. In addition, the CalloutAnnotation also has a method called PointAt(PdfPoint target) which will return a new CalloutLine object that points to the given point.

Property name	Property type	Description
Line	CalloutLine	Gets or sets an object that defines the geometry of the line that will be drawn for the annotation. CalloutLine is an abstract type and may be either a TwoPointCalloutLine or a ThreePointCalloutLine. Oddly enough, this property is valid if it is null. In this case, the CalloutAnnotation will render the same as a TextBoxAnnotation.
LineEnding	LineEndingKind	Gets or sets the line ending for the callout line which will appear at the target point.

The following C# code creates a CalloutAnnotation.

```
CalloutAnnotation annot = new CalloutAnnotation(new PdfBounds(72, 360,
300, 200),
      "Lorem ipsum sic dolor", new PdfPoint(144, 200));
somePage.Annotations.Add(annot);
```

CaretAnnotation

A CaretAnnotation represents an editor's markup where text or other content should be inserted. The caret is defined by the Bounds of the annotation. The caret symbol will be drawn such that it fills the bounds with the point of the caret centered left/right and pointing to the top of the bounds.

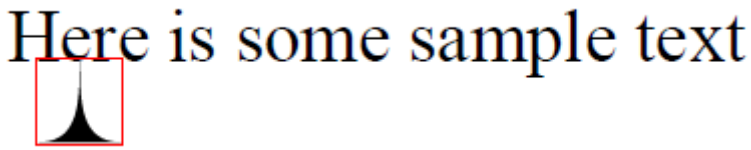
Property name	Property type	Description
InsetArea	PdfBounds	A rectangle that specifies margins around the caret symbol. The rectangle needs to be fully contained within the Bounds rectangle.

Property name	Property type	Description
Symbol	CaretSymbol	Changes the symbol used for the caret. When set to none, the symbol will be the default caret shape. When set to Paragraph, it will be the paragraph symbol (¶).

The following C# code creates a caret annotation and shows its bounds.

```
public void Caret()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
 string font = doc.Resources.Fonts.AddFromFontName("Times New Roman");
 page.DrawingList.Add(new PdfTextLine(font, 18, "Here is some sample text", new
 PdfPoint(72, 750)));
 CaretAnnotation caret = new CaretAnnotation(new PdfBounds(80, 730, 20, 20));
 page.Annotations.Add(caret);
 page.DrawingList.Add(new PdfRectangle(caret.Bounds.Expand(0.5),
 PdfColorFactory.FromRgb(1, 0, 0), .5));
 doc.Save("caret.pdf");
}
```

The code snippet produces the following output.



EllipseAnnotation

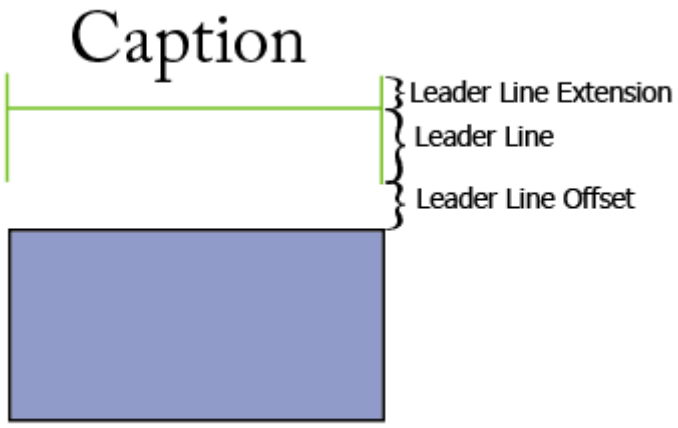
An EllipseAnnotation is identical to a RectangleAnnotation except that it is rendered as an ellipse that fits within the Bounds property.

LineAnnotation

A LineAnnotation is representation of a line on the page. It may contain decorative line endings, a caption, and an intended usage. Usage refers to the intent of the line which may be one of Line, Arrow, or Dimension.

When a line annotation has a caption, the caption may be positioned above the line or within the line by setting the CaptionPositioning property. Normally, captions are positioned centered along the length of the line and at a fixed vertical position based on CaptionPositioning, but by setting the CaptionOffset property, the caption will be moved relative to its normal placement based on that value. For example, if you wanted to position the caption below the line, you would set CaptionPositioning to Top and set CaptionOffset to new PdfPoint(0, -fontAscentInPoints).

A line may have a set of leader lines attached to it. Leader lines are perpendicular ends that extend from the line, usually to indicate a dimension.



A leader line is made from three parts, a leader line, a leader line extension and a leader line offset. A line should only have a leader line extension and a leader line offset if it also has a leader line. These elements are in PDF units.

Property name	Property type	Description
CaptionOffset	PdfPoint	The relative offset of placement from its normal position.
CaptionPositioning	CaptionPositionKind	One of either Top or Inline, specifying whether the text will appear above or within the line itself.
EndPt (Required)	PdfPoint	Gets or sets the end point of the line.
IsCaptioned	bool	Gets or sets whether the Content property will be used as a caption.
LeaderLineExtensionLength	double	Gets or sets the length of the leader line extensions (see diagram).
LeaderLineLength	double	Gets or sets the length of the leader lines.
LeaderLineOffset	double	Gets or sets the offset of the leader line from an object being measured.
LineEnding	LineEndingKind[]	A two entry array containing the LineEndingKind for the start and the end of the line.
StartPt (Required)	PdfPoint	Gets or sets the start point of the line.
Usage	LineUsageKind	Gets or sets the intent of the line.

PolygonAnnotation and PolylineAnnotation

A PolygonAnnotation is an annotation that is represented by three or more points connected in a closed path.

Property name	Property type	Description
Effect	BorderEffect	Gets or sets an effect to apply to the border of the polygon when it is rendered.
InternalColor	IPdfColor	Gets or sets an internal color of the polygon.
IsInternalColorTransparent	bool	When set to true, the internal color is transparent.
LineEnding	LineEndingKind[]	Gets or sets the line ending for an open polygon. The PDF specification indicates that for a polygon, these elements may be present even though they are ignored. The will be honored in PolylineAnnotation.
Vertices	IList<PdfPoint> List<PdfPoint>	A collection of PdfPoint that represent the vertices of the polygon. There should be a minimum of three points in the collection for a valid polygon.

A PolylineAnnotation is identical to a PolygonAnnotation except that it the LineEndings will be honored and a PolylineAnnotation is valid with a minimum of two points.

RectangleAnnotation

A RectangleAnnotation is an annotation that represents a rectangle drawn on the page. The rectangle may have an outline or it may be filled with a color. It may also have an effect applied to the border. The EllipseAnnotation inherits directly from RectangleAnnotation and is no different except in the shape that will be drawn on the page.

Property name	Property type	Meaning
Effect	BorderEffect	Gets or sets an effect to apply when rendering the border of the rectangle.
InternalColor	IPdfColor	Gets or sets the color used to fill the rectangle.

RedactionProposalAnnotation

The RedactionProposalAnnotation is an annotation that indicates an area on the page to be redacted later by a viewer or other PDF processing tool. The RedactionProposalAnnotation does not perform actual redaction nor does it change page content in any way. When a redaction is applied

by a viewer, the annotation is removed from the page, all content within the area of redaction will be stripped and the redaction appearance will be added to the page's content.

At a minimum, the RedactionProposalAnnotation needs the Bounds to be set to the area of the document to be redacted. You can also use the Regions property to create a set of PdfQuadrilateral objects that will be used for the redaction area.

There are a number of properties that can be set that affect how the redaction will appear *after* it has been applied. For example, if you set the OverlayText property, that text will be written into the redaction area. This is useful if you wanted each redaction to have a note on it to alert the reader why the content is not present ("removed by court order," for example).

Property name	Property type	Description
AutoGenerateBasicAppearance	bool	If set to true, the annotation will autogenerate a simple appearance upon being rendered. If the Regions collection is empty, it will generate a single rectangle outlined with the annotation's Color. If the Regions collection is not empty, it will generate a single PdfPath with each quadrilateral outlined in the annotation's Color.
DefaultTextAppearance	PdfTextAppearance	This property, if set, will represent how the OverlayText will appear on the annotation. If not set, the text, if any, will appear in Helvetica 12 point.
IsOverlayTextRepeated	bool	If set to true, the OverlayText string will be repeated over the surface of the redacted area when the redaction is applied.
OverlayText	string	Gets or sets text that will be rendered on the redaction area after the redaction has been applied.
RedactionInteriorColor	IPdfRgbColor	An RGB color that will be used to render the interior area of the redaction after it has been applied. If RedactionTemplate is set, this will be ignored.
RedactionTemplate	string	Gets or sets the name of a template resource to use when rendering redaction after it has been applied.

Property name	Property type	Description
Regions	PdfQuadrilateralCollection	Gets a collection of PdfQuadrilateral objects to use for the area(s) to be redacted. If this collection is non-empty, upon rendering, the Bounds property will be adjusted to reflect the contents of the Regions.
TextAlignment	AnnotationTextAlignment	Gets or sets how the OverlayText will appear when rendered.

The following C# code adds a simple redaction proposal to a page.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer sed diam id ipsum
 egestas lacinia. Nulla vel nulla sit amet elit aliquet feugiat. Donec varius euismod
 augue, vel lacinia arcu mollis nec. In tempor neque vitae velit dapibus cursus. Etiam
 ut sodales neque. Integer quis sem orci. Praesent tincidunt odio non sapien adipiscing
 vestibulum. Duis porttitor quam ut metus posuere at venenatis velit gravida. Nulla
 facilisi. Ut dapibus suscipit risus, vitae tempor velit adipiscing id. Vestibulum ante
 ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Fusce mattis
 volutpat metus, ac molestie tortor tristique sed. Cras lacinia facilisis lobortis.
 Duis elementum congue bibendum.");
page.DrawingList.Add(box);

RedactionProposalAnnotation redaction = new RedactionProposalAnnotation(new
 PdfBounds(72, 450, 150, 36));
redaction.Color = PdfColorFactory.FromRgb(1, 0, 0);

page.Annotations.Add(redaction);

doc.Save("simpleredact1.pdf");
```

This will add a red hollow box on page which when the redaction is actually applied by a viewer will remove the text below it and leave a blank spot behind.

RubberStampAnnotation

The RubberStampAnnotation is an annotation that is used to mark a page with standard text as if it was created by a rubber stamp. The PDF specification defines a list of standard rubber stamp types for use in this annotation. Even though the text of the rubber stamp can be set to anything, the specification indicates that only this set needs to be supported:

- Approved
- AsIs
- Confidential
- Departmental
- Draft
- Experimental
- Expired

- Final
- ForComment
- ForPublicRelease
- NotApproved
- NotForPublicRelease
- Sold
- TopSecret

If you want to ensure that you create RubberStampAnnotation objects with supported rubber stamp kinds, either use the RubberStampAnnotation that takes a RubberStampKind or use the utility method FromRubberStampKind() to covert a RubberStampKind to a string.

Property Name	Property Type	Description
StampLabel (Required)	string	This is the label that will be used for the rubber stamp. Although it can be any non-null, non-empty string, there is no guarantee that anything but the standard types can be rendered by a viewer. (Required)

The following C# code creates a TopSecret stamp.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

RubberStampAnnotation annot = new RubberStampAnnotation(RubberStampKind.TopSecret, new
 PdfBounds(72, 650, 144, 72));
page.Annotations.Add(annot);

doc.Save("topsecretstamp.pdf");
```

StickyNoteAnnotation

A StickyNoteAnnotation represents a note of information placed on the page. The text of the information is stored in the Contents property of the annotation. The annotation can also have one of a set of standard icons associated with it on the page and the annotation may be either an "open" or "closed" state. When a StickyNoteAnnotation is closed, only the icon is visible. When it is open, a PopupAnnnotation will be shown that shows the Contents and (possibly) allows it to be edited. Finally, StickyNoteAnnotations can be used as part of a review process. The PDF specification defines a general ReviewProcess and two specific ones that each have discrete states of the review. It is possible to define your own kinds of review process, but there is no guarantee that it will be supported by any particular PDF viewer.

Property name	Property type	Description
IconName	string	A name of an icon to use for the annotation on the page. If this property is not set, the icon will default to "Note."

Property name	Property type	Description
IsOpen	bool	Gets or sets the open state of the sticky note.
ReviewProcess	ReviewProcess	Gets or sets the review process for this sticky note.

The IconName can be set to a standard name by using static properties in StickyNoteAnnotation. The entire list can be retrieved from the StandIconNames static property.

The following C# code makes a help sticky note.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

StickyNoteAnnotation sticky = new StickyNoteAnnotation(new PdfBounds(144, 400, 72, 72),
 "note text here", new PdfBounds(156, 420, 100, 100));
sticky.Color = PdfColorFactory.FromRgb(1, 1, .8);
sticky.IconName = StickyNoteAnnotation.HelpIconName;
page.Annotations.Add(sticky);
doc.Save("stickynote.pdf");
```

If you use the StickyNoteAnnotation constructor that has a popupBounds parameter, the constructor will also construct and attach a PopupAnnotation to the StickyNote annotation.

TextBoxAnnotation

A TextBoxAnnotation is simply a box on the page with text in it. Unlike the StickyNoteAnnotation, the text box annotation doesn't have an open/closed state, but is instead always open and constrained by the bounds. The text may be either plain text, using the Content property or rich text, using the RichTextContent and the Content properties (the Content property should be set to a plain text equivalent of the rich text).

Property name	Property type	Description
DefaultTextAppearance	PdfTextAppearance	Gets or sets the appearance of text in the text box. If not set or set to null, the text appearance will default to 10pt Helvetica.
DefaultRichTextTyleString	string	Gets or sets the default style string used for rich text, for example "font: 12pt Arial".
Effect	BorderEffect	Gets or sets a border effect for the text box.
InsetArea	PdfBounds	Gets or sets the inset area for the text box, creating margins for the text. This property should be set so that it is fully contained within the Bounds property.

Property name	Property type	Description
TextAlignment	AnnotationTextAlignment	Gets or sets how the text will be aligned or justified in the Bounds.

The following C# code creates a TextBoxAnnotation.

```
TextBoxAnnotation annot = new TextBoxAnnotation(new PdfBounds(72, 360,
300, 200), "Lorem ipsum sic dolor");
annot.Color = PdfColorFactory.FromRgb(.39, .58, .92);
somePage.Annotations.Add(annot);
```

TextMarkupAnnotation

A TextMarkupAnnotation is not an annotation that contains text. Instead, it is a set of possible mark-ups to add to text on a page. The annotation is not itself associated with the text on the page at all. Any associations or relationships between the annotation and the text is made by the PDF viewing software.

The location of the markup is represented by the Regions property, which is a PdfQuadrilateralCollection of (possibly) disjoint quadrilaterals that surround areas of interest.

The appearance of the markup is determined by the MarkupKind property which is one of:

- Highlight
- Underline
- Squiggly
- StrikeOut

The particular markup will be rendered in the Color of the annotation.

Property name	Property type	Description
MarkupKind	TextMarkupKind	Gets or sets the type of the markup.
Regions	PdfQuadrilateralCollection	Defines the areas of interest for the annotation.

The following C# code creates a highlight TextMarkupAnnotation.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
"Lorem ipsum dolor sit amet, consectetur adipiscing elit.");
page.DrawingList.Add(box);

TextMarkupAnnotation textMarkup = new TextMarkupAnnotation(TextMarkupKind.Highlight);
textMarkup.Color = PdfColorFactory.FromRgb(1, 1, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
page.Annotations.Add(textMarkup);
doc.Save("highlightmarkupannot.pdf");
```

TypeWriterAnnotation

The TypeWriterAnnotation is used for placing text on the page in a way that implies no real constraints to the text boundary and very little extra in the appearance beyond the text itself. The annotation itself inherits from the TextBoxAnnotation. By default, the text is placed using the annotation's StartPoint property. This point will be the left edge and baseline of the text in the annotation. The PDF specification uses the Bounds property for the placement of the text, but this can be cumbersome. If the AutoGenerateBounds property is true, the Bounds will be calculated from the StartPoint, otherwise the bounds will be taken as is and the appearance may be unpredictable.

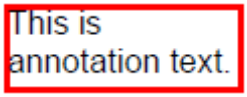
Property name	Property type	Description
AutoGenerateBounds	bool	If set to true (default), the annotation will use the StartPt property, the Contents property, and the font information to calculate the Bounds property at render time. Lines will be split at "\r" or "\n" characters.
AutoGenerateInsetArea	bool	If set to true and if AutoGenerateBounds is true, then the InsetArea will be calculated as if it were the bounds and the Bounds will be calculated by expanding the InsetArea by the margins.
LeftRightMargin	double	If AutoGenerateBounds is true, this value will be used to create margins on the left and right edges. Must be non-negative.
StartPoint	PdfPoint	If AutoGenerateBounds is true, this is starting point for text within the annotation. The X coordinate will be the left edge of the text and the Y coordinate will be the text baseline.
TopBottomMargin	double	If AutoGenerateBounds is true, this value will be used to create margins on the top and bottom edges. Must be non-negative.

Even though the PDF specification is clear about the intent and usage of the InsetArea of a TypeWriterAnnotation, Adobe Acrobat does not honor it correctly, nor does Acrobat honor a custom appearance for the annotation. The LeftRightMargin an TopBottomMargin are therefore not recommended for use with Adobe Acrobat.

The following C# code creates a TypeWriterAnnotation and shows its bounds.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
string fontName = "Helvetica";
```

```
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

TypeWriterAnnotation annot = new TypeWriterAnnotation(new PdfPoint(72, 750), "This is
\rannotation text.");
annot.DefaultTextAppearance = new PdfTextAppearance(fontName, 8);
page.Annotations.Add(annot);

// this is the method used by the annotation during rendering
PdfBounds bounds = annot.CalculateBounds(doc.Resources, annot.StartPoint,
 annot.Contents);
PdfRectangle boundsRect = new PdfRectangle(bounds, PdfColorFactory.FromRgb(1, 0, 0),
 1);
page.DrawingList.Add(boundsRect);
doc.Save("typewriter.pdf");
```



Widget annotations

Widget annotations are used for interactive forms. Each widget represents a specific type of user-interface element and implements the interface IFormElement, which describes the contents and behavior of a PDF form field. The supported types of widget annotations are:

- CheckboxWidgetAnnotation
- ChoiceWidgetAnnotation
- PushButtonWidgetAnnotation
- RadioButtonWidgetAnnotation
- SignatureWidgetAnnotation
- TextWidgetAnnotation

CheckboxWidgetAnnotation

A checkbox widget annotation is a widget annotation that represents a two-state selection. It is typically represented by an empty box when it is not selected and a box with a mark in it (an x or a tick mark).

The checkbox widget annotation does not include any text, it is just the graphic representation. The AppearanceSet is used to define how the widget will be drawn in the Normal, Rollover and Activated appearances. Within each appearance, there should be an appearance entry named after each state. The appearance entry for a checked widget will be named "Yes" and the appearance entry for not checked will be named "Off." You can use the properties CheckboxWidgetAnnotation.CheckedValue and CheckboxWidgetAnnotation.ClearedValue instead.

ℹ While the values for the checkbox on/off states can be any two different strings, you are strongly encouraged to use "Yes" and "Off."

Property name	Property type	Description
CheckedValue	string	Gets the recommended checked value string "Yes."

Property name	Property type	Description
ClearedValue	string	Gets the recommended cleared value string "Off."
Value	string	The current value of the widget.
DefaultValue	string	The default value of the widget.

Since there can be a great deal of code for creating appearances for checkboxes, DotPdf includes standard appearances which will be installed in your document Resources Templates. These templates will be shared among all CheckBoxWidgets that share them. This is done internally via the DefaultWidgetTemplates object.

If you do not supply appearances, Adobe Acrobat does not reliably render the widget.

The following C# code makes a check box.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
CheckboxWidgetAnnotation annot = new CheckboxWidgetAnnotation(doc.Resources, new
 PdfBounds(72, 360, 18, 18), "check", null, null);
annot.Value = CheckboxWidgetAnnotation.CheckedValue;
page.Annotations.Add(annot);
doc.Save("checkdocsimp.pdf");
```

Use this constructor to implicitly install default appearances in the widget and your document resources.

The following C# code manually installs standard appearances.

```
DefaultWidgetTemplates.InstallDefaultAppearances(doc.Resources, false);
myCheck.Appearance.Normal.Add(CheckboxWidgetAnnotation.CheckedValue,
 DefaultWidgetTemplates.CheckboxCheckedNormalName);
myCheck.Appearance.Normal.Add(CheckboxWidgetAnnotation.ClearedValue,
 DefaultWidgetTemplates.CheckboxClearedNormalName);
myCheck.Appearance.Activated.Add(CheckboxWidgetAnnotation.CheckedValue,
 DefaultWidgetTemplates.CheckboxCheckedActivatedName);
mycheck.Appearance.Activated.Add(CheckboxWidgetAnnotation.ClearedValue,
 DefaultWidgetTemplates.CheckboxClearedActivatedName);
```

When you add appearances, the second argument is always the name of a Template resource. InstallDefaultAppearances() will add in new Template resources using the names shown above.

ChoiceWidgetAnnotation

A choice widget annotation is an annotation that lets a user select one or more items from a list of possible choices. The list can either appear as a list in a box, a pop-up list, or a pop-up list with a text entry field (also called a combo box). The choices are set via a list of pairs of string objects. Each pair contains a display name and an export name. The export value is optional. If omitted, the display value will instead be used. The purpose of the pair is so that, for example, it would be possible to generate separate forms in different languages that display in the native language but all submit with the same export values, making the data submitted language neutral.

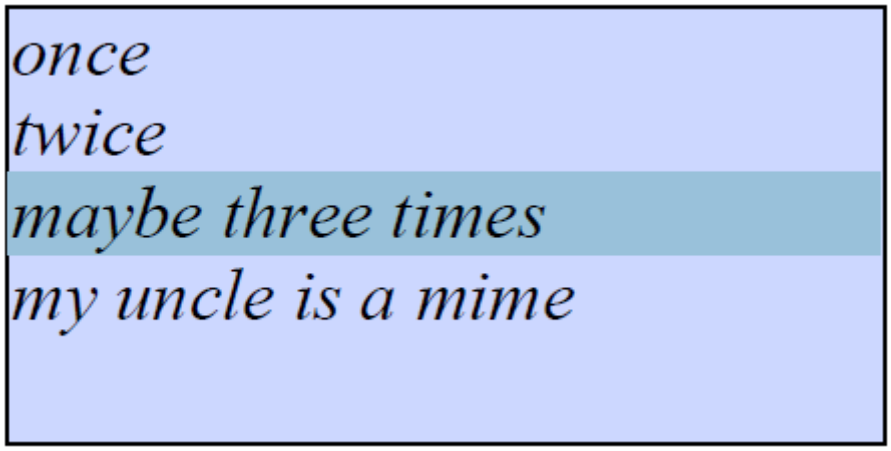
Like all widgets, ChoiceWidgetAnnotation requires an appearance for the widget. This appearance can't be shared between different ChoiceWidgetAnnotations and is built lazily - just before a render - so that it will be unaffected by changes in Bounds.

Text of items in the list will be rendered using the DefaultTextAppearance property.

Property name	Property type	Description
AllowMultiSelect	bool	If set to true, the user can have multiple items selected.
AutoGenerateBasicAppearance	bool	If set to true (default), the widget will make and install a basic appearance for the widget.
Choices	IList<ChoicePair>	A list of elements to present to the user. Each choice pair has a DisplayName and an optional ExportName. The DisplayName will be presented to the user. The ExportName (or the DisplayName, if the ExportName is null)will be used when submitting the data.
CurrentSelection	IList<int>	Contains a list of indexes of current selections. If AllowMultiSelect is false, only the first value (if any) will be used.
FirstVisibleChoice	int	Gets or sets the index of the first visible choice in the list.
ValueAsString	string	Returns a comma separated list of the choices.

The following C# code creates a simple ChoiceWidgetAnnotation.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
ChoiceWidgetAnnotation anno = new ChoiceWidgetAnnotation(ChoiceWidgetKind.ListBox,
 "choices", new PdfBounds(72, 400, 288, 144),
 "once", "twice", "maybe three times", "my uncle is a mime");
anno.DefaultTextAppearance = new PdfTextAppearance();
anno.DefaultTextAppearance.FontName = "Times-Italic";
anno.DefaultTextAppearance.FontSize = 24;
anno.AutoGenerateBasicAppearance = true;
anno.CurrentSelection.Add(2);

doc.Form = new PdfForm();
page.Annotations.Add(anno);
doc.Form.Fields.Add(anno);
doc.Save("choicelist.pdf");
```



The following C# code creates the appearance for the list.

```
private double StartLineBottom(PdfBounds bounds, int index, double lineHeight)
{
 return bounds.Top - 1 - ((index + 1) * lineHeight);
}

private string MakeBasicAppearanceList(GlobalResources gr, PdfBounds bounds,
 PdfTextAppearance app, double borderWidth, IPdfColor outlineColor, IPdfColor
 fillColor, IList<ChoicePair> choices, IList<int> currentSelection)
{
 var res = gr.Fonts.Get(app.FontName);
 double lineHeight = res.Metrics.LineSpacing(app.FontSize);
 double baseLine = (res.Metrics.Descent * app.FontSize) / -1000.0;

 bounds = new PdfBounds(0, 0, bounds.Width, bounds.Height);
 DrawingTemplate template = new DrawingTemplate(bounds);
 if (outlineColor == null && fillColor == null)
  return null;

 PdfRectangle rect = new PdfRectangle(bounds, fillColor);
 template.DrawingList.Add(rect);

 PdfMarkedContent markedContent = new PdfMarkedContent("Tx");
 template.DrawingList.Add(markedContent);


 PdfBounds inset = new PdfBounds(bounds.Left + 1, bounds.Bottom + 1, bounds.Width - 2,
 bounds.Height - 2);
 rect = new PdfRectangle(inset, outlineColor);
 rect.Clip = true;
 markedContent.DrawingList.Add(rect);

 markedContent.DrawingList.Add(new GSave());


 if (currentSelection != null)
 {
  IPdfColor selColor = PdfColorFactory.FromRgb(0.6, 0.75866, 0.854904);
  foreach (int sel in currentSelection)
  {
   double selY = StartLineBottom(bounds, sel, lineHeight);
   PdfBounds selBounds = new PdfBounds(1, selY, inset.Width - 1, lineHeight);
```

```
   rect = new PdfRectangle(selBounds, selColor);
   markedContent.DrawingList.Add(rect);
  }
 }

 for (int i = 0; i < choices.Count; i++)
 {
  ChoicePair pair = choices[i];
  double selY = StartLineBottom(bounds, i, lineHeight) + baseLine;
  PdfTextLine line = new PdfTextLine(app.FontName, app.FontSize, pair.DisplayName ??
 pair.ExportName,
  new PdfPoint(2.0, selY));
  markedContent.DrawingList.Add(line);
 }

 markedContent.DrawingList.Add(new GRestore());

 string name = gr.Templates.Add(template);
 return name;
}
```

ℹ The actual content of the list is put within a PdfMarkedContent object with the "Tx" mark, setting it off as the text content of the box.

PushButtonWidgetAnnotation

A PushButtonWidgetAnnotation is the simplest type of widget annotation. It has no value associated with it. Instead, it only serves to trigger actions of some kind. This is done by adding a new action to its AdditionalActions.ClickDown list. Like other widgets, a PushButtonWidgetAnnotation needs to have one or more appearances in order to be rendered. The class includes a property to automatically generate an appearance as well as a public factory method for creating one.

To ensure that an appearance is made for the button, set the AutoGenerateAppearance property to null.

ℹ The auto-generated appearance for a button is an outlined round-cornered rectangle with centered text clipped to the outline.

The following C# code creates a button that plays a sound.

```
using (FileStream stm = new FileStream(@"mysound.wav",
   FileMode.Open, FileAccess.Read, FileShare.Read))
{
 WavReader reader = new WavReader(stm);
 Sound sound = Sound.FromWavReader(reader);

 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
 PushButtonWidgetAnnotation button = new PushButtonWidgetAnnotation(new PdfBounds(72,
 400, 144, 40),
    "Now Hear This", null, null);
 button.AutoGenerateBasicAppearance = true;
 PdfSoundAction action = new PdfSoundAction(sound);
 button.AdditionalActions.OnClickDown.Add(action);

 doc.Form = new PdfForm();
 page.Annotations.Add(button);
 doc.Form.Fields.Add(button);
```

```
 doc.Save("soundbutton.pdf");
}
```

RadioButtonWidgetAnnotation

RadioButtonWidgetAnnotation are a button widget that is represented by a set/cleared state. When radio buttons are cleared, they are represented by the value "Off". When they are set, they are represented by a string value that is unique among the group of radio buttons. RadioButtonWidgetAnnotations are unusual among widgets in that they are not usable in isolation. RadioButtonWidgetAnnotation objects need to have a parent RadioButtonFormField which contains the semantics for the entire group.

Like CheckBoxWidgetAnnotation objects, RadioButtonWidgetAnnotations do not have any particular text associated with their appearance - they are usually just the button itself. It does need its own set of appearances, but these can be created at construction time and can be shared among all radio buttons.

The steps for creating a set of RadioButtonWidgetAnnotation objects is as follows:

1. Make RadioButtonWidgetAnnotations for each choice, setting the FieldName to null and passing in the string name of the "selected" value as the onValue.
2. Set the Value and DefaultAppearanceState to the either RadioButtonWidgetAnnotation.ClearedValue or to the string name of its "selected" value.
3. Create a RadioButtonFormField object.
4. Set the form field's Value and Default Value to the radio button you would like selected.
5. Set the form field's FieldName.
6. Put each radio button into the form field's ChildFields collection.
7. Set each radio button's ParentField to the form field.
8. Add each radio button to the page's Annotations collection.
9. Construct a new PdfForm and assign it to the document's Form property.
10. Add the form field to the document's Form's Fields collection.

These steps are illustrated (in a slightly different order) in this sample C# code for making radio buttons:

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
string font = doc.Resources.Fonts.AddFromFontName("Arial");
RadioButtonWidgetAnnotation yesButton = new RadioButtonWidgetAnnotation(doc.Resources,
 new PdfBounds(72, 700, 12, 12),
 null, null, null, "Yes", true);
yesButton.DefaultAppearanceState = yesButton.Value = "Yes";

RadioButtonWidgetAnnotation noButton = new RadioButtonWidgetAnnotation(doc.Resources,
 new PdfBounds(72, 680, 12, 12),
 null, null, null, "No", true);
noButton.DefaultAppearanceState = noButton.Value =
 RadioButtonWidgetAnnotation.ClearedValue;
```

```
RadioButtonWidgetAnnotation undecidedButton = new
 RadioButtonWidgetAnnotation(doc.Resources, new PdfBounds(72, 660, 12, 12),
 null, null, null, "Undecided", true);
undecidedButton.DefaultAppearanceState = undecidedButton.Value =
 RadioButtonWidgetAnnotation.ClearedValue;

page.Annotations.Add(yesButton);
page.DrawingList.Add(new PdfTextLine(font, 12, "Yes",
 new PdfPoint(yesButton.Bounds.Right + 4, yesButton.Bounds.Bottom)));

page.Annotations.Add(noButton);
page.DrawingList.Add(new PdfTextLine(font, 12, "No",
 new PdfPoint(noButton.Bounds.Right + 4, noButton.Bounds.Bottom)));

page.Annotations.Add(undecidedButton);
page.DrawingList.Add(new PdfTextLine(font, 12, "Undecided",
 new PdfPoint(undecidedButton.Bounds.Right + 4, undecidedButton.Bounds.Bottom)));

RadioButtonFormField ff = new RadioButtonFormField();
ff.FieldName = "Choice";
ff.ChildFields.Add(yesButton);
yesButton.ParentField = ff;
ff.ChildFields.Add(noButton);
noButton.ParentField = ff;
ff.ChildFields.Add(undecidedButton);
undecidedButton.ParentField = ff;
ff.Value = "Yes";
ff.DefaultValue = "Yes";
doc.Form.Fields.Add(ff);
doc.Save("threechoice.pdf");
```

RadioButtonFormField has several factory methods that do most of this work for you. It is strongly recommended that you use these methods to avoid errors in creation of the fields.

The following C# code creates a radio button set using the convenience factory method.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
string font = doc.Resources.Fonts.AddFromFontName("Arial");

string[] values = new string[] { "Yes", "No", "Undecided" };
PdfBounds[] bounds = new PdfBounds[] {
 new PdfBounds(72, 700, 12, 12),
 new PdfBounds(72, 680, 12, 12),
 new PdfBounds(72, 660, 12, 12)
};

RadioButtonFormField ff = RadioButtonFormField.MakeRadioSet(doc.Resources, page,
 "Choice", values[0], values[0],
 values, bounds);
doc.Form.Fields.Add(ff);

for (int i = 0; i < values.Length; i++)
{
 page.DrawingList.Add(new PdfTextLine(font, 12, values[i],
       new PdfPoint(bounds[i].Right + 4, bounds[i].Bottom)));
}
doc.Save("threechoiceeasy.pdf");
```

SignatureWidgetAnnotation

The SignatureWidgetAnnotation is used to indicate an area that needs to be signed by a user reading the document. The SignatureWidgetAnnotation does not sign the document, it indicates that a document needs a signature. The area for the signature is represented by the Bounds. This annotation doesn't need an appearance added it.

The following C# code adds a signature to a document.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
SignatureWidgetAnnotation sig = new SignatureWidgetAnnotation(new PdfBounds(72, 600,
 200, 40), "Signature", null, null);
page.Annotations.Add(sig);
doc.Form.Fields.Add(sig);
doc.Save("signhere.pdf");
```

TextWidgetAnnotation

The TextWidgetAnnotation is a widget that is used to building forms with text entry. It has a number of properties that dictate the formatting of text in the widget, making it one of the most configurable widgets. Like most of the widget annotations, it should have an appearance associated with it, which can be done for you if AutoGenerateBasicAppearance is true.

Property name	Property type	Description
AutoGenerateBasicAppearance	bool	If set to true, before rendering the widget will generate a basic appearance for the text box.
DefaultRichTextStyleString	string	Gets or sets a default rich text string to be used to define the style of the RichTextValue of the widget. Note that if you use RichTextValue, you need to also set the Vlue property to a plain text version of the rich text.
DefaultTextValue	string	Gets or sets the default value for the widget.
IsColumns	bool	If set to true, the MaximumLength property will be used to define columnar layout of the text. Note that IsColumns only makes sense if IsPassword, IsScrollable, and IsFileSelection are all false.
IsFileSelection	bool	If set to true, the text is meant to represent a file selection, in which case the value entered is supposed to be the path to the file.

Property name	Property type	Description
IsMultiLine	bool	If set to true, the text entered will be allowed to be multiple lines, otherwise it will be forced to be a single line. The default is false.
IsPassword	bool	If set to true, then the text entered will be treated as a password and will not be displayed direction. Note that text entered as a password should never be stored within the PDF, but should instead be used and removed from the field. If the PDF is saved without encryption and with a password value entered, the password will be stored in clear text.
IsRichText	bool	If set to true, then the content of the field will be rendered using rich text. Even if RichText is set to true, any setting of the RichTextValue should be reflected in the Value property as well.
IsScrollable	bool	If set to true, then the text widget will have a scroll bar on it if needed.
IsSpellChecked	bool	If set to true, then the text in the text widget will be marked for any spelling errors using a client service, if available.
MaximumLength	int	The greatest number of characters that may be entered into the field. This value must be non-negative.
RichTextValue	XmlDocument	The representation of the text content using rich text.
TextAlignment	AnnotationTextAlignment	Gets or sets the justification of the text displayed in the widget.
TextValue	string	The value to display in the text box.

The following C# code creates a text field with existing text.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
TextWidgetAnnotation tw = new TextWidgetAnnotation(new PdfBounds(72, 350, 300, 50),
 "noname", "");
tw.TextValue = "Spoon";

tw.DefaultTextAppearance = new PdfTextAppearance();
tw.DefaultTextAppearance.FontName = "Times-Italic";
tw.DefaultTextAppearance.FontSize = 42;
page.Annotations.Add(tw);

doc.Form = new PdfForm();
```

```
doc.Form.Fields.Add(tw);
doc.Save("textwidget.pdf");
```

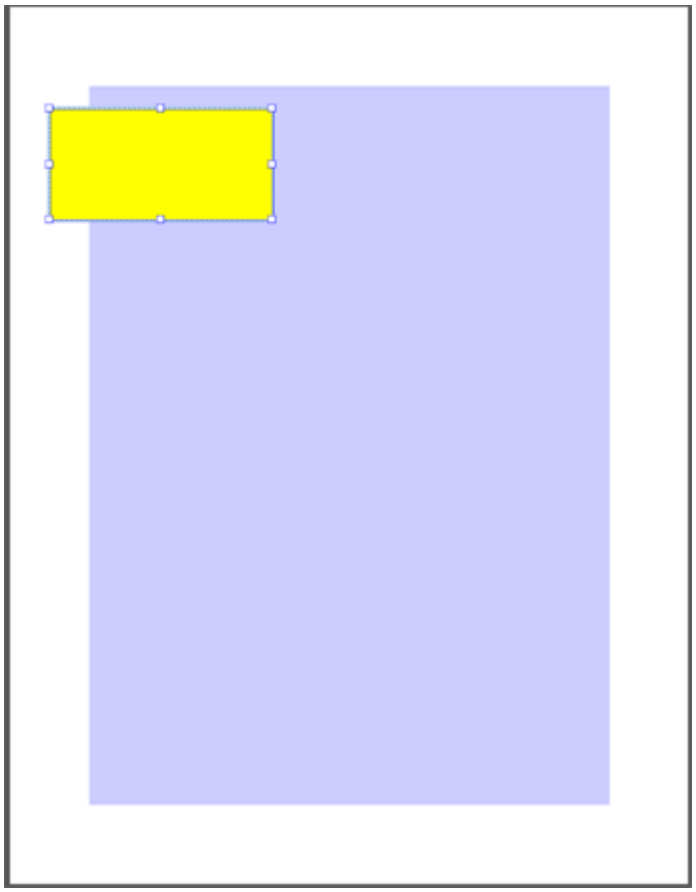
Use annotations

The following is a set of common tasks that can be done with the DotPdf annotation objects.

Place an annotation

This C# sample creates a page with a large light blue rectangle on it and then adds a yellow rectangle annotation with no border.

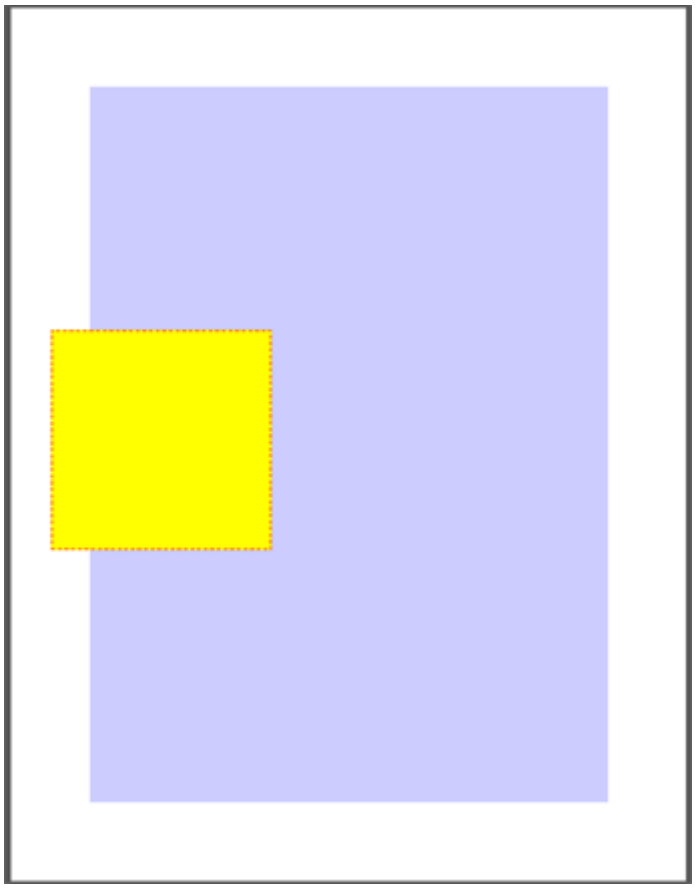
```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
page.DrawingList.Add(new PdfRectangle(new PdfBounds(72, 72, page.MediaBox.Width - 144,
 page.MediaBox.Height - 144),
PdfColorFactory.FromRgb(.8, .8, 1)));

RectangleAnnotation rectAnnot = new RectangleAnnotation(new PdfBounds(36, 600, 200,
 100));
rectAnnot.InternalColor = PdfColorFactory.FromRgb(1, 1, 0);
rectAnnot.Color = null;
page.Annotations.Add(rectAnnot);

doc.Save("simpleannot1.pdf");
```



Create an annotation with a custom border

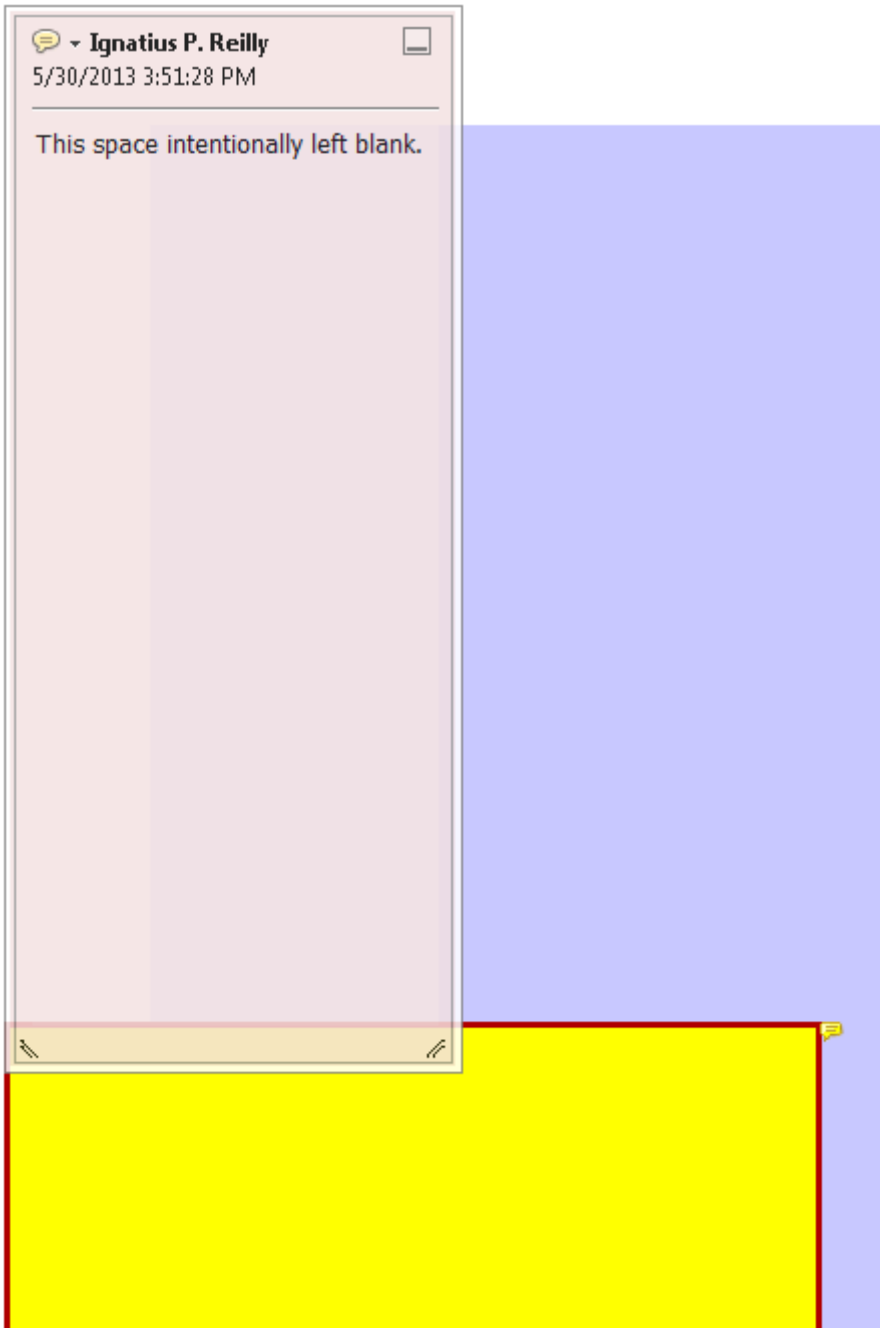
This C# sample creates a page with a light blue rectangle and a yellow rectangle annotation with an orange dashed border.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
page.DrawingList.Add(new PdfRectangle(new PdfBounds(72, 72, page.MediaBox.Width - 144,
 page.MediaBox.Height - 144),
PdfColorFactory.FromRgb(.8, .8, 1)));

RectangleAnnotation rectAnnot = new RectangleAnnotation(new PdfBounds(36, 300, 200,
 200));
rectAnnot.InternalColor = PdfColorFactory.FromRgb(1, 1, 0);
rectAnnot.Color = PdfColorFactory.FromRgb(1, .5, 0);
rectAnnot.Border = new AnnotationBorder(0, 0, 1.5, new double[] { 4, 1 });
page.Annotations.Add(rectAnnot);

doc.Save("simpleannot2.pdf");
```



Add a pop-up to a markup annotation

This C# sample shows how to add an open pop-up annotation to a markup annotation (in this case a rectangle annotation). Note that setting the pop-up color also changes the border color of the rectangle annotation

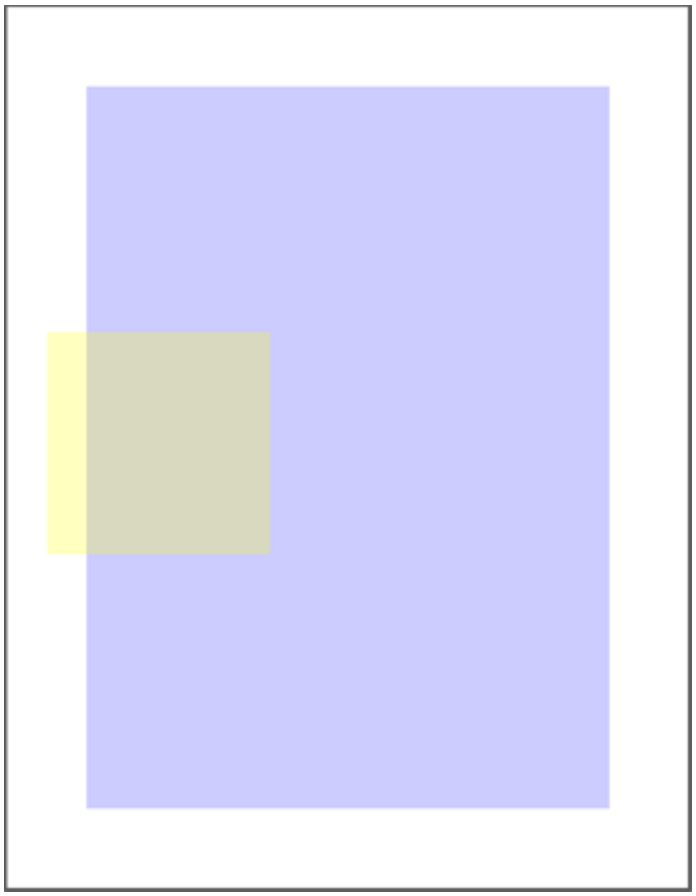


Create an annotation with transparency

This C# sample shows how to set transparency in a rectangle annotation.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;
```

```
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
page.DrawingList.Add(new PdfRectangle(new PdfBounds(72, 72, page.MediaBox.Width - 144,
 page.MediaBox.Height - 144),
PdfColorFactory.FromRgb(.8, .8, 1)));

RectangleAnnotation rectAnnot = new RectangleAnnotation(new PdfBounds(36, 300, 200,
 200));
rectAnnot.InternalColor = PdfColorFactory.FromRgb(1, 1, 0);
rectAnnot.Color = null;
rectAnnot.IsTransparent = true;
rectAnnot.Transparency = 0.75;
page.Annotations.Add(rectAnnot);

doc.Save("simpleannot4.pdf");
```

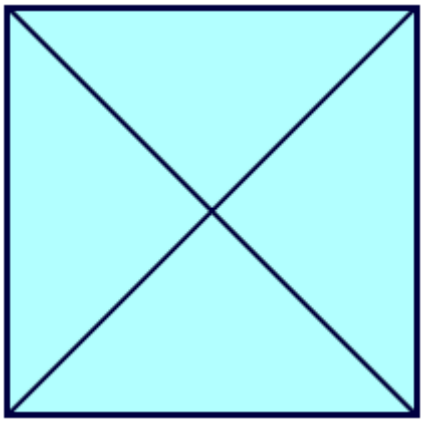


Skin an annotation

This C# sample demonstrates how to create an annotation with a custom "Normal" appearance. For simple skinning, you should create exactly one appearance and put it in the Normal collection under the name AppearanceSet.DefaultAppearanceName. This creates a rectangle with an x.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
```

```
DrawingTemplate template = new DrawingTemplate(new PdfBounds(0, 0, 100, 100));
IPdfColor outlineColor = PdfColorFactory.FromRgb(0, 0, .25);
IPdfColor fillColor = PdfColorFactory.FromRgb(.7, 1, 1);
template.DrawingList.Add(new PdfRectangle(new PdfBounds(1, 1, 98, 98), outlineColor, 1,
 fillColor));
PdfPath path = new PdfPath(outlineColor, 1, null);
path.MoveTo(1, 1); path.LineTo(99, 99);
path.MoveTo(1, 99); path.LineTo(99, 1);
template.DrawingList.Add(path);
string templateName = doc.Resources.Templates.Add(template);

RectangleAnnotation annot = new RectangleAnnotation(new PdfBounds(72, 300, 102, 102));
annot.Appearance = new AppearanceSet();
annot.Appearance.Normal.Add(AppearanceSet.DefaultAppearanceName, templateName);
page.Annotations.Add(annot);
doc.Save("simpleannot5.pdf");
```

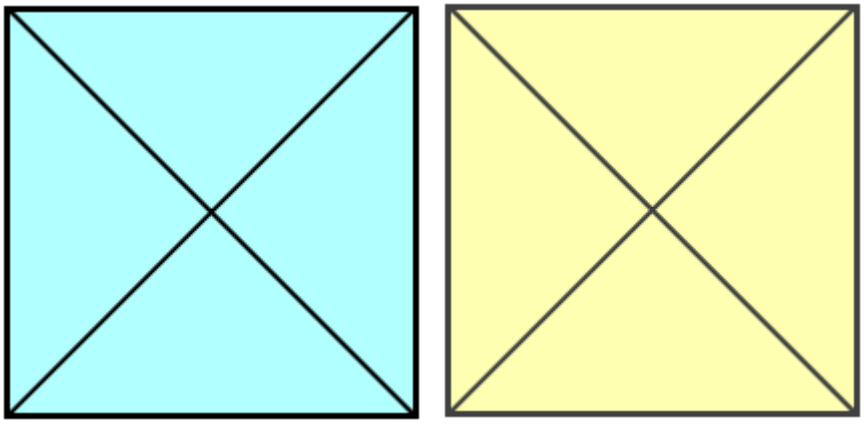


Make an annotation with a rollover appearance

Annotations can have different appearances for their normal and rollover states. The following C# code creates a rollover appearance.

```
public string MakeAppearance(PdfBounds bounds, IPdfColor outline, IPdfColor fill,
 GlobalResources resources)
{
 DrawingTemplate template = new DrawingTemplate(bounds);
 bounds = bounds.Expand(-1);
 template.DrawingList.Add(new PdfRectangle(bounds, outline, 1, fill));
 PdfPath path = new PdfPath(outline, 1);
 path.MoveTo(bounds.Left, bounds.Bottom);
 path.LineTo(bounds.Right, bounds.Top);
 path.MoveTo(bounds.Left, bounds.Top);
 path.LineTo(bounds.Right, bounds.Bottom);
 template.DrawingList.Add(path);
 return resources.Templates.Add(template);
}

public void MakeAnAnnotationWithARolloverAppearance()
{
 PdfGeneratedDocument doc = new PdfGeneratedDocument();
 doc.EmbedGeneratedContent = false;
 PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);
```

```
 RectangleAnnotation annot = new RectangleAnnotation(new PdfBounds(72, 300, 102, 102));
 annot.Appearance = new AppearanceSet();
 PdfBounds bounds = new PdfBounds(0, 0, 100, 100);
 annot.Appearance.Normal.Add(
  AppearanceSet.DefaultAppearanceName,
  MakeAppearance(
   bounds,
   PdfColorFactory.FromRgb(0, 0, 0),
   PdfColorFactory.FromRgb(.7, 1, 1),
   doc.Resources));
 annot.Appearance.Rollover.Add(
  AppearanceSet.DefaultAppearanceName,
  MakeAppearance(
   bounds,
   PdfColorFactory.FromRgb(.25, .25, .25),
   PdfColorFactory.FromRgb(1, 1, .7),
   doc.Resources));
 page.Annotations.Add(annot);
 doc.Save("simpleannot6.pdf");
}
```



Make a sticky note annotation

This C# code sample shows how to make a closed StickyNoteAnnotation with a "Help" icon.

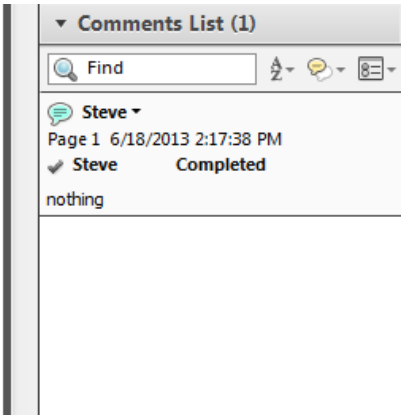
```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

StickyNoteAnnotation sticky = new StickyNoteAnnotation(new PdfBounds(144, 400, 72, 72),
 "note text here", new PdfBounds(156, 420, 100, 100));
sticky.Color = PdfColorFactory.FromRgb(1, 1, .8);
sticky.IconName = StickyNoteAnnotation.HelpIconName;
page.Annotations.Add(sticky);
doc.Save("simpleannot7.pdf");
```

Add a review state to a sticky note

This C# sample shows how to add review conditions to a Sticky Note annotation.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;
```

```
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

StickyNoteAnnotation sticky1 = new StickyNoteAnnotation(new PdfBounds(72, 600, 72, 72),
 "nothing", new PdfBounds(156, 420, 100, 100));
sticky1.IconName = StickyNoteAnnotation.CommentIconName;
sticky1.Color = PdfColorFactory.FromRgb(0, 1, .8);
sticky1.AuthorName = "Steve";
page.Annotations.Add(sticky1);

StickyNoteAnnotation sticky = new StickyNoteAnnotation(new PdfBounds(144, 600, 72, 72),
 "Completed set by steve hawley", new PdfBounds(156, 420, 100, 100));
sticky.Color = PdfColorFactory.FromRgb(1, 1, .8);
sticky.IconName = StickyNoteAnnotation.CommentIconName;
GeneralReview generalReview = new GeneralReview();
generalReview.CurrentState = GeneralReview.CompletedStateIndex;
sticky.ReviewProcess = generalReview;
sticky.InReplyTo = sticky1;
sticky.Hidden = true;
sticky.AuthorName = "Steve";

page.Annotations.Add(sticky);
doc.Save("simpleannot8.pdf");
```



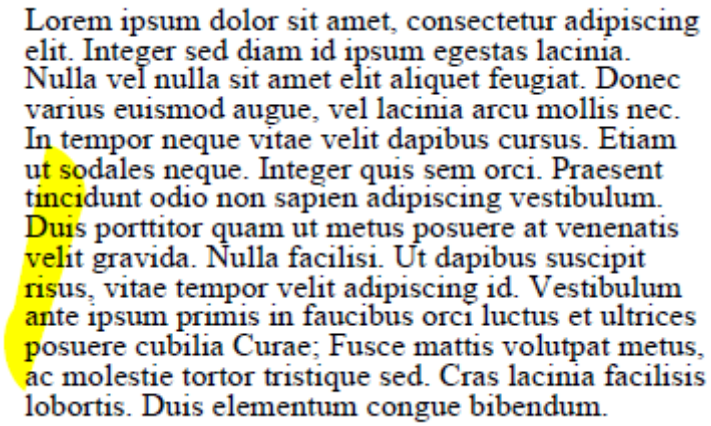
Make a highlight annotation

Highlight annotations are represented by a set of quadrilaterals. They are not directly associated with any text on the page. Any correspondence with text on the page must be made by the creation software.

The following C# code creates a highlight association.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "...lorem ipsum text...");
page.DrawingList.Add(box);

TextMarkupAnnotation textMarkup = new TextMarkupAnnotation(TextMarkupKind.Highlight);
textMarkup.Color = PdfColorFactory.FromRgb(1, 1, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
```

```
page.Annotations.Add(textMarkup);
doc.Save("simpleannot10.pdf");
```

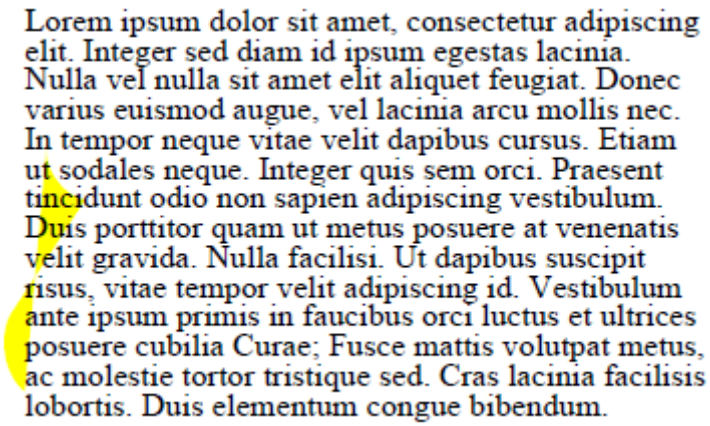


Make a bow tie annotation

When the mark up type is changed to an underline, you can see where the line is drawn relative to the quadrilateral. For underline, it is oriented towards the logical bottom which is the edge from the first point to the second point.

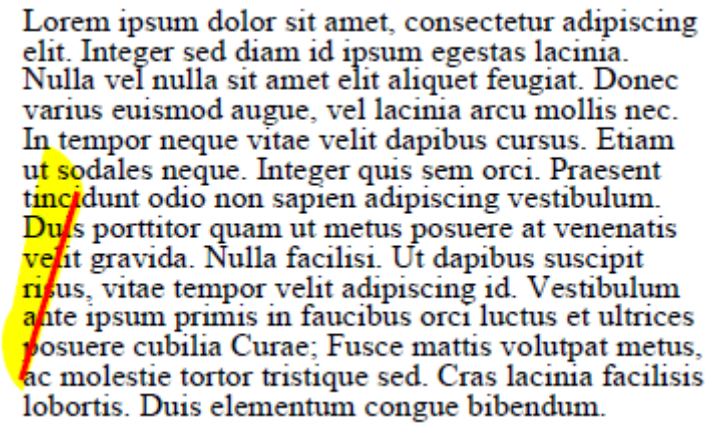
Quadrilaterals may look unusual if the points are ordered differently. If the quadrilateral is a simple rectangle, the first point is the lower left, the second point is the lower right, the third point is the upper right, and the last point is the upper left. By swapping the second and third points, you will get a "bowtie" shape, as shown in the following C# sample.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "...lorem ipsum text...");
page.DrawingList.Add(box);

TextMarkupAnnotation textMarkup = new TextMarkupAnnotation(TextMarkupKind.Highlight);
textMarkup.Color = PdfColorFactory.FromRgb(1, 1, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 80, 500, 94, 480, 68, 440));
page.Annotations.Add(textMarkup);
doc.Save("simpleannot11.pdf");
```



When the mark up type is changed to an underline, you can see where the line is drawn relative to the quadrilateral. For underline, it is oriented towards the logical bottom which is the edge from the first point to the second point, as shown in this C# sample.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "...lorem ipsum text...");
page.DrawingList.Add(box);

TextMarkupAnnotation textMarkup = new TextMarkupAnnotation(TextMarkupKind.Highlight);
textMarkup.Color = PdfColorFactory.FromRgb(1, 1, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
page.Annotations.Add(textMarkup);

textMarkup = new TextMarkupAnnotation(TextMarkupKind.Underline);
textMarkup.Color = PdfColorFactory.FromRgb(1, 0, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
page.Annotations.Add(textMarkup);
doc.Save("simpleannot12.pdf");
```

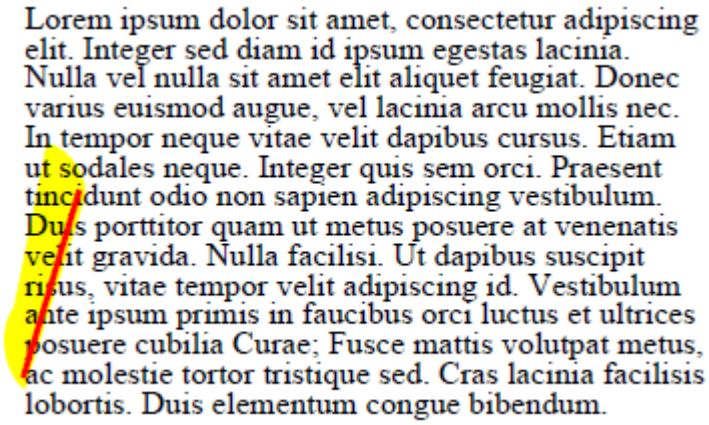


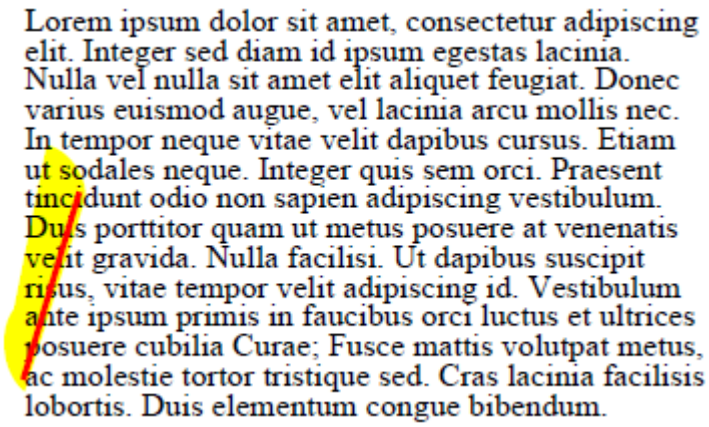
If the TextMarkupAnnotation is constructed with TextMarkupKind.StrikeOut, the line will run midway between the top and bottom edges. If it is constructed with Squiggly, a zig-zag line will be drawn along the bottom edge.

Show the underline location relative to a highlight annotation

C#

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "...lorem ipsum text...");
page.DrawingList.Add(box);

TextMarkupAnnotation textMarkup = new TextMarkupAnnotation(TextMarkupKind.Highlight);
textMarkup.Color = PdfColorFactory.FromRgb(1, 1, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
page.Annotations.Add(textMarkup);

textMarkup = new TextMarkupAnnotation(TextMarkupKind.Underline);
textMarkup.Color = PdfColorFactory.FromRgb(1, 0, 0);
textMarkup.Regions.Add(new PdfQuadrilateral(72, 410, 94, 480, 80, 500, 68, 440));
page.Annotations.Add(textMarkup);
doc.Save("simpleannot12.pdf");
```



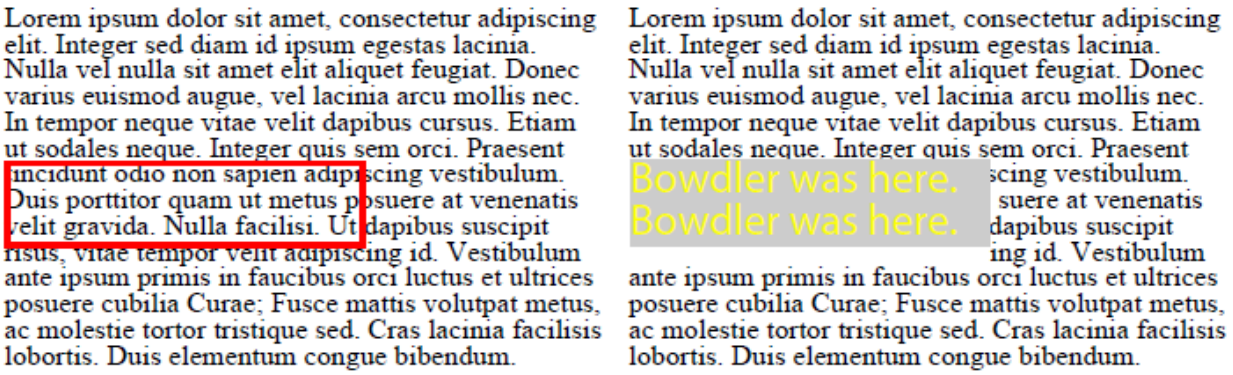


Set a redaction area

The RedactionProposalAnnotation is used to set an area for later redaction by the viewer. The annotation itself does not remove content from the document but instead requires the viewing application to perform that task. This example shows how a redaction can be placed with custom text to show when the redaction has been applied.

The following C# code places a redaction with redaction text.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.EmbedGeneratedContent = false;

PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

PdfTextBox box = new PdfTextBox(new PdfBounds(72, 400, 250, 150), "Times-Roman", 12,
 "...lorem ipsum text...");
page.DrawingList.Add(box);

RedactionProposalAnnotation redaction = new RedactionProposalAnnotation(new
 PdfBounds(72, 450, 150, 36));
redaction.Color = PdfColorFactory.FromRgb(1, 0, 0);
redaction.DefaultTextAppearance.FontSize = 18;
redaction.DefaultTextAppearance.StrokeColor = PdfColorFactory.FromRgb(1, 1, 0);
redaction.OverlayText = "Bowdler was here.";
redaction.IsOverlayTextRepeated = true;
redaction.RedactionInteriorColor = PdfColorFactory.FromRgb(.8, .8, .8);

page.Annotations.Add(redaction);
doc.Save("simpleredact2.pdf");
```

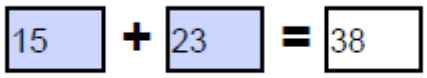


Use JavaScript to calculate values

PDF documents can contain form fields for user data entry. Using JavaScript, you can create actions to attach to actions to calculate values of make other dynamic changes to the document. For more information, see the JavaScript for Acrobat API Reference.

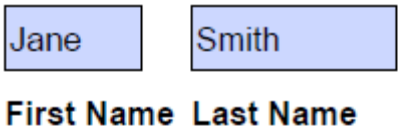
The following C# example uses the built-in function AFSimple_Calculate, which is provided by Adobe Acrobat (formerly, this was in the AForm.js file, but has been precompiled into byte code). Note that the sum field is marked read-only so that it will only show the sum.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

TextWidgetAnnotation tw = new TextWidgetAnnotation(new PdfBounds(72, 500, 36, 24),
 "Addend1", "0");
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);

PdfTextLine tl = new PdfTextLine("Helvetica-Bold", 20, "+", new PdfPoint(114, 506));
page.DrawingList.Add(tl);

tw = new TextWidgetAnnotation(new PdfBounds(130, 500, 36, 24), "Addend2", "0");
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);

tl = new PdfTextLine("Helvetica-Bold", 20, "=", new PdfPoint(172, 506));
page.DrawingList.Add(tl);

tw = new TextWidgetAnnotation(new PdfBounds(188, 500, 36, 24), "Sum", "0");
tw.IsFieldReadOnly = true;
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);
tw.AdditionalActions.OnFieldRecalculating.Add(new
 PdfJavaScriptAction("AFSimple_Calculate(\"SUM\", new Array (\"Addend1\",
 \"Addend2\"));"));
doc.Form.FieldCalculationSequence.Add(tw);

doc.Save("simplesum.pdf");
```



Similarly, you can use the contents of fields together to join data. For example, if you wanted to create a signable document that contained fields for the user's first and last names with a place to display their entire name you could make a read-only full name field which takes its values from the other fields

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

TextWidgetAnnotation tw = new TextWidgetAnnotation(new PdfBounds(72, 500, 50, 24),
 "First", "");
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);

PdfTextLine tl = new PdfTextLine("Helvetica-Bold", 12, "First Name", new PdfPoint(72,
 480));
page.DrawingList.Add(tl);

tw = new TextWidgetAnnotation(new PdfBounds(140, 500, 75, 24), "Last", "");
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);

tl = new PdfTextLine("Helvetica-Bold", 12, "Last Name", new PdfPoint(140, 480));
page.DrawingList.Add(tl);

tw = new TextWidgetAnnotation(new PdfBounds(72, 200, 200, 24), "Full", "0");
tw.IsFieldReadOnly = true;
page.Annotations.Add(tw);
doc.Form.Fields.Add(tw);
tw.AdditionalActions.OnFieldRecalculating.Add(new PdfJavaScriptAction("var
 fname = this.getField(\"First\").value + \" \" + this.getField(\"Last\").value;
 this.getField(\"Full\").value = fname;"));
doc.Form.FieldCalculationSequence.Add(tw);

SignatureWidgetAnnotation sig = new SignatureWidgetAnnotation(new PdfBounds(72, 230,
 200, 40), "Signature", null, null);
page.Annotations.Add(sig);
doc.Form.Fields.Add(sig);

doc.Save("simplenamer.pdf");
```



PDF Forms

PDF Forms are a mechanism within PDF to display information and provide interaction and data collection facilities. In the PDF Specification, these are referred to AcroForms. A PDF Form is a hierarchical collection of fields that represent the form data as well as some information to indication calculation order and general field appearance characteristics.

Fields are any object that implements the interface IFormElement, this interface defines core characteristics that are common to all fields, but in practice there are two broad types of fields: nodes and leaves. A node can have child fields and a leaf can have no child fields. In DotPdf, all leaves will be a subclass of BaseWidgetAnnotation and all nodes will be BaseFormField.

In the PDF specification, certain properties in a form field will be inherited from its parent. DotPdf does not support this directly. When a form is read in, the inheritance is flattened, but projecting parent properties onto their children. Upon writing, the properties are written directly from each field. It is the client's responsibility to enforce the effect of inheritance.

PdfForm

PdfForm is the object that represents a form for data collection and all its elements. If a document has a PdfForm, it will be accessed through the Form property of a PdfGeneratedDocument object. Through this object, you can access the fields in a document and their values (if any). The form also contains properties that define default appearances for text in the fields as well as information regarding digital signatures.

ℹ The Form property in a PdfGeneratedDocument is null by default. To create a form, you need to assign a new PdfForm object to this property. PdForm objects can be moved from one document to another, but care must be taken in the process because the leaf nodes of a PdfForm tree are all BaseWidgetAnnotation obects and therefore must also be placed on appropriate pages in the target document. Further, the PdfForm and its form fields may refer to JavaScript methods that are defined in the source document's GlobalJavaScript actions which must also be moved to the target document. It is strictly the client's responsibility to

The process of making a new form from scratch can be as simple as making a PdfForm object and assigning it to a PdfGeneratedDocument then putting fields in the form and on the pages of the document. However, PdfForm objects can represent a tree of hierarchical fields. In order for the hierarchy to be properly represented, each parent node will contain a collection of child nodes. Each child should also have a reference to its parent. Since many operations may be performed before putting a form element in a parent collection, DotPdf allows the client code to set the parent-child and child-parent relationships. There is a utility method in PdfForm called EnforceParentage() which will descend the tree and ensure that the relationship is correct. Be aware that if you depend on any particular form field's FieldFullName to be correct, the parentage must be set correctly.

ℹ When you save a PdfGeneratedDocument which contains a PdfForm, EnforceParentage() will get called automatically. The form will also be checked for cycles and other field relationship issues. If there are issues that cannot be repaired, DotPdf will throw an exception.

The following C# code creates a simple form.

```
public class WidgetPagePair {
 public BaseWidgetAnnotation Field { get; set; }
 public int PageIndex { get; set; }
}
// ...
public void PlaceFields(PdfGeneratedDocument doc, IEnumerable<WidgetPagePair> pairs)
{
 foreach (WidgetPagePair pair in pairs)
 {
  PdfGeneratedPage page = doc.Pages[pair.PageIndex] as PdfGeneratedPage;
  if (page == null) continue;
  if (doc.Form == null) doc.Form = new PdfForm();
  if (!page.Annotations.Contains(pair.Field))
   page.Annotations.Add(pair.Field);
  doc.Form.Fields.Add(pair.Field);
 }
}
```

Node form fields

PDF forms may represent a tree of form information. For example, you may want to collect similar information in different places, but want to use similar names for the actual data fields. You can do this by having a tree structure to your form. For example, you might have a parent node named "Contact" with a child named "Phone" that has three children named "Work," "Home," "Mobile," each with a child named "Number." "Contact" might have another child named "Address" with children named "Work" and "Home," each with children named "Street," "City," "State," and "Zip." In this way, the names of the leaves can be the same and can be treated generically by consuming code.

If the child and parent relationships of the fields are enforced, the full name of the phone number fields would be Contact.Phone.Work.Number, Contact.Phone.Home.Number, and Contact.Phone.Mobile.Number.

In DotPdf, there are several types of node form fields. Each is typed against what its *expected* children would be. For example, a TextFormField would expect to have children that are either TextFormField or TextWidgetAnnotation and a PushButtonFormField would expect to have children that are either PushButtonFormField or PushButtonWidgetAnnotation. If a form field is expected to have heterogeneous children, it is best to use a GenericFormField.

All form fields, whether they are node or leaf form fields will implement the interface IFormElement. This element defines the properties and behaviors of a PDF form field. A node form field can have children and will therefore have a valid ChildFields property, whereas a leaf form field will always have a null ChildFields property.

ⓘ While the PDF specification does not forbid that a CheckBoxFormField having non-CheckBox children, it is likely the field inheritance in the final PDF will do unexpected things. When a document with a PdfForm is saved, DotPdf will flag and optionally repair fields that have mismatched children by substituting the appropriate form field type or a GenericFormField if the children are heterogeneous.

RadioButtonWidgetAnnotations will not function as a group without a parent. The RadioButtonFormField comes with a set of static factory methods for making a RadioButtonFormField and correctly constructing and associating a set of RadioButtonWidgetAnnotations with that field. When accessing the "value" of a radio set, it is more common to look at the parent field rather than all of the children to determine the current value.

Leaf form fields

In PDF forms, leaf form fields are form elements that can have no children and in nearly all cases contain the actual data of a field value. In DotPdf, all leaf form fields are implemented as subclasses of BaseWidgetAnnotation. For specifics of using widgets annotations, see Widget annotations.

Visiting nodes

While it's straight forward to lop over all the nodes within a PdfForm object, DotPdf provides a number of utility methods for enumerating through the nodes in a form. The main mechanism for doing this is via the FormVisitor object, which provides methods for visiting each of the nodes in

breadth first and depth first order as well as specializations for visiting only BaseWidgetAnnotation objects.

Each of the methods returns IEnumerable<IFormElement> or IEnumerable<BaseWidgetAnnotation>.

The following C# code converts a PdfForm to XML.

```
private static XDocument ToXml(PdfForm form)
{
 XDocument xdoc = new XDocument(new XElement("fields",
  from widget in FormVisitor.WidgetsDepthFirst(form)
  select new XElement("field",
   new XAttribute("bounds", String.Format("{0} {1} {2} {3}",
    widget.Bounds.Left, widget.Bounds.Bottom, widget.Bounds.Width,
 widget.Bounds.Height)),
   new XAttribute("type", TypeFromWidget(widget)),
   widget.FieldFullName != null ? new XAttribute("name", widget.FieldFullName) : null,
   widget.ValueAsString != null ? new XAttribute("value", widget.ValueAsString) : null,
   widget.DefaultValueAsString != null ? new XAttribute("default",
 widget.DefaultValueAsString) : null
   )));
  return xdoc;
}

private static string TypeFromWidget(BaseWidgetAnnotation widget)
{
 if (widget is TextWidgetAnnotation) return "text";
 if (widget is CheckboxWidgetAnnotation) return "check";
 // etc...
 return "unknown";
}
```

Form actions

There are two global form actions available: reset and submit. Upon executing a PdfResetFormAction, all fields or all specified fields will be reset to their default value. Upon executing a PdfSubmitFormAction, all fields or all specified fields (and other data) will be submitted to a URI.

The following C# code creates a form with a field reset.

```
PdfGeneratedDocument doc = new PdfGeneratedDocument();
doc.Form = new PdfForm();
PdfGeneratedPage page = doc.AddPage(PdfDefaultPages.Letter);

TextWidgetAnnotation color = new TextWidgetAnnotation(new PdfBounds(36, 700, 400, 24),
 "color", "Orange");
color.TextValue = color.DefaultTextValue;
doc.Form.Fields.Add(color);
page.Annotations.Add(color);

PdfTextLine label = new PdfTextLine("Helvetica", 14, "Favorite Color:", new
 PdfPoint(36, 730));
page.DrawingList.Add(label);

PushButtonWidgetAnnotation reset = new PushButtonWidgetAnnotation(new PdfBounds(36,
 670, 100, 25), "Reset", null, null);
reset.ClickActions.Add(new PdfResetFormAction());
page.Annotations.Add(reset);
```

```
doc.Save("resetform.pdf");
```

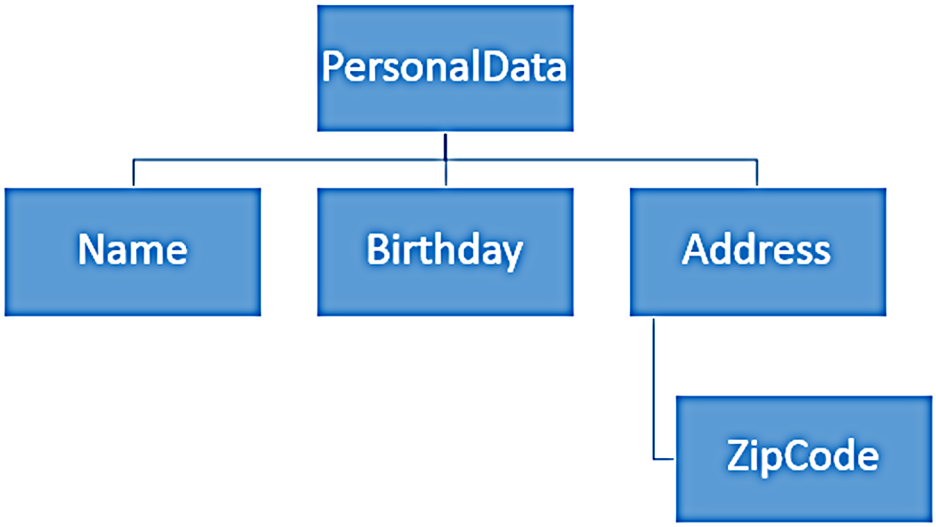
In this example, the reset button was added to the page, but not to the form. This will prevent it from being subjected to the reset. This could also have been accomplished by putting the button in the form and adding it to the Fields property of the PdfResetFormAction. Since PushButtonFields and PushButtonWidgetAnnotation objects do not have a value, resetting them to doesn't make sense.

Merge PDF forms

An interactive form - sometimes referred to as an AcroForm - is a collection of fields for gathering information interactively from the user. Each field is associated with one or more widget annotations that define its appearance on the page.

For example, a `Date` field could be associated with multiple widget annotations, each of which could be placed on different pages. If one of these annotations is filled, the rest of them are automatically given the same value.

The field may have a partial field name. A fully qualified field name is constructed from the partial field name of the field and all of its ancestors.



The fully qualified field name for `ZipCode` is `PersonalData.Address.ZipCode` (a period (.) is used as a separator for fully qualified names). If the form contains fields with identical fully qualified names, the annotations of these fields are invalid.

In order to merge PDF documents that contain forms, it is necessary to merge forms as well. If forms in different documents contain fields with the same name, the user should either rename or merge them.

The `PdfGeneratedDocument` class provides tools for merging PDF documents with forms.

Import pages

`PdfGeneratedDocument` contains the `ImportPages()` method. This method loads pages from an external PDF document. The `ImportPages()` method has two arguments: `path to file` and `import options`.

The `ImportOptions` object has properties to specify passwords and `RepairOptions` for opening external documents, indexes of pages, and inserting at the specified location. If the current `PdfGeneratedDocument` and external PDF document contain one or more forms, use `ImportOptions.FormFieldsConflictHandler` to define the merge behavior.

Merge forms

If the current `PdfGeneratedDocument` and external PDF document contain forms, they should be merged. `FormFieldsConflictHandler` is called if there are fields from external and current documents with identical fully qualified names. This handler must resolve conflicts by choosing one of the following conflict resolution strategies (defined as `FormFieldsConflictResult` enum):

- `KeepCurrentFieldAndMergeChildren` - only keep the field from the current form and merge children with the external field's children.
- `KeepExternalFieldAndMergeChildren` - only keep the field from the external form and merge children of the external and current fields.
- `KeepBoth` - keep both fields. In this case one of the fields should be renamed.

Sample for merging fields with same type and renaming fields with different types

```
public void CombinePdfForms()
{
    using (var stm = File.OpenRead(@"TwoPagesForm.pdf"))

    using (var genDoc = new PdfGeneratedDocument(stm))
    using (var streamForImport = File.OpenRead(@"docWithForms.pdf"))
    {
        genDoc.ImportPages(streamForImport, new ImportOptions
        {
            FormFieldsConflictHandler = ResolveFormFieldsConflict
        });

        using (var outStm = File.Create("CombinedForm.pdf"))
            genDoc.Save(outStm);
    }
}

private void ResolveFormFieldsConflict(object s, FormFieldsConflictEventArgs a)
{
    if (a.AreFieldTypesEqual)
    {
        a.ConflictResolution =
            FormFieldsConflictResult.KeepCurrentFieldAndMergeChildren;
        return;
    }
    // generate new name for field
    a.ExternalField.FieldName = "new" + a.ExternalField.FieldName;
    a.ConflictResolution = FormFieldsConflictResult.KeepBoth;
}
```

Default merging

If the `FormFieldsConflictHandler` property is not set, all fields with identical fully qualified names and the same field types are merged:

- The current field of the current form is not changed.
- All child fields of the external field are added to the children collection of the current field.

If fields with equal fully qualified names have different field types `PdfException` is thrown with the following message:

Fields with different types have identical fully qualified names. Use `ImportOptions` to specify a conflict resolving handler and rename one of the fields.

DotPdf repair

Starting with DotPdf version 10.4, DotPdf includes the facility to detect and repair damaged PDF documents. These repairs include:

- Repairing dictionary objects that have missing required values.
- Repairing dictionary objects that have incorrect optional values.
- Repairing array objects that have syntactically incorrect values or references to non-existent objects.
- Repairing stream objects that have incorrect length values or are missing the endstream keyword or have incorrect line-ending placement.
- Repairing damaged or incorrect cross-reference tables.
- Repairing incorrect PDF file versions.
- Restoring "orphaned" pages.
- Substitute blank pages for unreadable pages.

In most cases, client code will use the repair mechanism as is, but it is possible to hook into the repair process to help inform decisions for repairs to the document and its contents. This can include allowing or disallowing repairs that may remove content from the document or otherwise change the document's appearance, reporting errors and repairs as they happen, or providing an alternative to the value that will be used to replace an incorrect or missing value in a repair.

DotPdf repair process

Generally speaking, DotPdf avoids reading entire PDF documents at any one time. For example, when you create a PdfDocument object from an existing PDF document, DotPdf only reads the document metadata and enough information to determine how many pages are in the document as well as the orientation of each page.

No other information will be read from the PDF document until PdfDocument.Save() is called. At this point only the "live" objects in the PDF document will be read. For example, if you open a multipage PdfDocument and remove one or more pages from the document then save, the pages

you removed (and all the objects they reference, provided they aren't referenced by other pages) are no longer live and won't be read.

By contrast, PdfGeneratedDocument reads in substantially more objects when constructed from an existing PDF.

Error detection happens at three possible points in time: when a PDF document is initially opened, when PDF objects are read, and when PDF objects are written. When errors are detected, they are reported and a request is made to accept the error for potential repair. If errors are not accepted for potential repair, DotPdf will throw a PdfException. Often an entire PDF object has been read, any errors will be checked for repair. An error will be repaired if the repair system is configured to perform that class of repair and if the consequences of the repair are acceptable. If the proposed repair and consequences are acceptable, it will be performed. After all repairs have been completed for the object, if there were any unperformed repairs, a PdfException is thrown, otherwise processing continues.

When a PdfException is thrown during repair, it may get caught inside DotPdf and induce further repairs. If it was not caught, it will be passed on to client code and the repair has failed.

One exception to the process is the repair of the document cross-reference table. The cross-reference table is a structure within a PDF that is used to locate all the other objects within the file. If the cross-reference table is damaged or can't be located, then the cross-reference table will be rebuilt by scanning the entire contents of file. If this error is not repaired, nothing else can be done with the file.

Detect errors

In general, any time any content is loaded or saved from a PDF document an a PdfException is thrown, the document is a candidate for repair. DotPdf defines two types of exception, PdfException and PdfParseException. The latter inherits from PdfException and is thrown when DotPdf is unable to locate the document cross reference table or the cross reference table is damaged.

The following C# code detects errors.

```
public bool PdfHasErrors(Stream inPdf)
{
    Stream outStm = GetTemporaryStream();
    try {
        PdfDocument doc = new PdfDocument(inPdf);
        doc.Save(outStm);
        return false;
    }
    catch (PdfException) {
        return true;
    }
    finally {
        RemoveTemporaryStream(outStm);
    }
}
```

It should be noted that this will be a potentially expensive process as the entire document will be scanned. In a workflow environment, it may be more convenient to catch PdfException when a file is being processed, mark it as a failure, repair it later and then resubmit it for processing.

Errors can also be fixed as part of the normal course of events. Be aware that not all errors can be repaired and repairing some errors may remove or otherwise change visual content in a PDF.

It is never acceptable to blindly copy a repaired document over the original document.

Repair errors

In order to request that errors should be repaired in a PDF in the course of processing it, construct a PdfDocument object or a PdfGeneratedDocument passing in a RepairOptions object. Passing in null is equivalent to performing no repair.

The RepairOptions object contains sets of properties that determine if and in some cases how errors will be repaired. It also contains event objects that an application can use to track errors.

The default values in RepairOptions represent a good balance of repairing problems without excessively damaging the appearance or content of the document.

The following C# code repairs errors.

```
RepairOptions repairOptions = new RepairOptions();
try
{
 PdfDocument doc = new PdfDocument(null, null, pdfStream, null, repairOptions);
 doc.Save(outputStream);
}
catch (PdfException)
{
 // clean up outputStream
}
finally
{
 if (repairOptions.StructureOptions.RepairedStoredStream != null)
  repairOptions.StructureOptions.RepairedStoredStream.Dispose();
}
```

This example opens a PDF document (with no passwords) and copies it to the output, repairing errors. The try/catch is necessary since repairs may fail and client code should manage the output stream since it may contain partial/invalid PDF. The finally clause is necessary since repairs may require rebuilding the entire file. Under such circumstances, a temporary file will be created. The call to Dispose() will remove the temporary file.

As a convenience, PdfDocument contains several flavors of the static method Repair() which is equivalent to the above code except with no catch block:

```
try {
 PdfDocument.Repair(pdfStream, outputStream, new RepairOptions());
}
catch (PdfException)
{
 // clean up outputStream
}
```

Repair events

In order to provide feedback about what is happening during the repair process, the RepairOptions object contains the following events:

- ProblemEncountered: Fired when a problem is first encountered.

- ProblemRepaired: Fired when a problem has been repaired.
- ProblemSkipped: Fired when a problem was skipped during repair.

Each event will include the ProblemEventArgs object. Within the ProblemEventArgs is a property named Problem of type BaseProblem. This object describes the nature of the problem in the Description property and possible consequences of enacting the repair in the Consequences property.

In the case of DotPdf, the Consequences object will be of type PdfRepairConsequences. This object contains information about the severity of the problem as well as a description of what may happen if the repair is enacted.

```
int problemsEncountered = 0;
int problemsRepaired = 0;
int problemsSkipped = 0;
RepairOptions repairOptions = new RepairOptions();
repairOptions.ProblemEncountered += (s, e) => problemsEncountered++;
repairOptions.ProblemRepaired += (s, e) => problemsRepaired++;
repairOptions.ProblemSkipped += (s, e) => problemsSkipped++;
PdfDocument.Repair(pdfInStream, pdfOutStream, options);
Console.WriteLine("The document had {0} errors, {1} repaired, {2} skipped.",
     problemsEncountered, problemsRepaired, problemsSkipped);
```

This C# code sample shows how to track the number of errors and repairs in a PDF document.

The event mechanism is separate from the problem selection process. No filtering is done with events.

Repair filtering

The RepairOptions object in DotPdf has two levels of filtering, the first is when a problem is encountered. This is to decide if the problem should be accepted for repair. The second is at repair time to choose if a repair will be enacted. An application could choose to filter based on the type of problem or on the severity of the consequences or on the number of problems encountered.

The properties are named ProblemSelectorand RepairSelector. It is not necessary to set either. Setting them to null (default) will instruct DotPdf to ignore them.

Both delegates return enumerated types which include the value Default, which is an indication that DotPdf should take its default action.

The following C# code sets RepairOptions to filter based on severity.

```
RepairOptions options = new RepairOptions();
options.RepairSelector = (sender, problem) =>
    {
        PdfRepairConsequences consequences = problem.Consequences as
 PdfRepairConsequences;
        if (consequences == null) return RepairAction.Default;
        if ((int)consequences.Severity > (int)Severity.Serious)
            return RepairAction.NoRepair;
        return RepairAction.Repair;
    };
```

ℹ You can get this same behavior without a filter by setting RepairOptions.MaximumAllowableSeverity.

Structure options

In the RepairOptions object there is a property named StructureOptions of type StructureRepairOptions. This object contains a set of properties that are used to control what structural elements within a PDF will be repaired.

Property name	Property type	Default value	Description
RebuildCross ReferenceTable	bool	false	If set to true, a PDF with a damaged cross-reference table will have the entire file rebuilt with a correct cross-reference table. Upon completion, the property RepairedStoredStream, if non-null will be set to the StoredStream that was used for a temporary file. This type of repair may be expensive in terms of time and storage. It is appropriate to use this repair if opening a PDF throws a PdfParseException.
StoredStreamProvider	IStoredStreamProvider	TempFileStreamProvider	This object is used to create a StoredStream object that is used to hold the contents of a PDF file that had to have its cross-reference table rebuilt.
RepairedStoredStream	StoredStream	null	After repairing a cross reference table, this property will be non-null and will contain the Stream that holds the repaired PDF. ❗ Dispose this object after you are done with the document.

Property name	Property type	Default value	Description
RestoreOrphanedPages	bool	true	If set to true and RebuildCrossReference Table is set to true, then any pages found in the document during rebuilding that aren't part of the document's page collection will be appended to the end of the page collection.
CreateBlankPageIf NoPagesFound	bool	true	PDF documents must have at least one page. If set to true and a document contains no pages or nothing but damaged pages, a blank letter-sized page will be added to the page collection. Although the document will have no content on pages, it is still may be possible to access metadata, forms and form data, scripts, and other non-page content.
CorrectInvalidData StreamLengths	bool	true	If set to true, embedded data stream objects with incorrect lengths will be repaired by measuring the actual length of the stream.
RepairNameTrees	bool	true	Name trees are structures stored within PDF documents that hold information associated with names. For example, there is a name tree that is used to hold JavaScript objects that are used globally within the document. If set to true, damaged name trees will be repaired.
DuplicateNameTree EntryRepairAction	DuplicateNameTree EntryRepairAction	None	Determines an action to take when duplicate name entries are found. None is equivalent to ignoring any newer duplicates. Other options include remove the previous one or renaming either.

Property name	Property type	Default value	Description
AllowPartialNameTrees	bool	true	If an unrecoverable error happens while reading a name tree, this will allow whatever name tree entries have already been read to be passed on. Partial name trees may result in later errors when links try to find missing named destinations or named JavaScripts.
NameSelector	NameReplacer	null	Given a DuplicateNameTree EntryRepairAction that requires renaming an element, this property will be used to rename the duplicate entry. This delegate will be passed the name to rename and a list of all other names in the tree. This delegate should return a new name that is not contained within the list.

Array options

When elements of arrays are damaged, this set of options will be used to determine how to repair the elements

Property name	Property type	Default value	Description
RepairDamaged Elements	bool	true	If set to true, DotPdf will attempt to repair damaged elements PDF arrays. This is done, be default, by putting in a reasonable default for the item.
ElementReplacer	ArrayElementReplacer	null	This delegate, when non-null, will be called by DotPdf to create an appropriate value for a damaged array element.

Property repair

Most of the internals of PDF documents consist of Dictionary objects that have property names associated with values. The PDF specification defines the content and meaning of elements within

dictionaries. For example, a dictionary may have a property that is required and the dictionary is incorrect if the property is missing.

DotPdf has a mechanism for tracking the meanings and settings of PDF dictionaries and automatically determines appropriate ways to repair them if they are damaged or missing. It is possible to override the default behaviors in DotPdf by setting the PropertyValueReplacer in the PropertyRepairOptions object.

Note that it is not possible for client code to make appropriate substitutions for all damaged dictionary properties since many dictionary properties (including the dictionaries themselves) are internal types and inaccessible to client code. Further, changes to dictionary contents typically require deep understanding of the PDF specification.

The following C# code repairs damaged URI objects.

```
RepairOptions options = new RepairOptions();
options.PropertyOptions.Replacer = UriRepairer;
//...
public bool UriRepairer(PropertyInfo property, string propertyName, object
 propertyOwner,
         object defaultValue, object fileParsedValue, object fileSuppliedValue,
         ref object replacementValue)
{
 if (property.PropertyType == typeof(Uri))
 {
  replacementValue = new Uri("http://www.mywebsite.com");
  return true;
 }
 return false;
}
```

By default, DotPdf replaces invalid URI objects with a Uri that points to http://127.0.01. This code will override that setting and replace them with http://www.mywebsite.com.

Digital signatures

Digital signing is process whereupon an electronic document can be marked so that the document's origin can be verified and changes to the document can be detected.

In PDF, there are two main operations for signing a document: certification, and signing.

Both operations involve the signature annotation widget, but the meanings of certification and signature are different.

In the case of a certification, you are placing a signature widget annotation on the document (either visible or invisible) along with a set of rules that dictate what changes are allowed to be made to the document as a whole. When a document has been certified, the person applying the certification is saying, "I declare that the content of this document is exactly what it should be at the time of the certification and you may only make the following changes..."

In the case of signing, you are placing an equivalent to a physical signature in the document and which carries the same implications of physical signing (accepting terms of a contract, verifying that information is complete and so on). The signature may dictate that other widget annotations should become locked when it has been signed.

In DotPdf, you can certify an unsigned, uncertified document and you can sign a certified or uncertified document. In addition, you can sign an already signed or unsigned document as long as there are signature annotations that are unsigned and that the document allows that.

DotPdf signs a document using the PKCS7 standard and the modification detection can be configured to use any of SHA1, SHA256, SHA384, and SHA512 as the message digest. The actual digital signature content is represented by an X509 certificate or a chain of X509 certificates.

When working with DotPdf for digital signatures, there are four main actions that are available:

1. Certifying a PDF document
2. Getting information about a signed or certified document
3. Making allowable changes to a signed or certified document
4. Signing a document

This document will be organized around each of these actions and how to do them.

Note the following:

- DotPdf only supports signing and getting information about PDF documents signed using the PKCS7 standard.
- DotPdf tries to use the .NET object RSACryptoServiceProvider to perform signing and certifying operations. This object is retrieved from the X509Certificate2 object provided by client code. Some versions of this object can not sign with anything but the SHA1 message digest algorithm. The CmsInformation object, upon construction, checks to see if the requested message digest algorithm is supported by the RSACryptoServiceProvider. If it is not, the CmsInformation object checks to see if the X509Certificate2 object can be transferred to an equivalent supported by BouncyCastle. If not, then at signing time, the signing code will either fall back to using SHA1 or will throw an exception. This behavior is controlled by the UnsupportedContentDigestAlgorithmAction property in the CmsInformation object, set by the constructor. If the X5092Certificate came from a file, such as a .pfx file and was opened requesting the ability to export the private key, then if the RSACryptoServiceProvider is unable to sign the document then BouncyCastle will be used.
- SHA1 digest method is deprecated in PDF 2.0. CmsInformation with the SHA1 digest method call causes PdfException("SHA1 PdfContentDigestMethod is not supported in PDF 2.0.") in PdfDocumentSigner on the signing PDF 2.0 document attempt.
- Future version of DotPdf digital signatures are likely to include more direct access to certificates via BouncyCastle.

Certify documents

To certify a PDF document is to apply an X509 certificate to the document and a set of rules to prevent and detect tampering with the original document. In DotPdf, this is done through the PdfDocument object or the PdfGeneratedDocument object. Both objects contain a property called DocumentCertification which describes how the document should be certified when it is saved.

The DocumentCertification property is either a PdfDocumentCertification object or a PdfGeneratedDocumentCertification object. Both objects descend from a common base class. The main properties are:

Property name	Property type	Description
IsVisible	bool	Determines if the certification should be visible in the document or not. Typically certifications are invisible, but the user should have the choice.
CmsInformation	CmsInformation	This object contains the chain of X509 digital signatures that will be used for the object. As well an information on how the digital signature will be built.
AllowedChanges	DocumentMDPAllowed Changes	Specifies what changes may be made to the document after certification • None: No changes are allowed. • FillFormsAndSigning: Only widget annotations (form fields) may be modified. • FillFormsSigningAnd Annotation: Widget annotations (form fields) may be modified and any type of annotation can be added to the document.

The real difference between the two is that the PdfDocument object is extremely light weight and can only describe the certification and how it is to be applied in very simple ways. For certifying a Pdf through PdfDocument, you can only specify the page number of the page upon which the signature annotation that will represent the certificate will appear and the bounding box that will contain it. The appearance of the signature will be the default appearance and is not changeable. Using PdfGeneratedDocument, you can have the entire suite of PDF generation tools available and the signature can have a custom appearance. Rather than calling out a page and location for the signature, you place a SignatureWidgetAnnotation on PdfGeneratedPage through its annotation collection as well as putting it in the PdfGeneratedDocumentCertification object.

Select PdfDocument or PdfGeneratedDocument for certification

Consider the following criteria when selecting between PdfDocument and PdfGeneratedDocument for certification.

Criteria	PdfDocument	PdfGeneratedDocument
Certification signature will be invisible.	X	X
Memory may be an issue on target system.	X	
Appearance of signature is important.		X

Criteria	PdfDocument	PdfGeneratedDocument
Placement of signature is important relative to other annotations.		X

Controlling changes to certified documents

You can choose a set of global rules for how the document may be used post certification. This is done by setting the AllowedChanges property of the DocumentCertification object. This setting will depend upon your needs for the document. Use this guide to help choose the appropriate setting:

Value	When to use
None	The document should never be changed in any way after it has been certified. For example, a transcription of an agreement.
FillFormsAndSigning	Only widget annotations (form fields) may be modified after certification. This is useful for creating a document that will to be signed by another party at a later date and will might have other information added to the document. For example, a permission form might contain a signature box as well as a checkbox to indicate that the signer is acting as a parent or guardian.
FillFormsSigningAndAnnotations	It will be possible to edit any and all annotations that are associated with the document (unless they have been locked by a previous signature). This setting is useful if you are creating a document that should not be modified in its content, but is under review by other people who will mark up the document with annotations.
NotSpecified	This value cannot be used in DotPdf when certifying a document nor will DotPdf generate a document with this value. It is present because it is possible to create documents with other tools that have no meaningful value for this property. One would only see this value in examining the certification settings on an existing PDF.

Get signer information

When examining a PDF document, you might want a way to display or act on information about signature or certification properties present in the document. PdfDocumentSignatureInformation provides a lightweight mechanism for accessing this information as well as additional tools present to verify the PDF-oriented aspects of the document and its contents.

DotPdf does not attempt to validate the content of the X509 certificate chain used in the document, but the objects representing the certificate chain are readily available.

In the PdfDocumentSigner object, there is a method, GetInfo() which accepts a PDF stream and optionally a password and returns a new PdfDocumentSignatureInformation object which describes the certificate and signatures, if any, that were in the supplied PDF.

PdfDocumentSignatureInformation contains the following properties:

Property name	Property type	Description
HasSignatures	bool	True if the document is contains signature widget annotations, false otherwise. If the document contains signatures, those signatures may be unsigned.
IsCertified	bool	True if the document contains a certification signature, false otherwise.
AllowedChanges	DocumentMDPAllowedChanges	If IsCertified is true, this property indicates what changes may be made to the document (if any). If IsCertified is false, this property will contain NotSpecified.
Certificate	PdfSignatureInformation	If IsCertified is true, this property will contain a PdfSignatureInformation object that describes the certificate. If IsCertified is false, this property will be null.
SignatureCount	int	Returns the total number of signature widget annotations in the document, 0 if there are none.
SignedSignatureCount	int	Returns the total number of signature widget annotations that are signed, 0 if there are none.
Signatures	IList<PdfSignatureAnnotation>	Gets a list of information about all signature widget annotations in the document. This list will contain both signed and unsigned signature widgets. There are no signatures, this list will be empty.

Property name	Property type	Description
ErrorsEncountered	IList<SignatureValidationError>	If any errors occurred in the process of retrieving the document signature information, this list will contain a description of those errors. Errors may be either PDF specification related errors or errors encountered while retrieving the signature data. Unlike PdfDocument and PdfGeneratedDocument, repair of errors within a damaged PDF are not possible because repairing the errors would invalidate any signature in the file. Errors will be marked with their severity.

Getting the PdfDocumentInformation object does not perform an exhaustive check on all signatures as that can be very time-consuming. For example, when a signature widget has been signed it may forbid changes to any (or all) other widget annotations on the page. The PdfDocumentSignatureInformaion object will *not* give feedback about this class of errors. To do that, call the Validate() method in PdfDocumentSignatureInformation, which will do an exhaustive check to ensure that no changes have been made to the document that violate the allowable changes. Validate() returns a list of SignatureValidationError describing what problems were found. Validate *does not* attempt to validate the contents of any of the X509 certificates used to sign signatures.

PdfSignatureInformation object

The PdfSignatureInformation object describes an individual signature with a PDF document. This information includes the physical location of the signature as well the X509 Certificate used with that signature. It contains the following properties:

Property name	Property type	Description
IsSigned	bool	True if the signature widget annotation associated with the PdfSignatureInformation object has been signed, false otherwise.
IsVisible	bool	True if the signature is visible on page, false otherwise. The PDF specification has multiple ways of determining if a signature widget annotation is visible. IsVisible will be false if any of those indicate that the signature is not visible.
Certification	PdfCertification	Returns an object that describes the certificate used to sign the signature widget annotation or null if it is not yet signed.

Property name	Property type	Description
PageNumber	int	The 0-based index on which the signature widget annotation can be found. Note that even invisible signatures should exist on a page.
AnnotationIndex	int	The 0-based index within the annotation collection where the signature widget annotation can be found.
SignatureIteration	int	Each time one of more signatures in an existing PDF document has been signed, all changes are encapsulated within the PDF document as a revision. This number indicates in which revision the signature has been signed. It is meaningless if IsSigned is false.
SignatureFieldName	string	This is the dot-qualified name of the signature widget annotation. In PDF the fields can be represented as a tree of fields. The name of any given field will be its name prepended by its parent name and a period character in "parent.child" format. This corresponds to the FieldFullName property of the signature widget annotation.

PdfCertification and CmsInformation

The PdfCertification object is a container for the certificate that was used to sign a given signature. Currently, it only represents X509 certificate objects, but in the future may represent other types of certificates as well. CMS is Cryptographic Message Syntax which is used to sign, digest, authenticate or encrypt information. The CmsInformation object in DotPdf contains the chain of certificates that were used to sign a document. It also contains the digest algorithm that will be used when creating a digital signature, but that property does *not* reflect the actual file content when getting information about a file at present.

For documents version PDF 2.0, certificates with ECDSA private key algorithms are supported.

Document signing operations

A document that has been certified or contains signed signatures has to be handled in a very particular way. For example, a PDF document that has been certified may not allow any changes to the document whatsoever or it may allow form fields to be filled in. Both PdfDocument and PdfGeneratedDocument operate in a way that requires them to rewrite the entire document upon doing a save operation. This type of action would completely invalidate and certificate or signed signatures. In PDF, when making changes to such a document, it is necessary to append any changes as a revision to the existing document.

DotPdf manages this class of operation through the PdfDocumentSigner object. PdfDocumentSigner in many ways is similar to PdfGeneratedDocument in that it has a representation of the PdfForm object contained within a PDF as well as the a representation of all annotations on all pages and a set of document resources.

With PdfDocumentSigner, you can add, remove, or change annotations or form fields contained within a PDF, but only if those changes are allowed by the document's certification or signatures. For example, if a field within a PDF document had been marked read-only as a side-effect of a signature being applied, then attempting to change properties in that field will generate an exception. BaseWidgetAnnotation and BaseFormField have new properties, IsReadOnlyOrFieldReadOnly and IsFieldReadOnly respectively. When that property is true, any attempt to change another public property within that object will throw an InvalidOperationException.

PdfDocumentSigner Object

A PdfDocumentSigner Object is constructed from a Stream that allows both read and write operations (an ImageOutputStream). Once constructed, the object gives you access to the annotations and fields contained within the PDF document and allows/disallows editing of those objects (depending on the permissions). When the changes are committed, they will be appended onto the supplied stream.

Note the following:

- The PdfDocumentSigner object can only commit one round of changes. If you need multiple sets of changes, you will need to construct a new PdfDocumentSigner object for each revision.
- PdfDocumentSigner appends changes to the stream supplied in the constructor. If you cannot make changes to your source PDF, it is your responsibility to make a copy first. PdfDocumentSigner will *not* make a copy for you.

The following properties are available in the PdfDocumentSigner object:

Property name	Property type	Description
Info	PdfDocumentSignatureInformation	Upon construction, PdfDocumentSigner will create a PdfDocumentSignatureInformation object that is uses (in part) to create the rest of the contents of PdfDocumentSigner. This object provides information as to what signatures are present within the document, if they are signed, and what changes are allowable to the document. For more information, see Get signer information.

Property name	Property type	Description
Resources	GlobalResources	This object is used to hold resources that are necessary for rendering new annotations or editing existing annotations (for example, Templates to use as appearances). Unlike PdfGeneratedDocument, no effort is made to import existing resources from the PDF document. Sharing or changing previous resources may produce a document that is either invalid or violates the security of previous signatures or certifications.
PagesOfAnnotations	ReadOnlyCollection <IList<BaseAnnotation>>	This collection represents the annotations on each page by using one entry for every page. Each entry in PagesOfAnnotations is a list of annotations that are on the corresponding page. If a page has no annotations, the corresponding list will be non-null, but empty. If document forbids adding or removing fields or annotations, each sub-collection will also be read-only. If a document forbids editing annotations or fields, those objects will be marked read-only and any attempt to change a property in that object will throw an InvalidOperationException. Although the top level properties in annotations are read-only, sub-objects such as AppearanceSet objects are not. Even though it appears like you can change these objects, changes to sub-elements in a read-only object will be ignored. This prevents malicious code from attempting to change the appearance of a signed signature widget annotation (for example).

Property name	Property type	Description
Form	PdfForm	The form fields in a PDF document are represented as a conceptual tree of fields such that the leaves of the tree, which will always be a sub-class of BaseWidgetAnnotation, contain the actual data. Although the tree can be built in exactly one level, it is possible to organize data in the tree such that related elements are in the same hierarchy (for example Person.Name.First and Person.Name.Last share the same general structure in the tree except for the terminal fields First and Last). If a document forbids adding or removing fields or widget annotations, each collection of child fields will be read-only. ℹ When adding or removing a widget annotation from the document via PagesOfAnnotations, it is imperative that the parallel change be made in Form.

Use signatures and certifications

See the following for instructions on using signatures and certifications.

- Retrieving Field Data
- Collecting Signature Information
- Editing Annotations and Fields During Review
- Editing Annotations, Fields, and Signature Widgets

ℹ For each of these procedures, the Append...Final() methods have a bool argument that instructs DotPdf to close the stream once the changes are made. Although closing the stream is not strictly necessary, this is there to remind you that the changes that you have made to the stream represent a final step. Any attempt to call these methods subsequently will result in a PdfException.

Customize signature appearance

In PDF the appearance of a signature widget annotation is managed through the regular annotation appearance mechanism. Any annotation may choose to associate a set of appearances with itself that will be used by PDF viewers to determine the visual styling of the object. If there is

no style present, it is up to the viewed to determine the appearance. For more details, see Skin an annotation.

The easiest way to manage the appearance of a signature annotation is to allow DotPdf to do it for you. When you create a PdfDocumentSignature object, there is a property named AutoGenerateSignatureAppearance which, when set to true, will induce DotPdf to call the method SignatureWidgetAnnotation.MakeBasicAppearance. This method generates a new Template resource and returns the name of the resource.

When this is method is called automatically, it will use the signature widget annotations Bounds, BorderColor, BackgroundColor, and DefaultTextAppearance values. If either BorderColor and BackgroundColor are null, black and white respectively will be used instead. If DefaultTextAppearance is null, DotPdf will use 12 point Helvetica.

When you call the method yourself, you can set any of these values as you want and can also disable the default logo, if you so choose.

Beyond these customizations, you can also retrieve the automatically generated Template resource and edit it directly as well. You can also choose to not use the automatically generated appearance and make your own from scratch.

Certify a document with PdfDocument

This sample certifies an existing PDF with an X509 certificate. The certificate will be invisible.

If you do not choose to provide a diget method to the CmsInformation constructor, it will use SHA256 by default.

ℹ While the SHA1 digest method is available, the PDF specification does not recommend its use.

The following C# code certifies a document.

```
public void CertifyDocument(Stream inPdf, Stream outPdf,
              X509Certificate2Collection certChain, PdfContentDigestMethod
 digestMethod)
{
    PdfDocument doc = new PdfDocument(inPdf);
    CmsInformation cmsInfo = new CmsInformation(certChain, digestMethod,
                 UnsupportedContentDigestAlgorithmAction.FallBackToSHA1);
    doc.DocumentCertification = new PdfDocumentCertification(cmsInfo,
 DocumentMDPAllowedChanges.None,
        false, 0, PdfBounds.Empty);
    doc.Save(outPdf);
}
```

Determine if a document is certified or signed

This C# sample opens an existing PDF and determines if it has been signed or certified.

```
public bool DocumentIsSigned(Stream inPdf)
{
    PdfDocumentSignatureInformation info = PdfDocumentSigner.GetInfo(inPdf);
    if (info.ErrorsEncountered.Count > 0)
        ReportErrors(info.ErrorsEncountered);
    return info.IsCertified || info.SignedSignatureCount > 0;
}
```

Fill fields of a certified document

This C# sample fills in text fields in a previously signed PDF document.

```
public void FillFields(Stream inPdf, Dictionary<string, string> fieldNamesAndValues)
{
    PdfDocumentSigner doc = new PdfDocumentSigner(inPdf, null);
    if (doc.Info.AllowedChanges == DocumentMDPAllowedChanges.None)
        throw new Exception("Document may not be changed.");
    foreach (BaseWidgetAnnotation anno in FormVisitor.WidgetsBreadthFirst(doc.Form)) {
        string value = null;
        TextWidgetAnnotation txAnno = anno as TextWidgetAnnotation;
        if (txAnno == null || anno.IsReadOnlyOrFieldReadOnly)
             continue;
        if (fieldNamesAndValues.TryGetValue(txAnno.FieldFullName, out value))
           txAnno.TextValue = value;
    }
    doc.AppendChangesFinal(true); // close the stream
}
```

Sign a document with an existing signature

This C# sample signs a preexisting signature widget annotation in a PDF document. Specifically, it signs the first unsigned annotation in the document.

```
private SignatureWidgetAnnotation FindFirstSig(PdfDocumentSigner doc)
{
    SignatureWidgetAnnotation sig = null;
    for (int i = 0; i < doc.PagesOfAnnotations.Count; i++) {
        for (int j=0; j < doc.PagesOfAnnotations[i].Count; j++) {
            sig = doc.PagesOfAnnotations[i][j] as SignatureWidgetAnnotation;
            if (sig != null && !sig.IsSigned) return sig;
        }
    }
    return null;
}
public void SignFirstSignatureWidget(Stream stm, CmsInformation sigData)
{
    PdfDocumentSigner doc = new PdfDocumentSigner(stm, null);
    SignatureWidgetAnnotation sig = FindFirstSig(doc);
    if (sig == null)
        throw new Exception("No signature found.");
    PdfDocumentSignature docsig = new PdfDocumentSignature(sigData, sig, true, true);
    doc.AppendSignaturesFinal(true, new PdfDocumentSignature[]{ docsig });
}
```

Add a signature to a document

This C# sample signs a possibly certified document by adding a widget annotation and signing it.

```
public void AddAndSign(Stream stm, CmsInformation sigData, PdfBounds bounds)
{
    PdfDocumentSigner doc = new PdfDocumentSigner(stm, null);
    if (doc.Info.AllowedChanges !=
 DocumentMDPAllowedChanges.FillFormsSigningAndAnnotations)
        throw new Exception("No changes allowed.");
    SignatureWidgetAnnotation sig =
           new SignatureWidgetAnnotation(bounds, "NewSig", null, null);
    doc.PagesOfAnnotations[0].Add(sig);
```

```
    doc.Form.Fields.Add(sig);
    PdfDocumentSignature docsig = new PdfDocumentSignature(sigData, sig, true, true);
    doc.AppendSignaturesFinal(true, new PdfDocumentSignature[]{ docsig });
}
```

Linearized PDF

A Linearized PDF-file (also known as "Fast Web View") is a PDF-file with a specific structure and additional information that makes it possible to display the first page quickly, before the entire file is downloaded from the Web server. So, the total number of pages and size of the linearized PDF document should have little or no effect on the user-perceived performance of viewing any particular page.

In order to be linearized, a PDF file must meet the following criteria:

- The linearization parameter dictionary must be entirely contained within the first 1024 bytes of the PDF file. This limits the amount of data a conforming reader must read before deciding whether the file is linearized.
- A mismatch in the length of the file in the linearization dictionary and the actual file size indicates that the file is not linearized. Such files will be treated as an ordinary PDF. In this case the linearization information will be ignored. (If the mismatch resulted from appending an update, the linearization information may still be correct but requires validation).
- Linearized PDF files must have a specific structure according section F.3 of PDF 32000#1:2008 (found at http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf.)
- If hint tables are damaged or missed the PDF is not linearized.

DotImage allows create linearized PDF using these classes:

- `PdfDocument` - linearizes existing PDF files
- `PdfGeneratedDocument` - generates a new linearized PDF files and linearizes existing PDF files
- `PdfEncoder` - allows generate new linearized PDF files from images.

PdfDocument and PdfGeneratedDocument integraton

`PdfDocument` and `PdfGeneratedDocument` have an overload of `Save()` method with the `PdfSaveOption` parameter. Set the linearization flag in `PdfSaveOption` and include it in the `Save()` method to get a linearized PDF document.

The following example shows how to save linearized PDF using PdfDocument:

```
PdfDocument document = new PdfDocument("fileName.pdf");
            PdfSaveOptions options = new PdfSaveOptions {Linearized = true};
            document.Save("linearizedPdf.pdf", options);
This C# code linearizes PDF file using PdfGeneratedDocument
        using (var stm = File.Open("fileName.pdf", FileMode.Open, FileAccess.Read))
        {
            PdfGeneratedDocument document = new PdfGeneratedDocument(stm);
            PdfSaveOptions options = new PdfSaveOptions {Linearized = true};
            document.Save("linearizedPdf.pdf", options, null);
        }
```

PdfEncoder integration

To create linearized PDF with `PdfEncoder`, set the linearization flag in `PdfEncoder`. The following C# sample shows how to save linearized PDF using PdfEncoder:

```
FileSystemImageSource fs =
            new FileSystemImageSource(new[] {@"images.tif"}, true);
          PdfEncoder encoder = new PdfEncoder {Linearized = true};
          string outFile = "output.pdf";
          using (FileStream outfs =
              File.Open(outFile, FileMode.Create, FileAccess.ReadWrite))
          {
              encoder.Save(outfs, fs, null);
          }
```

PDF/A

PDF/A is a version of the Portable Document Format (PDF) designed to use in the archiving and long-term preservation of electronic documents. It restricts certain features as well as enforcing requirements to preserve the visual appearance of the document. All images must include color profiles to ensure proper color reproduction. All fonts must be embedded within generated PDF documents.

PDF/A in PdfDocument

For document-level manipulation of PDF documents Atalasoft DotImage provides the `PdfDocument` class, which can be used to add, move and remove pages, edit bookmarks and perform other operations on documents.

The `PdfDocument` class is designed to work with existent PDF documents and cannot be used to create PDF (or PDF/A) documents from the scratch. Nor can it fully work with page elements such as annotations, images and text.

The `PdfDocument` class cannot be used to convert PDF documents into PDF/A compliant documents because it order to do this it is necessary to access a page's elements such as images and text, which is not possible with PdfDocument.

However, the `PdfDocument` class can be used to work with existing PDF/A documents of the following versions:

- Pdf/A-1 (a, b).
- Pdf/A-2 (a, b, u).
- Pdf/A-3 (a, b, u) (if it does not contain a portfolio).

Changing part and conformance level of PDF/A is not supported.

A PDF/A document can be saved using the `PdfDocument` class, if:

- All source PDF documents conform to the PDF/A specification.
- All source PDF documents have color profiles with equal color spaces.

Saving a PDF/A document

If the source document is PDF/A compliant, the `PdfDocument` class saves PDF/A document without additional configuration.

```
var pdfDoc = new PdfDocument("inPdfA.pdf");
// ...
pdfDoc.Save("outPdfA.pdf");
```

If the source file (or one of source files) is not PDF/A compliant, the behavior of the `PdfDocument` class can be configured with `PdfASavingBehavior` property of the `PdfSavingOptions` class. The following options are available:

- `PreserveOriginalPdfType` - saves a PDF/A document or throws a PdfAException, if all source documents are PDF/A. Otherwise a regular PDF document will be saved. This value is used by default.
- `SavePdfA` - saves a PDF/A document, if all source documents are PDF/A and have color profiles with the same color spaces. Otherwise, it throws a PdfAException.
- `SavePdf` - saves regular PDF document without PDF/A metadata, regardless of the source document.

Saving behavior for single source document

PdfASavingBehavior	Single source document type	
	Regular PDF	PDF/A
PreserveOriginalPdfType	Regular PDF	PDF/A
SavePdfA	PdfAException	PDF/A
SavePdf	Regular PDF	Regular PDF

Example using PdfASavingBehavior:

```
var pdfDoc = new PdfDocument("inPdfA.pdf");
// ...
var options = new PdfSaveOptions
{
   PdfASavingBehavior = PdfASavingBehavior.SavePdf
};
pdfDoc.Save("regularPdf.pdf", options);
```

In order to save pages from several PDF/A documents into a single one, the following requirements are applied to the source documents:

- All source documents are PDF/A compliant.
- All source PDF documents have color profiles with equal color spaces.

Saving behavior for multiple source document

PdfASavingBehavior	Multiple source document type		
	Regular PDF	Mixed	PDF/A

PreserveOriginalPdfType	Regular PDF	Regular PDF	PDF/A or PdfAException
SavePdfA	PdfAException	PdfAException	PDF/A or PdfAException
SavePdf	Regular PDF	Regular PDF	Regular PDF

PDF/A or PdfAException means that result depends on versions of source PDF/A documents (see the PDF/A compatibility table) and main color profiles.

According to the specification, a single ICC color profile should be defined for a PDF/A document and placed in the `OutputIntents` PDF dictionary. Only color profiles with RGB or CMYK color spaces should be used.

`OutputIntents` provides the means for matching the color characteristics of a PDF document with those of a target output device or production environment in which the document will be printed.

Therefore, if one of the PDF documents contains a color profile with a different color space, the PDF/A document cannot be saved. In this case, the `PdfDocument` class will throw a PdfAException.

In order to find out whether a PDF/A document can be saved, the `IsPdfACompatible()` method of the `PdfDocument` class can be used. The `IsPdfACompatible()` method checks the main color profiles and the compatibility level (obtained from metadata) of all source documents.

The `IsPdfACompatible()` method does not check for compliance with the PDF/A standard.

```
var firstDoc = new PdfDocument("first.pdf");
var secondDoc = new PdfDocument("second.pdf");
firstDoc.Pages.AddRange(secondDoc.Pages);
var options = new PdfSaveOptions
{
   PdfASavingBehavior = firstDoc.IsPdfACompatible()
      ? PdfASavingBehavior.SavePdfA
      : PdfASavingBehavior.SavePdf
};

firstDoc.Save("output.pdf", options);
```

In case of saving PDF document, which contains pages from several documents with different part and conformance level of PDF/A standard, part and conformance level for the target document will be set according to the PDF/A compatibility table. The resulting conformance level is determined based on input documents levels that are read from metadata in the PDF/A compatibility table.

		Pdf/A-1		Pdf/A-2			Pdf/A-3		
		a	b	a	b	u	a	b	u
Pdf/A-1	a	Pdf/A-1a							
	b	Pdf/A-1b	Pdf/A-1b						
Pdf/A-2	a	exception	exception	Pdf/A-2a					
	b	exception	exception	Pdf/A-2b	Pdf/A-2b				
	u	exception	exception	Pdf/A-2b	Pdf/A-2b	Pdf/A-2u			
Pdf/A-3	a	exception	exception	exception	exception	exception	Pdf/A-3a		

	b	exception	exception	exception	exception	exception	Pdf/A-3b	Pdf/A-3b	
	u	exception	exception	exception	exception	exception	Pdf/A-3b	Pdf/A-3b	Pdf/A-3u

PDF/A data in PdfDocumentMetadata

Most PDF documents contain metadata in XML format that stores information about the time of creation and modification of the document, the author, etc.

According to the specification, a PDF/A document should contain XML-metadata with information about the part and compliance level of the PDF/A standard.

This data can be obtained from the `PdfAVersion` property of the `PdfDocumentMetadata` object:

```
public enum PdfAVersion
{
    PdfA1a,
    PdfA1b,
    PdfA2a,
    PdfA2b,
    PdfA2u,
    PdfA3a,
    PdfA3b,
    PdfA3u,
    NotPdfA
}
```

The PdfAVersion property value is set based on metadata retrieved from the PDF document. This metadata may be incorrect if PDF document itself does not conform to the PDF/A standard at all or to the specified version of the PDF/A standard. The `PdfDocument` class does not check for compliance with the PDF/A standard, therefore, the `PdfAVersion` property value may also be incorrect.

The `PdfAVersion` property can be obtained from:

- `PdfDocument.Metadata.PdfAVersion`
- `ExaminerResults.Metadata.PdfAVersion`
- `PdfGeneratedDocument.Metadata.PdfAVersion`
- `PdfDocumentMetadata.FromStream(…).PdfAVersion`

PDF/A in PdfGeneratedDocument

PdfGeneratedDocument can be saved as PDF/A-1b using PdfARenderer class.

PdfARenderer is inherited from PdfRenderer and placed in Atalasoft.dotImage.PdfDoc. It is responsible for creating PDF/A-1b files/streams from PdfGeneratedDocument objects.

The following code is a demonstration of PdfARenderer usage:

```
using (var source = File.OpenRead("doc.pdf"))
using (var genDoc = new PdfGeneratedDocument(source))
using (var cmykProf = new PdfIccColorSpaceResource(
       File.OpenRead("CMYK.icm"), true)) // see "Color Spaces"
using (var result = File.Create("result.pdf"))
{
```

```
        PdfARenderer renderer = new PdfARenderer(result)
        {
            // see "Color Spaces"
            CmykColorSpace = cmykProf,
            // see "Page Extraction"
            ImageExtractor = new AtalaImageExtractor(),
            // see "Annotations and Actions"
            IgnoreUnsupportedAnnotsAndActions = true,
            // see "Convert pages to images"
            ConvertIncompatiblePagesToImages = true
        };
        // see "Streamless Fonts"
        renderer.StreamlessFontFound += (o, arg) =>
                 arg.AlternativeFontPath = GetTTFont(arg.FontResource);

        renderer.Render(genDoc);
    }
```

For more information about properties and methods used in this sample, see subsequent chapters in this guide.

Convert pages to images

Not all components in a PDF document can be converted to the PDF/ A standard. To handle this case, PdfARenderer has a ConvertIncompatiblePagesToImages property. If this property is set to the "true" value and the page cannot be converted to PDF/A, the page is converted to the image instead of generating a PdfAException.

If the ConvertIncompatiblePagesToImages flag is set, AtalaImageExtractor should be provided to PdfARenderer and AtalaImageCompressor should be provided to compressors of PdfGeneratedDocument. Otherwise, the PdfARenderer class throws a PdfAException with the following message:

```
"To use ConvertIncompatiblePagesToImages, ImageExtractor is required."
```

AtalaImageExtractor is a class that can be set to ImageExtractor-property to extract a page image from a PDF document. This class located in Atalasoft.dotImage.PdfDoc.Bridge.dll and uses PdfDecoder from Atalasoft.dotImage.PdfReader.dll.

```
    using (var source = File.OpenRead("doc.pdf"))
    using (var genDoc = new PdfGeneratedDocument(source))
    using (var result = File.Create("result.pdf"))
    {
            genDoc.Resources.Images.Compressors.Add(new AtalaImageCompressor());
            PdfARenderer renderer = new PdfARenderer(result, false)
            {
                ConvertIncompatiblePagesToImages = true,
                ImageExtractor = new AtalaImageExtractor()
            };

            renderer.Render(genDoc);
    }
```

Color spaces

According to the PDF/A specification, each image and color space should use a specific color profile. PdfARenderer uses RgbColorSpace and CmykColorSpace to provide a specific color profile stream for each image. ICC and ICM files can be used as color profiles.

If RgbColorSpace is not specified, DefaultRgbColorSpace is used.

If a CMYK image is found and CmykColorSpace is not specified, the PdfAImageAndColorSpaceException is thrown with the following message:

```
"The PDF document contains an image with the DeviceCMYK color space. The PDF/
A specification requires the use of a specific color profile. Please, provide
the CmykColorSpace."
```

```
using (var source = File.OpenRead("doc.pdf"))
using (var genDoc = new PdfGeneratedDocument(source))
// Provide CMYK color profile
using (var cmykProf = new PdfIccColorSpaceResource
      (File.OpenRead("Microsoft Free CMYK Standard - RSWOP.ICM"), true))
// Provide RGB color profile
using (var rgbProf = new PdfIccColorSpaceResource(
        File.OpenRead("ISO22028-2_ROMM-RGB.icc"), true))
using (var result = File.Create("result.pdf"))
{
        PdfARenderer renderer = new PdfARenderer(result)
        {
           CmykColorSpace = cmykProf,
           RgbColorSpace = rgbProf
        };
        renderer.Render(genDoc);
}
```

Images

PdfARenderer does not support images with a mask. If a page contains an image with a mask, PdfARenderer throws an PdfAImageAndColorSpaceException with the following message:

```
"The PDF/A standard does not support images with a mask. Use
ConvertIncompatiblePagesToImages to convert the page to an image."
```

As mentioned in the error message, the user can use the ConvertIncompatiblePagesToImages flag to convert the entire page into the image.

The PDF/A-1 standard prohibits images with more than 8 bits per color component. Such images will be extracted and converted using an ImageExtractor.

In addition, the PDF/A-1 standard prohibits images with JPEG2000 compression. Such images will be extracted using an ImageExtractor and then recompressed.

Fonts

A PDF document can use streamless fonts, which are based on metrics and descriptions, but do not contain a font stream (FontFile, FontFile2 and FontFile3 in terms of the PDF specification). These fonts should be well-known to PDF readers.

According to the PDF/A specification, all fonts must be embedded to the target file; in addition to the metrics and descriptions of fonts, the document should contain a font stream.

Standard fonts

The PDF standard contains 14 Standard Type 1 Fonts (standard fonts) that are well-known to PDF readers and specified only by font name in PDF documents.

According to the PDF/A specification, standard fonts used in a document must be embedded in the target file. Therefore, a standard font used in the document is replaced by the system font, which is loaded from the system font directory according to the following table.

Original Standard Font	**System Font**
Times-Roman	Times New Roman
Times-Bold	Times New Roman Bold
Times-Italic	Times New Roman Italic
Times-BoldItalic	Times New Roman Bold Italic
Helvetica	Arial
Helvetica-Bold	Arial Bold
Helvetica-Oblique	Arial Italic
Helvetica-BoldOblique	Arial Bold Italic
Courier	Courier New
Courier-Bold	Courier New Bold
Courier-Oblique	Courier New Italic
Courier-BoldOblique	Courier New Bold Italic
Symbol	*Not supported
ZapfDingbats	*Not supported

* No similar system fonts exist for the Symbol and ZapfDingbats standard fonts. PdfARenderer can't save PDF/A documents with these fonts without generating a PdfAException at runtime with the following message:

```
"Could not find alternate font file for font with name [Symbol/ZapfDingbats].
You can handle the StreamlessFontFound event by providing your own font."
```

Or, if ConvertIncompatiblePagesToImages property is set, the pages that use these fonts are converted to images.

To replace a standard font with another one, the event StreamlessFontFound is used. See the next section.

Streamless fonts

If no stream exists in the font, PdfGeneratedDocument tries to find a similar system font by the name specified in /BaseFont.

Then the StreamlessFontFound event, placed in the PdfARenderer class, is called. The arguments of the StreamlessFontFound event contain:

- FontResource: PDF font resource that contains information about the font.
- GlobalFontName: Global name of PDF font resource that does not have a stream. If for some reason the font is not imported to global resources, this value is null.
- AlternativeFontPath: Path to a file with a similar system font. If a similar font is not found, the value is null.

The user can replace AlternativeFontPath with the path to a specific file with a TrueType font on handling this event.

```
        using (var source = File.OpenRead("doc.pdf"))
        using (var genDoc = new PdfGeneratedDocument(source))
        using (var outStm = File.Create("mergedPdfA.pdf"))
         {
            var renderer = new PdfARenderer(outStm);
            renderer.StreamlessFontFound += (o, arg) =>
                 arg.AlternativeFontPath = GetTTFont(arg.FontResource);
            renderer.Render(genDoc);
        }
```

If a similar font is not found or is not provided by the user, PdfAFontException is thrown with the following message:

```
"Could not find alternate font file for font with name [base font name]. You
can handle the StreamlessFontFound event by providing your own font"
```

Or, if ConvertIncompatiblePagesToImages property is set, the pages that use this font are converted to images.

If a font file is found in system fonts or a user file is provided, the PDF file font is converted to TrueType and the preceding file is used as the stream.

Encoding

According to PDF specifications, fonts may include encoding to support the use of ANSI character codes that represent Unicode symbols, without using Unicode strings and CID fonts.

If a file contains a Type 1 streamless font, it is converted to the TrueType font by PdfARenderer. However, this capability is limited, because nonsymbolic TrueType fonts can have only WinAnsiEncoding or MacRomanEncoding. PdfARenderer does not support the Type 1 fonts with custom encoding. Therefore, if any input document contains such fonts, the PdfAFontException is thrown. To avoid the exception, be sure to edit your input document to remove Type 1 fonts that include custom encoding.

Transparency

A PDF document can contain objects or groups of objects with full or partial transparency (see "11 Transparency" in PDF 32000-1:2008). Text, images, annotations and other objects can be transparent.

According to the PDF/A-1b specification, transparency is not supported. So, all transparent objects become opaque.

Annotations and Actions

The following action types are **not permitted** in PDF/A.

- ImportData
- JavaScript
- Launch
- Movie
- ResetForm
- SetState
- Sound

If PdfGeneratedDocument contains one of these actions, PdfARenderer throws a PdfAActionException with the following message:

```
"PDF/A does not support [action type] actions."
```

The following annotation types are **permitted** in PDF/A:

- Text
- Link
- FreeText
- Line
- Square
- Circle
- Highlight
- Underline
- Squiggly
- StrikeOut
- Stamp
- Popup
- Widget
- PrinterMark
- TrapNet

If PdfGeneratedDocument contains other annotations, PdfARenderer throws a PdfAAnnotationException with the following message:

```
"PDF/A does not support [annotation type] annotations."
```

The IgnoreUnsupportedAnnotsAndActions-property of PdfARenderer class allows the ability to ignore annotations and actions that not supported by the PDF/A-1b standard, instead of generating an exception.

All partially transparent annotations become opaque, because transparency is not supported by the PDF/A-1b standard.

Merge PDF/A documents

To merge PDF documents, create a PdfGeneratedDocument object based on one of the documents, call the ImportPages() method with the path to other documents. For more information, see Merge PDF Forms.

After that, use PdfARenderer to save PDF/A document:

```
    using (var stream = File.OpenRead(@"first.pdf"))
    using (var genDoc = new PdfGeneratedDocument(stream))
    using (var streamForImport = File.OpenRead(@"second.pdf"))
    {
        genDoc.ImportPages(streamForImport);
        using (var outStm = File.Create("mergedPdfA.pdf"))
        {
            var renderer = new PdfARenderer(outStm)
            {
                 ImageExtractor = new AtalaImageExtractor()
            };

            renderer.Render(genDoc);
        }
    }
```

Error handling

While saving the PDF/A document using the PdfARenderer class, the following types of PdfAExceptions can be thrown.

Error type	Message	Solution
PdfAException	To use ConvertIncompatiblePagesToImages, ImageExtractor is required.	Set AtalaImageExtractor to the PdfARenderer.ImageExtractor-property.
PdfAException	PDF/A standard does not support transparency.	Set ConvertIncompatiblePagesToImages = true.
PdfAException	PDF/A standard does not support transfer functions.	Set ConvertIncompatiblePagesToImages = true.
PdfAException	Form or form elements do not comply with the PDF/A standard. See the inner exception for details.	Solve the inner exception or remove Form from the document. This issue and inner exception cannot be solved with the ConvertIncompatiblePagesToImages or IgnoreUnsupportedAnnotsAndActions flags.
PdfAActionException	PDF/A does not support [action type] actions.	Set IgnoreUnsupportedAnnotsAndActions = true.

PdfAActionException	PDF/A does not support AdditionalActions in widget annotations.	Clear AdditionalActions property in widget annotations or set IgnoreUnsupportedAnnotsAndActions = true.
PdfAActionException	PDF/A does not support ClickActions in widget annotations.	Clear ClickActions-property in widget annotations or set IgnoreUnsupportedAnnotsAndActions = true.
PdfAActionException	PDF/A does not support AdditionalActions in form fields.	Clear AdditionalActions-property in form fields or set IgnoreUnsupportedAnnotsAndActions = true.
PdfAAnnotationException	PDF/A does not support [annotation type] annotations.	Set IgnoreUnsupportedAnnotsAndActions = true.
PdfAAnnotationException	Annotation appearance does not comply with the PDF/A standard. See the inner exception for details.	Solve inner exception, replace annotation appearance or set IgnoreUnsupportedAnnotsAndActions = true. This and inner exception cannot be solved with ConvertIncompatiblePagesToImages flag.
PdfAFontException	Could not find alternate font file for font with name [BaseFont]. You can handle the StreamlessFontFound event by providing your own font.	Handle StreamlessFontFound event and provide font to AlternativeFontPath argument. Or set ConvertIncompatiblePagesToImages = true.
PdfAFontException	Type1 streamless fonts with custom encoding are not supported.	Use external tool to modify input document to remove any Type 1 fonts with custom encoding. Or set ConvertIncompatiblePagesToImages = true.
PdfAImageAndColorSpaceException	PDF/A standard does not support images with a mask. Use ConvertIncompatiblePagesToImages to convert the page to an image.	Set ConvertIncompatiblePagesToImages = true.
PdfAImageAndColorSpaceException	The PDF document contains an image with the DeviceCMYK color space. The PDF/ A specification requires the use of a specific color profile. Please provide the CmykColorSpace.	Provide CMYK color profile stream to CmykColorSpace-property. Or set ConvertIncompatiblePagesToImages = true.
PdfAImageAndColorSpaceException	The PDF document contains content with the DeviceCMYK color space. The PDF/ A specification requires the use of a specific color profile. Please provide the CmykColorSpace.	Provide CMYK color profile stream to CmykColorSpace-property. Or set ConvertIncompatiblePagesToImages = true.

PdfAImageAndColorSpaceException	Page contains unimported images that may contain prohibited parameters for PDF/A-1b.	Set ConvertIncompatiblePagesToImages = true.
PdfAImageAndColorSpaceException	Image [global name] has [value] bits per color component. PDF/A-1 prohibits the use of images that exceed 8 bits. Provide ImageExtractor to extract and convert the image.	Set AtalaImageExtractor to the PdfARenderer.ImageExtractor property.
PdfAImageAndColorSpaceException	Image [global name] used JPEG2000 compression. PDF/A-1 prohibits the use of JPEG2000 compression. Provide ImageExtractor to recompress the image.	Set AtalaImageExtractor to the PdfARenderer.ImageExtractor property.
PdfAImageAndColorSpaceException	Image [global name] has an unknown color space that can be prohibited for PDF/A-1. Provide ImageExtractor to extract and convert the image.	Set AtalaImageExtractor to the PdfARenderer.ImageExtractor property.

PDF 2.0

PDF 2.0 is an ISO-standardized second version of the PDF, that extend basic standard with new features.

PdfGeneratedDocument supports creation, opening, saving, editing of PDF 2.0 documents.

Document update to version 2.0 is also supported if all annotations contain an Appearance. Otherwise PdfException is thrown.

```
using (var doc = new PdfGeneratedDocument(stm))
{
     doc.PdfVersion = 2.0;
     doc.Save("result.pdf");
}
```

Sound annotations and actions are deprecated in PDF 2.0. SoundAnnotation and PdfSoundAction classes are preserved for PDF 1.7 and earlier, but they should not be used in PDF 2.0 documents. Otherwise, PdfGeneratedDocument will skip them or throw PdfException ("Object cannot be saved. Object type: {objectType}. Reason: object deprecated in PDF 2.0") on the document save attempt. For more information, see Document upgrade to PDF 2.0.

Creation of new digital signatures, 3D-annotations and other objects, specified in PDF 2.0, are not supported. But these objects will be preserved "As Is" if they already exist in the original document.

Document upgrade to PDF 2.0

Several PDF objects are deprecated in PDF 2.0. Some of the deprecated objects can be safely skipped because they contain redundant data. Skipping other deprecated objects may cause data loss. Here the list of such objects:

- XFA
- Sound annotation
- Movie annotation
- Sound action
- Movie action

During the upgrade document with deprecated objects, that cannot be safely skipped, PdfDocument and PdfGeneratedDocument can skip these objects or throw PdfException with object type and reason.

For this purpose, ObjectCannotBeSaved-event is added to PdfSaveOptions. This event occurs on saving PDF 2.0 document, before writing deprecated objects.

```
 using (var stm = File.OpenRead(@"pdf1_5.pdf"))
{
    var doc = new PdfGeneratedDocument(stm);
    doc.PdfVersion = 2.0;
    var options = new PdfSaveOptions();
    options.ObjectCannotBeSaved += (arg, obj) =>
    {
       switch (obj.ObjectType)
       {
            case "XFA":
                 obj.Action = ObjectCannotBeSavedEventArg.SaveObjectAction.Skip;
                 break;
            case "SoundAnnot":
                 obj.Action =
 ObjectCannotBeSavedEventArg.SaveObjectAction.ThrowException;
                 break;
        }
     };
     doc.Save("pdf2_0.pdf", options, null);
}
```

Chapter 6

DotTwain

DotTwain is a 100% Managed .NET Windows Form component with fast capture of digital images from scanners, cameras, and other devices supporting TWAIN.

- Utilizes the improvements of the TWAIN 2.4 specifications for ultra fast scanning.
- Works with a .NET Bitmap object and has no dependencies other than the .NET Framework.

Supported features

- Full context-sensitive help and documentation.
- Object-oriented design; a TWAIN extension to the .NET Framework.
- Create a custom interface to scan images, or use the default TWAIN interface.
- Automatic Document Feeder support, with full control of the feeder operations.
- Support for duplex scanning.
- File system support for navigating the internal device memory.
- Acquire images directly to file utilizing compression algorithms supported by the device.
- Select a region of the scanning bed to be acquired with frame support.
- Upload a scanned image to a server with the built in HTTP Post class.
- Simplified automatic capture allowing the device to quickly capture multiple images for later acquisition.
- Imprinter/Endorser methods allowing a text string to be stamped onto the acquired image during the acquisition.
- Barcode/Patchcode classes to access barcode and patchcode recognition abilities of the device.
- A low-level class allowing even more control over the scanning session for advanced users.
- Extensible to allow custom driver capabilities access.
- Image dataset support enables acquisition of specific images from a camera in a single operation.
- Quickly find out what capabilities, compression modes, frame sizes, resolutions and more, are supported by a device.
- Over 80 properties and 50 methods to give total control over the image acquisition.

DotTwain has been tested on many industry leading document scanners including those from Kodak, Fujitsu, Panasonic, Canon, Visioneer, Xerox, and Bell & Howell.

About DotTwain

This introduces you to the basic classes you need to know about to gain a general understanding of DotTwain and to begin using it.

Acquire images

Acquisition

The Acquisition object is the primary class in DotTwain. You can add this component to the toolbox and then drop it onto a form. You can also instantiate the component directly. This is the only class you need to add standard image acquisition capabilities to an application.

For greater control over the acquire process, this class contains a collection of Device objects that controls numerous properties used for the image acquisition.

TwainController

The TwainController object is a low-level TWAIN class used by the Device and Acquisition objects. By using this class you gain more direct access to TWAIN and you can use custom driver capabilities.

This class can be created as a standalone object or can be accessed through the Device.Controller property.

ⓘ Only advanced users with knowledge of the TWAIN specification should use this class.

Device

The Device object provides full access to a TWAIN compatible source on the system. Use it to open a connection to the device, to get and set properties, and then to acquire one or more images. Because this class represents a system device resource, you cannot create an instance of it. You can obtain an instance to a Device object by calling ShowSelectSource, or from the Devices collection in the Acquisition object.

DeviceCollection

The DeviceCollection holds a read-only collection of Device objects which represent all of the TWAIN compatible system resources. You can obtain the system default device from this collection. A suitable device also can be found by enumerating through the collection. Should system conditions change, such as a device being unplugged, the ScanForChanges () method can be used to recreate this collection.

Document feeder control

Many scanners have an automatic document feeder (ADF) that allow scanning multiple images in a single process. The DocumentFeeder class gives full control over the feeder. You can also use this control to enable or disable the class.

Navigating files in a camera

FileSystem

Use the FileSystem object to walk through the file system structure in a TWAIN compatible camera's internal storage. You can also use the FileSystem to create, delete, copy and rename files and directories directly in the camera.

Use the SetImageDataset () method in the Device object to acquire images from the camera, then call the Acquire () method of the same Device object.

ActiveX control API reference

AcquisitionControl provides the central interface for DotTwain ActiveX acquisition and the transfer of images.

Class API

void **Initialize()**

Initializes the control and begins the licensing process. This must be called before any other method. In most cases, the ideal place to call this from is your page's <body onload=""> event.

bool **IsTwainAvailable()**

true: if Twain is installed on the client system and at least one Twain device is present.

false: if Twain is not installed on the system or no Twain devices are found.

String[] **GetAvailableDevices()**

Returns an array of strings, each representing an installed Twain device.

void **Scan(** String deviceName)

Begins the scanning process with the specified Twain device. The passed deviceName must match a string returned from GetAvailableDevices exactly.

IJSDevice **GetDevice**(String deviceName)

Used for batch scanning. The passed deviceName must match a string returned from GetAvailableDevices exactly. Returns a queryable representation of a Twain device.

IJSBatch **CreateNewBatch**(IJSDevice device)

Used for batch scanning. Given an IJSDevice from the GetDevice method, creates a new IJSBatch which is a parameterized representation of the batch acquisition process.

void **BatchScan**(IJSBatch batch)

Used for batch scanning. Given an IJSBatch from the CreateNewBatch method, starts the Twain acquisition process.

Class events

ControlInitializing(JSControlInitializationStartingEventArgs args)

Occurs immediately after the Initialize() method is called.

ControlInitializationComplete(JSControlInitializationCompleteEventArgs args)

Occurs after the control has been successfully initialized and licensed.

BatchStarting(JSBatchStartingEventArgs args)

Occurs immediately after the Scan() or BatchScan() methods are called.

ImageAcquired(JSImageAcquiredEventArgs args)

Occurs after each Twain image acquisition.

TransferStarting(JSTransferStartingEventArgs args)

Occurs before the transfer of each image.

TransferComplete(JSTransferCompleteEventArgs args)

Occurs when the transfer of each image has completed.

BatchComplete(JSBatchCompleteEventArgs args)

Occurs when all Twain images have been acquired and all image transfers have completed or otherwise stopped.

ImageAcquisitionCancelled(JSImageAcquisitionCanceledEventArgs args)

Occurs when the user clicks the cancel button in the Twain driver UI.

Error(JSErrorEventArgs args)

Occurs when an internal exception has occurred in the AcquisitionControl.

Getting started

Add DotTwain to the toolbox

Add the Acquisition component to the toolbox.

1. On the Visual Studio .NET menu, click **Tools** > **Customize Toolbox.**
2. Make sure the .NET Framework Components tab is selected, and find the Atalasoft.Twain namespace.

3. Select the checkbox that corresponds to the Acquisition component.
4. Click OK.

Set application information

After you add the Acquisition control to your form, you should set the ApplicationIdentity properties which will be used by the TWAIN driver.

If you do not set the ApplicationIdentity properties, default values are used.

Setting the following properties is optional but recommended:

- Country
- Info
- Language
- Manufacturer
- ProductFamily
- ProductName
- VersionMajor
- VersionMinor

A Parent property is used by TWAIN when displaying dialogs and acquiring images. You must set the Parent property to the parent form or control in order to acquire images.

Model acquisition

By default, the Acquire() method is asynchronous and returns before scanning is complete. This method can be made to work in a synchronous manner by using the Device object and setting its ModalAcquire property to true. The example below illustrates this technique.

C#

```
this.device = this.acquisition.Devices.Default;
this.device.ModalAcquire = true;
this.device.Acquire();
```

Set up events

You need to use events when acquiring images. When an image is acquired, the ImageAcquired event fires providing an AcquireEventArgs object containing the image. At the very least, the ImageAcquired event must be handled, but it is recommended that the AcquireCanceled and AcquireFinished events also be handled. The following code shows how the image is handled.

C#

```
this.acquisition.ImageAcquired += new ImageAcquiredEventHandler(OnImageAcquired);
private void OnImageAcquired(object sender, AcquireEventArgs e)
{
    // If the image exists, load it into a WorkspaceViewer control.
 if (e.Image != null)
    {
```

```
        this.viewer.Images.Add(AtalaImage.FromBitmap(e.Image));
        e.Image.Dispose();
    }
}
```

Show the Select Source dialog

Your application should allow users to select which TWAIN device they want to use. This is accomplished by displaying the "Select Source" dialog using the ShowSelectSource() method. The code below assumes the Acquisition component is named acquisition.

Properties

To get or set a device property, you must open a connection to the device using the Open() method. Whenever the Open() method is invoked, the Close() method must be invoked to close the connection. Closing a connection resets all of the device properties to their default values. Because of this, it is best to close a device after the image, or all desired properties, are acquired.

Open() and Close() only need to be used when getting or setting properties on the device.

The code below opens a connection to the device in order to retrieve the default Resolution and BitDepth values of the device, then closes the connection. This technique can be useful if you are looking for a device in the DeviceCollection with specific default properties or capabilities. See QueryCapability for more information.

C#

```
[C#]
device.Open();
ResolutionData res = device.Resolution;
int bitDepth = device.BitDepth;
device.Close();
```

Acquire an image

You can acquire an image through the Acquisition object or from a Device object. If you do not need to get or set any properties, the easiest to use the Acquire() method from the Acquisition object. This method uses the system default device, as shown here.

C#

```
this.acquisition.Acquire();
```

You can choose to hide the device interface and/or ask that the device save the acquired image directly to file.

Once the image has been acquired, the ImageAcquired event fires and provides an AcquireEventArgs object containing the image. If you invoked the Open() method in order to set properties before the acquire, invoke the Close() method in the AcquireFinished event to close the device.

Acquire images with TWAIN

Unless your documents are already stored digitally, the first step in any document imaging application is to acquire the images via a scanner. Atalasoft DotImage has a Microsoft .NET component called DotTwain, included in Atalasoft DotImage Document Imaging, that has advanced TWAIN acquisition features. Once the document is in digital form, the image can be cleaned-up, displayed, compressed, archived, and recognized using other third party systems.

DotTwain returns System.Drawing.Bitmap images that can easily be converted to Atalasoft DotImage AtalaImage objects.

Acquire a select region of the device

Some scanners allow you to select the rectangular region of the scanning bed you want to acquire. If you know that you only need a specific area or page size, setting a region can increase your scanning speed.

This region is represented by the Device object, Frame property. Alternatively you can use the ImageLayout property. In some cases a driver only supports one of these properties.

As Frame values are in Units, you need to know the value of the Units property before setting the Frame size.

The code example below assumes you have already opened a connection to the device by calling the Open() method. This example shows how to acquire a specific size while the second shows how to acquire any size and position.

Example: Acquire a specific region size

If you only need to specify a standard region size, you can use the FrameSize property as shown below.

C#

```
// Make sure the FrameSize property is supported by the device.
if (this.device.QueryCapability(DeviceCapability.FrameSize, true)
{
    // Get a list of supported frames and choose the one that fits your needs.
    StaticFrameType[] frames = this.device.GetSupportedFrameSizes();
    foreach (StaticFrameType frame in frames) {
        if (frame == StaticFrameType.LetterUS)
        {
            this.device.FrameSize = frame;
            break;
        }
    }
}
```

Example: Acquire any size and position

If you need a more control over the size and position, or if the device does not support the FrameSize property, you can attempt to set the acquisition area using the Frame property.

C#

```
// Try to use Inches.
```

```
this.device.Units = UnitType.Inches;

if (this.device.Units != UnitType.Inches) return;
this.device.Frame = new System.Drawing.RectangleF(0, 0, 8.5, 11);
```

Acquire and save images directly to a file

Some devices allow to you acquire an image and save it directly to a file, instead of returning the image data. While the overall process is simple, there are some steps you must take:

- Check For Device Capabilities
- Set_Filenames
- Close_the_Connection

These are illustrated by the following example.

Example: Checking for device capabilities

Begin by testing the device to make sure it can save a file. If so, you then need to negotiate the type of file to save.

C#

```
// Open a connection to the device.
this.device.Open();
this.device.TransferMethod = TwainTransferMethod.TWSX_NATIVE;

// See if the device supports file transfer.
TwainTransferMethod[] methods = this.device.GetSupportedTransferMethods();
foreach (TwainTransferMethod method in methods)
{
if (method == TwainTransferMethod.TWSX_FILE2)
{
// Use TWSX_FILE2 when possible.
this.device.TransferMethod = method;
break;
}

if (method == TwainTransferMethod.TWSX_FILE)
this.device.TransferMethod = method;
}

// If it's not supported tell stop.
if (this.device.TransferMethod == TwainTransferMethod.TWSX_NATIVE)
{
// Close the connection.
this.device.Close();
MessageBox.Show("The current device does not support saving directly to a file.");
return;
}

// Find out which file types the device can save to.
SourceImageFormat[] formats = this.device.GetSupportedImageFormats();

// We want to save the image as a TIFF.
foreach (SourceImageFormat format in formats)
{
if (format == SourceImageFormat.Tiff)
{
// TIFF is supported, so set the FileFormat.
this.device.FileFormat = format;
```

```
// Now lets try to use Group4 or Group3 compression.
// We could use GetSupportedCompressionModes, but we
// will simply try setting the Compression property instead.
this.device.Compression = CompressionMode.Group4;
if (this.device.Compression != CompressionMode.Group4)
this.device.Compression = CompressionMode.Group3;

break;
}
}

// Start the acquire process, using the device's interface.
this.device.Acquire();
```

Example: Set file names

During the acquire process, the FileTransfer event is raised just before each file is acquired. You need to set the FileName property of the FileTransferEventArgs object passed into the event. This tells the device where the file should be saved.

C#

```
private void OnFileTransfer(object sender, FileTransferEventArgs e)
{
    e.FileName = @"C:\TwainImages\whatever.tif";
}
```

Example: Close the connection

When all of the images have been acquired, the AcquireFinished event is raised. Close the connection here.

C#

```
private void OnAcquireFinished(object sender, System.EventArgs e)
{
    this.device.Close();
}
```

Detect a camera device

There are times when you only want to use a camera device. Unfortunately, TWAIN does not provide a direct way of knowing what type of device is being used. You can, however, do a little investigating to pick out a camera from a scanner.

Use the QueryCapability() method to find out if a device supports certain features that are normally only supported by camera devices. You can check the following capabilities:

- Flash
- Flash2
- ExposureTime
- BatteryMinutes
- BatteryPercentage
- CameraPreviewInterface
- PowerSupply

- ZoomFactor

Upload an image to a server

There may be times when you want to send an image to a server. The HttpPost class was written for this specific purpose.

Upload the image

The following example sends a single image, along with a user name and password to identify the sender, to a server.

Example

C#

```
private void UploadImage(Atalasoft.Imaging.AtalaImage image)
{
// Create an instance of HttpPost and use the default image encoder.
Atalasoft.Imaging.HttpPost post = new Atalasoft.Imaging.HttpPost();

// Add the image to the form data collection.
post.FormData.Add("image1", image, "filename.png");

// Add data to identify the user posting the image.
post.FormData.Add("username", user);
post.FormData.Add("password", password);

// Post the form data to the server and retrieve a return value.
string ret = post.PostData("http://www.website.com/postImage.aspx");
}
```

The first part of the code creates an instance of HttpPost and uses the default image format, which is PNG. The image format determines the file type of the image when saved on the server.

Then the image is added to FormDataCollection, which includes the field name, image and filename for the image. It is best to think of HttpPost as an HTML FORM object. The FormData represents the FORM elements; in this case a FILE input box. You can update multiple images by adding them to the collection; just be sure each has a unique field name.

Next, add the username and password of the person sending this image. In many cases, additional information needs to be sent with the image in order to perform a specific action on the server. In terms of an HTML FORM, this would be a TEXTBOX field.

Finally, the FormData is sent to the server using the PostData() method. The return value of PostData is a string that is sent back from the server. Normally this is used to confirm the success of the call.

Saving the image

Once the image has been sent, save it to the server using ASP.NET. If necessary, you can modify this code to store the image in a database instead.

Example

C#

```
private void Page_Load(object sender, System.EventArgs e)
{
if (Request.Files.Count == 0) return;

// Save the file to the server.
string fileName = System.IO.Path.GetFileName(Request.Files[0].FileName);
Request.Files[0].SaveAs(GetNewFileName(fileName));

// Return the path to this file.
Response.Clear();
Response.Write("success");
Response.End();
}

private string GetNewFileName(string fileName)
{
// Create a unique filename.
string path = Server.MapPath("./images") + "\\" + Session.SessionID + fileName;
return path;
}
```

In the code above, the SaveAs() method saves the image to the server. Notice that the FileName property gets the name of the posted file. This is the same value that was passed to the FormData.Add() method when you added the image.

The data written to the Response object is returned by the PostData() method. It is a good idea to use the Response.Clear() method before adding your return value. This example simply returns success

Deploy DotTwain

To distribute DotTwain along with your .NET application, you need to include `Atalasoft.DotTwain.dll` and `Atalasoft.Shared.dll` in the same folder as the assembly that references it.

Be sure that the `.dll` versions you provide match that used to compile the assembly.

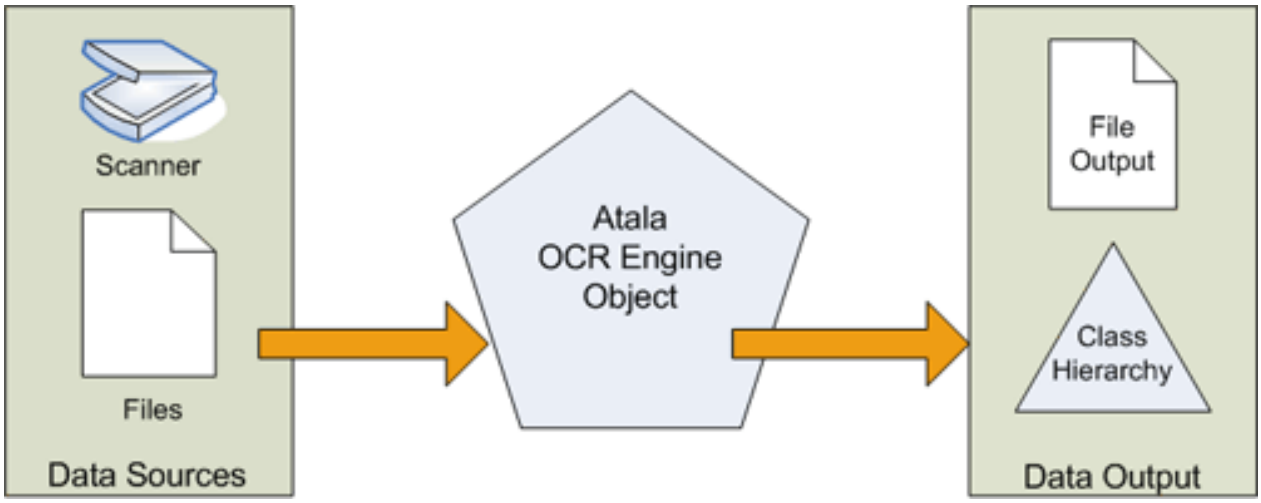
Web-based deployment is not available.

Chapter 7

OCR engine

Atalasoft DotImage OCR is designed to easily interface with other aspects of your application. It is extensible with an event driven object-oriented object model. In just a few lines of code, a developer can recognize an image and output that image to a file, or enumerate its lines, words, and characters with confidence.

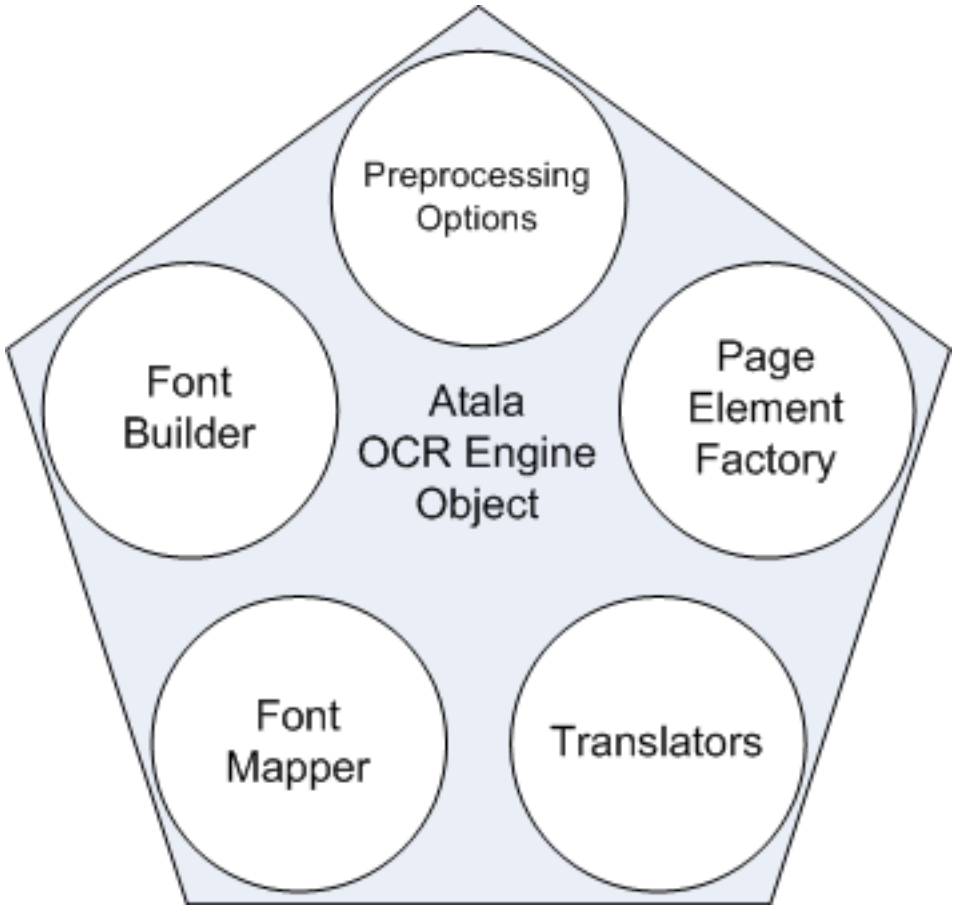
Data sources for the engine can be scanned images or files. The engine output consists of either a file or a class hierarchy. This model is illustrated below.



As OcrEngine object is abstract, you cannot create an instance of this object. Nevertheless, the object definition contains most of the necessary functionality needed for a concrete subclass to function with a minimum of extra code.

The OcrEngine object has five primary components as illustrated below:

- Preprocessing options
- Document translators
- Page element factory
- Font mapping
- Font building



OCR document design considerations

Since some 1-D barcodes are limited in what they can represent, it is useful to be able to ask the BarcodeWriter if it can correctly render a given string. This is done by calling the BarcodeWriter's Validate method. Validate takes a string and a boolean and will return true if the string can be rendered in the BarcodeWriter's current BarcodeStyle. If the boolean passed in is true, the BarcodeWriter will throw an ArgumentException if given an invalid string. If the boolean passed in is false, the Validate will return false if given an invalid string.

The design of the OcrDocument hierarchy reflects the conflicting needs of OCR engines and OCR clients.

It is highly likely that a client of an OcrEngine will modify the OcrDocument and OcrPage classes. For example, a client may add keywords to the OcrDocument object or thumbnail images to the pages. Therefore, concrete engine implementations should never construct an OcrDocument or OcrPage class directly. It is the responsibility of the client to supply code to construct the objects within the hierarchy.

On the other hand, the engine is likely to supply implementations of OcrLine, OcrWord, and OcrGlyph that can be tightly coupled to data supplied by the engine. This means the engine needs to be able to make very specific versions of these classes.

To manage these conflicting goals, the Ocr namespace uses two patterns. The first is a factory used to construct document elements. Never access constructors for document elements directly. Use the engine's Factory property instead as shown in the example below.

Use the factory property to construct a document

```
OcrPage page;

//Do not do this!
page = new OcrPage(width, height, resolution);

//This is much better
page = engine.Factory.OcrPage(width, height, resolution);
```

The page element object represents an extensible interface to the element without dictating the implementation. The data accessor defines how that data is stored and retrieved.

For example, the implementation of the Baseline property in OcrLine does not use a member variable to store the baseline. Instead, it uses an object called an OcrLineAccessor to get the baseline. In this way, a client can supply definitions for OcrLine, OcrWord, and OcrGlyph.The engine supplies the accessors that define how the object's information is retrieved. A client could then override OcrLine to include extra information about a line of text without affecting how any given engine is required to provide information that meets the core definition of an OcrLine.

Furthermore, once an OcrDocument has been constructed, its elements can be edited, augmented, deleted, or merged without affecting any of the existing components. For example, OcrPage objects taken from an OcrDocument recognized by one engine can be inserted into an OcrDocument that was produced by another engine, or words can be changed (spell corrected, moved, replaced) without affecting any of the other elements, *even if they are still tied to a specific engine.*

To get an OcrDocument or an OcrPage from an image, use the OcrEngine's Recognize() method. There are versions that operate on single images and ImageSource objects.

Tesseract engine

The Tesseract Engine, class name TesseractEngine, is an open source engine that we provide without charge to those who purchase the OCR Package. It is a commercial quality OCR engine originally developed at HP between 1985 and 1995. HP and UNLV open-sourced this engine in 2005.

Features

The Tesseract engine is fast and runtime royalty free although it is not quite as powerful as the other engines supported by Atalasoft DotImage. In particular, it lacks segmentation and it is not very good at recognizing low quality documents.

Supported languages

The TesseractEngine supports the following languages:

- Dutch
- English
- French
- German
- Italian
- Portuguese
- Spanish

Supported output formatters

The TesseractEngine supports the following output formatters and provides a structure that allows you to build your own.

- Text
- PDF

Deployment

- The assemblies listed below are required for deployment.
  - Atalasoft.dotImate.Ocr.Tesseract3
  - Atalasoft.dotImage
  - Atalasoft.dotImage.Ocr
  - Atalasoft.dotImage.Lib
  - System
  - System.Data
  - System.Drawing

Additionally, the Tesseract language files must be accessible. These are automatically placed in the Atalasoft DotImage directory during toolkit installation. When deploying, you must either copy the OcrResources to your application directory or tell the engine their location explicitly by passing it into the TesseractEngine constructor.

Please see the Tesseract3Engine class documentation for additional information.

The Tesseract Engine is used in exactly the same way as the other OCR engines, all of which inherit from the same base class, Atalasoft.dotImage.OCR.

Special considerations

Once the Tesseract Engine is used and recognize is called with a language, you cannot change to an alternate language. The initialization happens the first time a document is recognized. Attempting to change the language at any time beyond that point results in an exception being thrown.