

Kofax Web Capture Developer's Guide

Version: 11.5.0

Date: 2024-05-24

KOFAX

© 2024 Tungsten Automation. All rights reserved.

Tungsten and Tungsten Automation are trademarks of Tungsten Automation Corporation, 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 Tungsten Automation.

Table of Contents

Preface	8
Related documentation.....	8
Training.....	8
Getting help with Kofax products.....	8
Chapter 1: Deploy Kofax Web Capture	10
Visual C++ Runtime dependencies.....	10
Deploy Kofax Web Capture in ASP.NET.....	10
Dependencies using Kofax Web Capture class library.....	10
Dependencies using Kofax Web Capture with WebControls.....	10
Generating licenses.....	11
Chapter 2: Web scanning	13
Getting Started with Web Capture.....	13
Kofax Web Capture demos.....	13
Set up a new project.....	14
Add the Web Document Viewer handler.....	15
Add the Web Capture handler.....	15
Set up the scanning controls and viewer.....	18
Wrap-up.....	20
Deploy on multiuser environment.....	20
Chapter 3: Web scanning server	22
Troubleshoot Web Capture Handlers.....	22
Extend the KicHandler.....	23
Connect to Kofax Import Connector services.....	23
Modify <code>web.config</code>	23
Specify the Kofax Import Connector endpoint.....	24
Configure Kofax Import Connector.....	25
Required license.....	25
Configure the service.....	25
Configure the Electronic Documents plugin.....	26
Test the configuration.....	26
Chapter 4: Web scanning client	27
Initialize the control on the client.....	27
Include WebCapture Javascript.....	27
Initialize.....	27

Connect to UI controls.....	29
Examples of UI controls.....	30
Filter selection lists.....	31
Connect controls with no UI.....	32
Import loose pages.....	32
Batch fields.....	33
Display and enter values.....	33
Filter the displayed list.....	33
Set values through the initialize parameter list.....	34
Batch field validation.....	34
Index fields.....	35
Index field list filtering.....	35
Required fields.....	35
Hidden fields.....	35
Set index field values without connecting to UI.....	35
Index field validation.....	36
Skin the generated table.....	37
Handle events.....	37
ImageProxy properties and methods.....	39
Handle errors.....	43
Handler: onScanError(msg, params).....	43
Handler: onScanClientReady().....	45
Set scanning options.....	47
Upload Options.....	57
Connect to the Web Document Viewer.....	58
Licensing.....	60
File Formats and File Options.....	60
Use VirtualReScan (VRS).....	60
Test your application.....	61
Test in Edge, Firefox and Chrome.....	61
Test for error conditions.....	61
Troubleshoot scanning problems.....	62
Uninstall Web Capture MSI.....	64
Client API reference.....	64
Atlasoft.Controls.Capture.WebScanning.....	64
Atlasoft.Controls.Capture.CaptureService.....	72
Chapter 5: Web Document Viewer.....	75
Chapter 6: Program with DotPdf.....	76

Mathematical model.....	77
Transformations.....	78
PdfGeneratedDocument.....	80
Pages.....	80
Standard page sizes.....	81
Create stationery.....	81
Clipping.....	83
Colors.....	84
Rendering.....	85
Resources.....	85
Font resources.....	86
Type 1 symbol font encoding.....	87
Embed fonts.....	90
Color space resources.....	90
Image resources.....	91
Template resources.....	92
Shapes.....	92
PdfPath.....	92
PdfRectangle.....	95
PdfRoundedRectangle.....	96
PdfCircle.....	96
PdfArc.....	96
PdfImageShape.....	96
PDF text shapes.....	98
PdfTable.....	99
PdfTemplateShape.....	100
PostnetBarcodeShape.....	103
GSave / GRestore.....	103
Transform.....	104
Marked content.....	104
Make custom shapes.....	105
Round trip documents.....	108
Integrate with Web Capture.....	109
Actions.....	110
PdfAction.....	111
Go To View actions.....	111
URI actions.....	112
JavaScript actions.....	112

Sound actions.....	112
Show/Hide action.....	113
Named actions.....	114
Submit Form Actions.....	115
Reset Form Action.....	115
Annotations.....	115
Properties common to all annotations.....	116
Properties common to all mark up annotations.....	119
Properties common to all widget annotations.....	120
General annotations.....	121
Markup annotations.....	124
Widget annotations.....	135
Use annotations.....	144
Place an annotation.....	144
Create an annotation with a custom border.....	145
Add a pop-up to a markup annotation.....	146
Create an annotation with transparency.....	147
Skin an annotation.....	148
Make an annotation with a rollover appearance.....	149
Make a sticky note annotation.....	150
Add a review state to a sticky note.....	150
Make a highlight annotation.....	151
Set a redaction area.....	155
Use JavaScript to calculate values.....	156
PDF Forms.....	158
PdfForm.....	159
Node form fields.....	160
Leaf form fields.....	160
Visiting nodes.....	160
Form actions.....	161
Merge PDF forms.....	162
Import pages.....	163
Merge forms.....	163
Default merging.....	164
DotPdf repair.....	164
DotPdf repair process.....	164
Detect errors.....	165
Repair errors.....	166

Repair events.....	166
Repair filtering.....	167
Structure options.....	168
Array options.....	170
Property repair.....	170
Digital signatures.....	171
Certify documents.....	172
Get signer information.....	174
Document signing operations.....	177
Customize signature appearance.....	180
Certify a document with PdfDocument.....	181
Determine if a document is certified or signed.....	181
Fill fields of a certified document.....	182
Sign a document with an existing signature.....	182
Add a signature to a document.....	182
Linearized PDF.....	183
PdfDocument and PdfGeneratedDocument integraton.....	183
PdfEncoder integration.....	184
PDF/A.....	184
PDF/A in PdfDocument.....	184
PDF/A data in PdfDocumentMetadata.....	187
PDF/A in PdfGeneratedDocument.....	187
PDF 2.0.....	195
Document upgrade to PDF 2.0.....	196
Chapter 7: BarcodeReader.....	197
Use the BarcodeReader.....	198
Reading a bar code.....	198
Read a bar code with options set.....	199

Preface

The *Kofax Web Capture Developer's Guide* contains information about how to install and customize your Kofax Web Capture 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 Kofax Web Capture documentation set includes the following:

- [API Reference](#): Gives the complete Kofax Web Capture class library in online help format.
- [API Reference \(.chm file\)](#): Gives the complete Kofax Web Capture class library for offline use.
- [Kofax Web Capture Release Notes](#): Contains late-breaking product information not included in this guide.


Training

Kofax offers both classroom and online training to help you make the most of your product. To learn more about training courses and schedules, visit the [Kofax Education Portal](#) on the Kofax website.

Getting help with Kofax products

The [Kofax Knowledge Portal](#) repository contains articles that are updated on a regular basis to keep you informed about Kofax products. We encourage you to use the Knowledge Portal to obtain answers to your product questions.

To access the Kofax Knowledge Portal, go to <https://knowledge.kofax.com>.

 The Kofax Knowledge Portal is optimized for use with Google Chrome, Mozilla Firefox, or Microsoft Edge.

The Kofax Knowledge Portal provides:

- Powerful search capabilities to help you quickly locate the information you need.
Type your search terms or phrase into the **Search** box, and then click the search icon.

- Product information, configuration details and documentation, including release news.
To locate articles, go to the Knowledge Portal home page and select the applicable Solution Family for your product, or click the View All Products button.

From the Knowledge Portal home page, you can:

- Access the Kofax Community (for all customers).
On the Resources menu, click the **Community** link.
- Access the Kofax Customer Portal (for eligible customers).
Go to the [Support Portal Information](#) page and click **Log in to the Customer Portal**.
- Access the Kofax Partner Portal (for eligible partners).
Go to the [Support Portal Information](#) page and click **Log in to the Partner Portal**.
- Access Kofax support commitments, lifecycle policies, electronic fulfillment details, and self-service tools.
Go to the [Support Details](#) page and select the appropriate article.

Chapter 1

Deploy Kofax Web Capture

Kofax Web Capture does not contain COM components to register, and no Registry modifications are required to use the SDK. To deploy the SDK, copy Kofax Web Capture assemblies alongside your EXE.

Visual C++ Runtime dependencies

Kofax Web Capture is distributed in several configurations, which are listed in the *Kofax Web Capture Technical Specifications*.

Deploy Kofax Web Capture in ASP.NET

When deploying Kofax Web Capture in an ASP.NET application, the Kofax Web Capture license file must be located in the bin directory of the application.

Dependencies using Kofax Web Capture class library

The following files must be included on the server that uses Kofax Web Capture. This is all that is required when using the class library only:

- `Atalasoftware.dll`
- `Atalasoftware.Lib.dll`
- `Atalasoftware.Shared.dll`

All of these files must be placed in the application's `bin` folder.

Dependencies using Kofax Web Capture with WebControls

The following files must be included on the server that uses Kofax Web Capture with WebControls:

- `Atalasoftware.dll`
- `Atalasoftware.WebControls.dll`
- `Atalasoftware.Lib.dll`
- `Atalasoftware.Shared.dll`
- `Atalasoftware.Pdf.dll`
- `Atalasoftware.PdfReader.dll`
- `Atalasoftware.PdfDoc.Bridge.dll`

- `Atalasoftware.dotImage.PdfDoc.dll`
- `Atalasoftware.dotImage.Ocr.dll`
- `Atalasoftware.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 Atalasoftware 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 6 or later, the License Compiler is not supported. Instead, use the Atalasoftware License Compiler (`AtalasoftwareLicenseCompiler.exe`) provided with Kofax Web Capture to transform and embed the license binary resource. Just like the License Compiler, the Atalasoftware 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 Atalasoftware License Compiler can be run separately, and it uses the same command-line arguments as the License Compiler, as in this example:

```
AtalasoftwareLicenseCompiler.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 `Atalasoftware.dotImage.AtalasoftwareLicenseCompiler.x86` or `Atalasoftware.dotImage.AtalasoftwareLicenseCompiler.x64` NuGet package for .NET 6 project. The NuGet package includes `AtalasoftwareLicenseCompiler.exe` and the appropriate targets and instructions for `*.licenses` generation. Targets are added to the `.csproj` file during compilation.

To use the Atalasoftware License Compiler, follow these steps:

1. Install the NuGet package, either `Atalasoftware.dotImage.AtalasoftwareLicenseCompiler.x86` or `Atalasoftware.dotImage.AtalasoftwareLicenseCompiler.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 Atalasoftware website.

3. Build the project.

During compilation, the following takes place:

- a. The `AtalasoftwareLicenseCompiler.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.

i 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 2

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/Atalasoftware/web-capture-service/tree/master/docs>.

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.

Make sure you read the *Kofax Web Capture Technical Specifications* for supported products and versions.

Kofax Web Capture demos

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

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. 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 6

If you are using .NET 6, you need to modify the MSBuild project file (which has a .csproj extension) to add Windows Forms support and enable Kofax Web Capture 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 4.6.2 (64-bit): C:\Program Files (x86)\Kofax\Web Capture 11.5\bin\4.6.2\x64
- .NET Framework 4.6.2 (32-bit): C:\Program Files (x86)\Kofax\Web Capture 11.5\bin\4.6.2\x86
- .NET Framework 3.5 (64-bit): C:\Program Files (x86)\Kofax\Web Capture 11.5\bin\3.5\x64
- .NET Framework 3.5 (32-bit): C:\Program Files (x86)\Kofax\Web Capture 11.5\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

Web Capture comes with two sets of resources: WebCapture and WebDocumentViewer. In a default .Net installation, these directories are located in C:\Program Files (x86)\Kofax\Web Capture 11.5\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 `Atalasoftware.Imaging.WebControls`). Your handler should resemble the following example.

C#

```
using Atalasoftware.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.

 Kofax Import Connector handlers are only supported with .NET Framework 3.5 and 4.6.2.

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 Atalasoftware.Controls.WebDocumentViewer({
            parent: $(''.atala-document-container'),
            toolbarparent: $(''.atala-document-toolbar'),
```

```

        serverurl: 'WebDocViewerHandler'
    });

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

    Atalasoftware.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 `INSTALLSERVICE` command line option as shown below:

```
msiexec /I Kofax.WebCapture.Installer.msi INSTALLSERVICE=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.

Chapter 3

Web scanning server

The following sections cover the server-side handlers for displaying and processing scanned documents. This also includes sections on forwarding documents to remote services such as Kofax Import Connector.

i Kofax Import Connector handlers are only supported with .NET Framework 3.5 and 4.6.2.

Troubleshoot Web Capture Handlers

If you have difficulty getting this project to run, consider using a tool like Fiddler Web Debugger, which allows you to monitor the HTTP requests and responses that pass between the web scanning controls, and the handlers on the back-end. Exceptions in your handlers will present as 500 errors and will likely contain the exception information embedded in the response. Other errors in your handlers will present as JSON data in the response that does not contain the data you expect.

i When implementing the web capture handler, all of the data returned from the methods you override is converted into an equivalent JSON representation. Examining the JSON is an easy way to verify outside of the debugger that you are returning the right data.

Client errors will usually present as JavaScript errors. Use your browser's equivalent of F12 tools to access the JavaScript console to check for errors. The most likely source of errors is not correctly including all of the necessary web resources, not initializing the controls correctly, or running your page in an incompatible browser.

No documents appear in the Web Document Viewer after scanning

If you have successfully deployed your application to an application server with a Web Document Viewer, but the viewer does not appear to work, then the web document viewer handler may be failing and returning an HTTP 500 code. Use a tool such as Fiddler Web Debugger to see if this is the case. Check the error logs provided by your application server for more detailed information.

If the handler is returning an HTTP 200 code and there is no image, examine the JSON returned in the response. It may contain a key-value pair such as: "error": "There was a problem with your license..."

If this is the case, an SDK license is required, but has not been properly applied to your handler.

Another reason for not seeing anything is that you forgot to create the upload directory in which scanned images are sent for viewing.

Extend the KicHandler

1. Open the application project in your development environment.
2. Add a generic handler to the project.
3. Extend the handler that was just created with the KicHandler found in the Atalsoft.dotImage.WebControls.Capture namespace that in the Atalsoft.dotImage.WebControls.dll assembly.

Sample code snippet

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


namespace TheApplicationNamespace
{
    public class MyKicHandler : KicHandler
    {
    }
}
```

No other modifications are necessary.

Connect to Kofax Import Connector services

These instructions are for configuring an application to connect to an existing Kofax Import Connector server.

For information on configuring Kofax Import Connector, see [Configure Kofax Import Connector](#)

 Kofax Import Connector handlers are only supported with .NET Framework 3.5 and 4.6.2.

Modify `web.config`

To connect to Kofax Import Connector a WCF endpoint, a binding must be added to the application's `web.config`, or `app.config`. In the provided example a standard `basicHttpBinding` will be used, but other appropriate binding types are possible choices to use as the WCF binding.

Set the WCF EndPoint

`web.config` endpoint

```
<system.serviceModel>
  <client>
    <endpoint address="http://servername.domain.com:[http or https port]/
soap/tsl" binding="basicHttpBinding" bindingConfiguration="importBinding"
contract="importPortType" name="importPort" />
  </client>
</system.serviceModel>
```

Set the binding

HTTP

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="importBinding" closeTimeout="00:01:00" openTimeout="00:01:00"
receiveTimeout="00:10:00" sendTimeout="00:01:00" allowCookies="false"
bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
maxBufferSize="1655360" maxBufferPoolSize="15242880" maxReceivedMessageSize="1655360"
messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
useDefaultWebProxy="true">
        <readerQuotas maxDepth="32" maxStringContentLength="256000"
maxArrayLength="16384" maxBytesPerRead="4096" maxNameTableCharCount="16384" />
        <security mode="None">
          <transport clientCredentialType="None" proxyCredentialType="None" realm="" />
          <message clientCredentialType="UserName" algorithmSuite="Default" />
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
</system.serviceModel>
```

HTTPS

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="importBinding" closeTimeout="00:01:00" openTimeout="00:01:00"
receiveTimeout="00:10:00" sendTimeout="00:01:00" allowCookies="false"
bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
maxBufferSize="1655360" maxBufferPoolSize="15242880" maxReceivedMessageSize="1655360"
messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
useDefaultWebProxy="true">
        <readerQuotas maxDepth="32" maxStringContentLength="256000"
maxArrayLength="16384" maxBytesPerRead="4096" maxNameTableCharCount="16384" />
        <security mode="Transport">
          <transport clientCredentialType="None" proxyCredentialType="None" realm="" />
          <message clientCredentialType="UserName" algorithmSuite="Default" />
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
</system.serviceModel>
```

Specify the Kofax Import Connector endpoint

The KicHandler will connect to Kofax Import Connector through a services endpoint defined in a WSDL file provided by the KIC server. The location to the WSDL file must be specified in a WebInitParam annotation within your handler's WebServlet annotation, or in an init-param tag within your handler's web.xml servlet tag.

The following example includes the necessary configuration to map your handler to a public URL.

XML - configuring servlet in web.xml

```
<servlet>
  <servlet-name>KicHandler</servlet-name>
  <servlet-class>com.mydomain.mypackage.MyKicHandler</servlet-class>
  <init-param>
    <param-name>KicWsdLocation</param-name>
```




```
<param-value><http>://<server>:<port>/file/import.wsdl</param-value>
</init-param>
</servlet>
<servlet-mapping>
  <servlet-name>KicHandler</servlet-name>
  <url-pattern>/kichandler</url-pattern>
</servlet-mapping>
```

Modify the `KicWsdlLocation` value to point to the actual location of `import.wsdl` on your Kofax Import Connector server. Replace `<http>`, `<server>`, and `<port>` with values appropriate for your Kofax Import Connector server. If the WSDL file is not provided at the default location, ask your Kofax Import Connector administrator for assistance.

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 Web Capture 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 4.6.2.

Required license

For the KIC server to accept documents imported from the Web Capture 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 Web Capture Web scanning control connects via KIC's service via a server-side handler that extends the `KicHandler` found in the `Atalsoft.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>
```

Chapter 4

Web scanning client

Use the Web Scanning Control to add the control to a web page, configure, and connect to server-side handlers.

Initialize the control on the client

On the page of your application that will support scanning, you need to include the capture javascript, and initialize scanning and upload/import.

Include WebCapture Javascript

Add the needed includes in the `<head>` section of the document, like this: `<script src="jquery-3.4.1.min.js" type="text/javascript"></script> <script src="atalaWebCapture.js" type="text/javascript"></script>`

If you placed the capture resources in a subfolder under your application, you will need to modify the `src` attribute to the appropriate relative path.

Initialize

There are two parts of the control that need to be initialized, both can be initialized in the same script tag.

Initialize scanning

Scanning is initialized with a call to:

```
Atalasoftware.Controls.Capture.WebScanning.initialize({params})
```

This function takes a comma-separated list of arguments including the URL of the handler used on the server, event handlers, scanning options sent to the control, and error handling for the client. All of the arguments are optional except the URL of the server handler.

See [Client API reference](#) for details.

Initialize the Kofax Import Connector connection

The connection to the Kofax Import Connector server is initialized with a call to:

```
Atalasoftware.Controls.Capture.CaptureService.initialize({params})
```

This needs to be called in addition to the `WebScanning.initialize` function to populate any client UI controls with Kofax Import Connector `contentTypes`, and `contentTypeDescriptions`. It requires a handler argument, and accepts optional custom error handlers. When no selection dropdowns, or

other selection UI is desired values for the required `contentType`, and `contentTypeDescriptionName` are also set in the parameter list.

Example

The following example script shows both these objects being initialized:

Code snippet

```
<script type="text/javascript">
  // Initialize Web Scanning and Web Viewing
  $(function() {
    try {

      Atalasoftware.Controls.Capture.WebScanning.initialize({
        handlerUrl: 'KicWebCaptureHandler.ashx',

        onScanError: function(msg, params) { appendStatus(msg); },

        onScanStarted: function(eventName, eventObj) { appendStatus('Scan
        Started'); },
        onScanCompleted: function(eventName, eventObj) { appendStatus('Scan
        Completed: ' + eventObj.success); },

        onUploadError: function(msg, params) { appendStatus(msg); },
        onUploadStarted: function(eventName, eventObj)
        { appendStatus('Upload Started'); },
        onUploadCompleted: function(eventName, eventObj) {
          appendStatus('Upload Completed: ' + eventObj.success);
          if (eventObj.success) {
            viewer.OpenUrl('atala-capture-upload/' +
            eventObj.documentFilename);
          }
        },
        scanningOptions: { pixelType: 0 }
      });

      Atalasoftware.Controls.Capture.CaptureService.initialize({
        handlerUrl: 'KicWebCaptureHandler.ashx',
        onError: function(msg, params) { appendStatus(msg + ': ' +
        params.statusText); }
      });
    }
    catch (error) {
      //Do something with the error caught. Default is to just go
      //to the javascript error console in the browser.
    }
  });
}</script>
```

Example when no `contentType`, or `contentTypeDescriptionName` UI is desired:

```
<script type="text/javascript">
  // Initialize Web Scanning and Web Viewing
  $(function() {
    try {
      Atalasoftware.Controls.Capture.WebScanning.initialize({
        handlerUrl: 'KicWebHandler.ashx',

        onScanError: function(msg, params) { appendStatus(msg); },
        onScanStarted: function(eventName, eventObj) { appendStatus("Scan
        Started"); },
        onScanCompleted: function(eventName, eventObj) {
          appendStatus("Scan Completed: " + eventObj.success); },

```

```
        onUploadError: function(msg, params) { appendStatus(msg); },
        onUploadStarted: function(eventName, eventObj)
    { appendStatus("Upload Started"); },
        onUploadCompleted: function(eventName, eventObj) {
            appendStatus("Upload Completed: " + eventObj.success);
            if (eventObj.success) {
                appendStatus("atala-capture-upload/" +
eventObj.documentFilename);
                viewer.OpenUrl("atala-capture-upload/" +
eventObj.documentFilename);
                Atalasoftware.Controls.Capture.CaptureService.documentFilename
= eventObj.documentFilename;
            }
        },
        scanningOptions: { pixelType: 1 }
    });

    Atalasoftware.Controls.Capture.CaptureService.initialize({
        handlerUrl: 'KicWebHandler.ashx',
        //The required BatchClassName.
        contentType: 'AtalasoftwareEngineering',
        //The ContentTypeDescriptionName must be in the form of
        //'DocumentClassName / FormType'.
        contentTypeDescriptionName: 'PointOfOrigin / ClaimForms',
        onError: function(msg, params) { appendStatus(msg + ": " +
params.statusText); },
        onImportCompleted: function(params) { appendStatus(params.id + ": "
+ params.status ); },
        onTrackStatusReceived: function(params) {appendStatus("Import
status: "+ params); }
    });
    }
    catch (error) {
        appendStatus("Thrown error: " + error.description);
    }
    });
</script>
```

Connect to UI controls

The Web Scanning control automatically finds and connects to UI controls using their class="" identifiers, so it is sufficient for you to add, lay out and style the UI controls required by your application, and assign the appropriate classes to those controls.

Valid classes include

- atala-scan-button
- atala-scanner-list
- atala-content-type-list, and
- atala-content-type-document-list

Examples of UI controls

Scan button

```
<input type="button" class="atala-scan-button" value="Scan" />
```

This button will automatically be enabled when scanning is possible, and disabled otherwise.

When the user clicks this button, a scan is initiated with current scanner and document selections.

Scanner device list

This control is loaded with the list of available TWAIN devices, and the current visible selection will be used when a scan is initiated.

```
<select class="atala-scanner-list" disabled="disabled" name="scannerList" style="width:194px">  
  <option selected="selected">(no scanners available)</option>  
</select>
```

If the contents of the control were changed programmatically (for example, WIA scanners were filtered out of the list using jQuery), the currently selected value may not reflect the currently selected TWAIN scanner. To synchronize the two, the onchange event should be manually triggered on the control.

If scanning is not possible or there are no scanners available, this control will be disabled.

Kofax Import Connector content types

```
<select class="atala-content-type-list" style="width:385px"></select>
```

This control is automatically loaded with the list of available content types provided by the Kofax Import Connector (KIC) server, and the current visible selection is used when an import is initiated.

If a connection cannot be established to the KIC server, this control is disabled.

Kofax Import Connector content type descriptions

```
<select class="atala-content-type-document-list" style="width:385px"></select>
```

This control is automatically loaded with the list of available content type descriptions as provided by the Kofax Import Connector server, and the current visible selection is used when a scan is initiated.

If a connection cannot be established to the Kofax Import Connector server, this control is disabled.

Kofax Import Connector import button

```
<input type="button" class="atala-import-button" value="Import" />
```

This button is automatically enabled if KIC import is possible, and is disabled otherwise.

When the user clicks it, a KIC import (of the last scanned document) is initiated.

Kofax Import Connector track import button

```
<input type="button" class="atala-track-import-button" value="Track Import" />
```

When the user clicks it, the status of the last import is returned.

Kofax Import Connector index fields

```
<div class="atala-indexfield-list" style="width:600px; height:250px; overflow:scroll; border:solid 1px #CCC;"></div>
```

A table with the index field names for a label, and text input will be constructed at this div.

Kofax Import Connector batch fields

```
<div class="atala-batchfield-list" style="width:600px; height:250px; overflow:scroll; border:solid 1px #CCC;"></div>
```

A table with the batch field names for a label, and a text input will be constructed at this div.

Kofax Import Connector import with index fields

```
<input type="button" class="atala-import-index-field-button" value="Import with IndexFields" />
```

This button is automatically enabled if import is possible, and is disabled otherwise.

When the user clicks it, an import into Kofax Import Connector is initiated for the last scanned document, along with any entered index field values.

*One should also note, that any "button" that has a type="submit" will create an empty POST that will override any POST or GET that the scanning control sends.

Filter selection lists

Use the removedContentTypes, and removedContentTypeDescriptions initialization parameter to filter the lists displayed in the atala-contenttype-list, and atala-contenttype-document-list.

Example

```
Atalasoft.Controls.Capture.CaptureService.initialize({  
  handlerUrl: 'KicWebHandler.ashx',  
  loosePages: "true",  
  removedContentTypes: "KfxSingleMessageBatch",  
  removedContentTypeDescriptions: "KfxMultiDocument / NWestMulti",  
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },  
});
```

Connect controls with no UI

When using the client controls to connect to Kofax Capture through Kofax Import Connector, it is not desired to have the content type/repository name selection boxes on the page, because then a selected value can be passed through the capture service's initialize parameters.

Example

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  contentType: 'AtalasoftEngineering',
  batchFields: "BatchField1:value1, BatchField2:value2",
  contentTypeDescriptionName: 'Engineering / TestDocument',
  indexFields: "IndexField1: value1, IndexField2: value2",
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
});
```

At a minimum the contentType must be specified for all document imports into Kofax Capture through Kofax Import Connector.

Import loose pages

When connecting to Kofax Capture (KC) via the Kofax Import Connector (KIC) a loose page can be imported by not selecting or specifying a document class/ form type combination when importing a document via the scanning client, and by having the loosePages initialization parameter set to true. By default this parameter is set to 'false'. When set to 'true' by default a blank option will be added to the atala-contenttype-document-list (when available).

Example With UI

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  loosePages: "true, Loose Page: Test",
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
  onImportCompleted: function(params) { appendStatus(params.id + ": " +
    params.status); }
});
```


i With UI assumes that batch fields will be displayed, along with the atala-contenttype-list, and atala-contenttype-document-list.

Without UI

```
Atalasoftware.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  contentType: 'AtalasoftwareEngineering',
  batchFields: "BatchField1:123, BatchField2:321",
  contentTypeDescriptionName: '',
  loosePages: "true",
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); }
});
```

Batch fields

Batch Fields are much like index fields. They have the same hidden, and required class associated with them, and can be used to add meta data to loose pages imports into Kofax Capture. See [Index fields](#).

Display and enter values

Batch fields get displayed in the same <div> as index fields. See instructions on adding the index field <div> to a page: [Connect to UI Controls](#).

Filter the displayed list

To filter the batchfields that get displayed in the client page UI specify the batch fields to be displayed by setting the displayedBatchFields parameter.

Example

```
Atalasoftware.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  contentType: 'AtalasoftwareEngineering',
  displayedBatchFields: "BatchField1, BatchField2",
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
  onImportCompleted: function(params) { appendStatus(params.id + ": " +
    params.status); },
});
```

In the example above only BatchField1, and BatchField2 would be displayed in the generated table.

Set values through the initialize parameter list

When no indexfield div has been added to a page, but batch field values still need to be set they can be passed through the capture service's initialize method.

Example

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  contentType: 'AtalasoftEngineering',
  batchFields: "BatchField1:value1, BatchField2:value2",
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
});
```

In the above example the two batch fields (BatchField1, and BatchField2) for the AtalasoftEngineering batch class have each had a value set. The batchFields parameter takes a string where each bath field name value pair are comma separated, and the batch field name, and value are colon separated.

Batch field validation

There are two capture service initialization parameters that can be used to handle batch field validation on the client. There is the error handling event, onBatchFieldImportValidationError, and the custom client validation parameter, onBatchFieldTypeValidationStatus.

Example

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KicWebHandler.ashx',
  contentType: 'AtalasoftEngineering',
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
  onImportCompleted: function(params) { appendStatus(params.id + ": " +
params.status); },
  onTrackStatusReceived: function(params) { appendStatus("Import status: " +
params); },
  onBatchFieldImportValidationError: function(params) { appendStatus("BatchField
Validation Error:" + params); },
  onBatchFieldTypeValidationStatus: function(params)
{ customValidationFunction(params); }
});
```

Index fields

Index field list filtering

As part of the `Atalasoftware.WebScanning.CaptureService.Initialize`'s list of parameters that get passed in includes a mechanism to provide a list of the index fields that should be displayed in the generated table of index fields.

Example

```
Atalasoftware.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KICDemoHandler.ashx',
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
  onImportCompleted: function(params) { appendStatus(params.id + ": " +
    params.status); },
  onTrackStatusReceived: function(params) { appendStatus("Import status: " + params); },
  displayedIndexFields: 'Name, Title, Content Type'
});
```

In the above example the parameter "displayedIndexFields" specifies the list of index field that should be included for display. Only the index fields with named: "Name", "Title", and "Content type" will be displayed.

Required fields

Kofax Import Connector has required index fields that must be set so that a document import is successful. When the list of index fields is retrieved from the server the required field information is included with that information, and a class is added to the label of that index field. The class that gets added is:

```
class="atala-indexfield-required"
```

An example that shows how to use this class to add an asterisk to the beginning of the label can be found in the WebCapture demo included with the installation.

Hidden fields

Index fields in KC have an optional flag called hidden, when this is set to "true" in KC the field that it is applied to will have the following class applied to it:

```
class="atala-field-hidden"
```

Set index field values without connecting to UI

As with content types, and content type descriptions indexfields can also be passed in through to the import POST parameters via the `Atalasoftware.Controls.Capture.CaptureService.initialize` call.

Example

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KICDemoHandler.ashx',
  contentType: 'Documents',
  contentTypeDescriptionName: 'Document',
  indexFields: "Name: Adam, Title: Q3 results , Content Type: ",
});
```

The "indexFields" parameter takes a string where the index fields are comma separated with the name of the particular index field separated from the value being assigned to it by a ':', so "indexField1: indexfieldValue1, indexfield2:indexfieldvalue2, ..."

Index field validation

Client side validation

Any index field value validation beyond checking that required fields have values prior to import should be handled via the "onIndexFieldTypeValidationStatus" parameter in the `Atalasoft.Controls.Capture.CaptureService.initialize` setup.

```
Atalasoft.Controls.Capture.CaptureService.initialize({
  handlerUrl: 'KICDemoHandler.ashx',
  contentType: 'Documents',
  contentTypeDescriptionName: 'Document',
  onError: function(msg, params) { appendStatus(msg + ": " + params.statusText); },
  onImportCompleted: function(params) { appendStatus(params.id + ": " +
    params.status); },
  onTrackStatusReceived: function(params) { appendStatus("Import status: " + params); },
  onIndexFieldImportValidationError: function(params) { appendStatus("Index field
    validation error:" + params); },
  onIndexFieldTypeValidationStatus: function(params) { fieldTypeValidation(params); }
});
```

Where the function `fieldTypeValidation(params)` is a function elsewhere in the page that performs the additional index field input validation that could be run before importing. An example of this function is available in the WebCapture demo project included with the installer.

Server side validation

Validation on the server side checks that the index field input values are of the correct form for the culture specified in the `web.config`, or `app.config`. By default the handler will use the default culture of the server.

Handling validation error events in the client

As with the other import, and track status events index field validation has an `onIndexFieldImportValidationError` event that can be used to return information to the client in the case that a input value has been deemed invalid. See the WebCapture demo for an example.

Skin the generated table

It is possible to "skin" the generated table of index fields to suit the needs of the design aesthetic of the site.

Generated Table of index fields

Once connected to the UI (see [Connect to UI Controls](#)) a table will be added as a child to the `<div class="atala-indexfield-list"/>` with `id="atala-indexfield-table"` applied to it. The table has an specific id applied to it as do the index field labels, and input fields. Each input field will have an id applied to it as well, and will be in the form of `id="<indexfieldname>_inputId"`

Required index field values

Required index field have a class applied to them, `class="atala-indexfield-required"` for an example of how to use this to apply a red '*' to the beginning of the label name see the Web Capture demo included with the installation.

Handle events

The `Atalasoftware.Controls.WebScanning` control has the following events that can be used in the client:

- `onScanError`
- `onScanStarted`
- `onImageAcquired`
- `onScanCompleted`
- `onScanClientReady`
- `onUploadError`
- `onUploadStarted`
- `onUploadCompleted`

To use one, some, or all of the events add them to the `Atalasoftware.Controls.WebScanning.initialize` method's argument list. See [Client API reference](#).

An example where each event is used:

Initialize WebScanning with Event Handlers

```
try {
    Atalasoftware.Controls.Capture.WebScanning.initialize({
        handlerUrl: 'TestCaptureHandler.ashx',

        onScanError: function(msg, params) { appendStatus(msg); },
        onScanClientReady: function() { appendStatus('Scan-Client Ready'); },
        onScanStarted: function(eventName, eventObj) { appendStatus('Scan Started'); },
    });
}
```

```
    onImageAcquire: function(eventName, imageProxy) { appendStatus('Image
Acquired'); },
    onScanCompleted: function(eventName, eventObj) {
        appendStatus('Scan Completed: ' + eventObj.success);
    },
    onUploadError: function(msg, params) { appendStatus(msg); },
    onUploadStarted: function(eventName, eventObj) { appendStatus('Upload
Started'); },
    onUploadCompleted: function(eventName, eventObj) {
        appendStatus('Upload Completed: ' + eventObj.success);
        if (eventObj.success) {
            viewer.OpenUrl('atala-capture-upload/' + eventObj.documentFilename);
        }
    }
});
}
catch (error) {
    appendStatus("WebScanning initialization error: " + error.description);
}
```

Handler: onScanError(msg, params)

See [Handling errors](#)

Handler: onScanClientReady()

See [Handling errors](#)

Handler: onScanStarted(eventName, eventObj)

Called when scanning starts.

i Always followed by a call to onScanCompleted, even if the scan fails or is aborted.

Handler: onImageAcquired(eventName, imageProxy)

This handler will be called during scanning each time an image is received from the scanner and processed by Web Capture.

i If blank images are being discarded (the discardBlankPages scanning option has been set to true), any image that is determined to be 'blank' will be discarded during post-processing. This handler is not called for such images.

The second parameter is a 'proxy' object representing the acquired image, with a limited set of properties and methods that can be used inside the handler.

i Do not retain the proxy object outside the onImageAcquired handler, it is not valid after the handler returns. Generally, the proxy is valid if the image wasn't discarded or cleared by calling the proxy.clear() method, and before a new scan/import has been started. In most cases, not using a proxy outside handler is preferred because it is less error-prone.

ImageProxy properties and methods

imageProxy.discard

If the handler sets `imageProxy.discard` to `true`, the image will be discarded when the handler returns. Use this feature if you are uploading or otherwise disposing of each incoming image yourself, and do not want Web Capture to collect and upload all the scanned images at the end of the scan job.

An Image may be persisted to the local WCS hard disk storage as an encrypted local file. In this case, a file identifier will be returned. The image object can be cleared on the server by using `proxy.clear()`. The local file is stored as an image in the specified format, so it makes sense to use the target format because this avoids re-compressing the image when it is read.

Also, it's possible to configure automatic local file generation using `scanningOptions.deliverables`. In this case the image proxy would have a `localFile` property with a created local file identifier.

imageProxy.originalImage

Identifier of the local file that stores original image obtained from scanner. See `deliverables` settings for details.

imageProxy.filename


Present only when the image represents an entire imported PDF file, during an `importFiles` operation. The property contains the full path of the imported file e.g. `C:\Users\Hugh McLarty\Documents\Kofax.pdf`.

imageProxy.barcodes

When bar coderecognition is enabled, the image object will have a property **barcodes** whose value is an array describing any bar code symbols found in that image.

Each entry in the bar codes array represents one *barcode* found in the image, and has these properties:

- **data** string the data decoded from the symbol, *excluding* start, stop.

 standard error-checking codes are always stripped.

If the checksums option for barcoding was set `true`, the optional checksums of Code 39, Codabar and I25 are checked and stripped.

- **symbology** string symbology of the symbol.
- **bounds** rectangle bounding box of symbol, in pixels from upper-left of page, with properties `top`, `left`, `right`, and `bottom`.
- **orientation** integer degrees the symbol is rotated clockwise from 'conventional' orientation. Unrecognized symbols, symbols not in an enabled symbology, and symbols that fail error-check, are not listed. When bar code recognition is not enabled, the `barcodes` property of the image will be either an empty array, or `null`.

imageProxy.patchCode

Presents patch code found on the page.

Values are 0, 1, 2, 3, 4, 6, or T. The '0' value means "no patch code detected". The values are 1-character strings. When patch code detection is not enabled, the patchCode of each acquired image is 0.

```
scanningOptions = { onImageAcquired: this._onImageAcquired, patchCodes: true }
...
function _onImageAcquired(eventName, image) {
  // Check patch code:
  if (image.patchCode == 'T') {
    // 'T' patch-code detected
  }
}
```

imageProxy.width, imageProxy.height

Contains the width and height in pixels of the image.

imageProxy.bitsPerPixel

Contains the number of bits used to represent a pixel (typically 1, 8, or 24).

imageProxy.pixelType

pixelType image data format:

- 0 B&W - bitonal
- 1 Grayscale (8-bit linear)
- 2 RGB Color (24-bit)
- 3 Indexed color (8-bit)

imageProxy.dpi

The resolution of the image, in DPI (Dots Per Inch), in an array of [*horizontal dpi, vertical dpi*].

imageProxy.sheetNo (optional)

If present, the index of the physical sheet within the scan job of which this is an image (front or back side). The first sheet scanned is index 0. If not present, this information could not be confidently determined e.g. the scan is from a device that does not feed sheets.

imageProxy.newSheet

If present, this is **true** when this image came from a different physical sheet than the preceding image if any. It is **false** if this image is the flip (back, bottom) side of the same sheet as the preceding image. If omitted, it means this information could not be confidently determined.

imageProxy.asBase64String (format [, options[, callback]])

Returns the image formatted in the specified file format+options and encoded into a base64 string. (Everything is done in memory, no actual file is created on the client system.)

i Imported PDF files can only be requested in "pdf" format, no format conversion is allowed, options are ignored.

imageProxy.clear()

Resets the image object to an empty state, releasing any (possibly large) internal memory being used to store pixel data.

imageProxy.thumbnail(w, h)

The creation of a thumbnail image is available as a method thumbnail on the image object passed to the onImageAcquired handler. The thumbnail method takes a maximum width and height in pixels, and returns a copy of the base image, scaled down proportionally to not exceed the given width and height. The ratio of width to height in the base image is preserved as closely as possible in the thumbnail. If the base image does not need to be scaled down to fit within the given dimensions, it is simply copied.

The object returned by imageProxy.thumbnail(w,h) is itself an image object and has the same methods and properties. Note: 1-bit B&W images are rendered as grayscale thumbnails.

Code Example: thumbnail

```
scanningOptions = { onImageAcquired: this._onImageAcquired }
...
function _onImageAcquired(eventName, image) {
    // Make a 32x48 pixel (maximum) thumbnail of the acquired image:
    var thumbnail = image.thumbnail(32, 48);
    // get thumbnail in JPG format encoded as Base64
    var thumbData = thumbnail.asBase64String('jpg');
}
```

imageProxy.saveEncryptedLocal(format [,options[, callback]])

This method writes the image using the specified file format and options into an encrypted local file. Returns a unique identifier (a relatively meaningless string) representing that saved file. The name of the local file is automatically generated (partly to guarantee no conflicts), and the file is stored in the current user's local application data, under %USERPROFILE%\AppData\Local\Kofax\WebCapture\Persistent.

i Imported PDF files are always saved in "pdf" format, ignoring the format parameter. Any options are ignored.

If SymmetricEncryptionKey has not been set, this throws an exception.

imageProxy.asBase64String(fmt)

This method returns a base-64 encoded file containing the just-received image, in the file-format specified by the `fmt` parameter. The `fmt` parameter must be either the string, `png`, `bmp`, `tif` or `jpg`. Note that `jpg` won't work if you are receiving B&W images, because JPEG files can only hold grayscale or RGB color images.

This method is useful if you want to store or upload each scanned image separately as it arrives.

```
asBase64String(fmt, [options], [callback])

options: {
  [jpegCompression] : bool,
  [quality]: number
};
```

callback: function(base64){}with service architecture each image or file function have synchronous and asynchronous versions, depending whether callback function is passed as last parameter.

Synchronous versions are left for compatibility. Async is the preferred choice. Synchronous versions are not supported in IE8/9. Calling them will issue a sync request and execution won't be blocked.

Handler: onScanCompleted(eventName, eventObj)

Called when scanning ends, successfully or otherwise.

The `eventObj` has a property `success`. If it is true, the scan completed without error.

If `eventObj.success` is false, the scan was not fully successful, and there will be a string with more information in `eventObj.error.message`.

Usually when scanning fails, the `onScanError` handler will have already been called with a specific error message.

Handler: onUploadStarted(eventName, eventObj)

Called when an upload begins.

Handler: onUploadError(msg, params)

Called when an error is detected during upload to the server.

The `msg` parameter will be one of the following:

- `Atalasoftware.Controls.Capture.Errors.ajax` - could not create/initialize the XMLHttpRequest object.
- `Atalasoftware.Controls.Capture.Errors.serverNotResponding` - connection to the server timed out.
- `Atalasoftware.Controls.Capture.Errors.uploadError` - the `params` object will contain three properties: `responseStatus`, `response`, and `handlerUrl`.

Handler: onUploadCompleted(eventName, eventObj)

Called when an upload completes, whether successfully or not.

If the upload was successful, `eventObj.success` is true, and `eventObj.documentFilename` contains the unqualified name of the file in the upload directory on the server.

If the upload failed for some reason, `eventObj.success` is false. In this case, `onUploadError` will have been called to report the error.

Handle errors

By default all errors are sent to the javascript console in the browser. However, you can override this by specifying an error-handling function in the parameters to `Atalasoftware.Controls.WebScanning.initialize` and `Atalasoftware.Controls.CaptureService.initialize`. See [Client API reference](#).

This example shows the basic technique of specifying error-handling functions. There is a longer code example at the end of this section.

JavaScript

```
$ (function) () {
    try {
        Atalasoftware.Controls.Capture.WebScanning.initialize({
            handlerUrl: 'TestCaptureHandler.ashx',

            onScanError: function(msg, params) { appendStatus(msg); },
            onUploadError: function(msg, params) { appendStatus(msg); }
        });
    }
    catch (error) {
        appendStatus("WebScanning initialization error: " + error.description);
    }
});

function appendStatus(msg) {
    $('#status').append(''+msg+'');
}
```

This will display error messages to a div with `id=status`.

Handler: `onScanError(msg, params)`

The Web Capture service can be initialized with a scan error handler (see the Code Example at the end of this section), and that handler will potentially be called back by Web Capture with one of various scanning-related errors.

i It is essential to a well-functioning scanning application that you handle at least the `noPlugin` and `oldPlugin` errors. This is the standard way to deploy local service installer packages to the end user.

All WebCapture errors are string members of: `Atalasoftware.Controls.Capture.Errors`.

`Atalasoftware.Controls.Capture.Errors` contains quite a few properties, each of which represents a particular error. The values of each property is a string with an error description returned as the

`msg` parameter of the error callback. Error handling could be based on a comparison of `msg` with different `Atalasoftware.Controls.Capture.Errors` values.

Error strings could be localized by providing `params.localization` values to the `WebScanning.initialize` function.

The format of the localization object is same as for `Atalasoftware.Controls.Capture.Errors`. Specifically, corresponding keys will be replaced with new strings.

Below are the currently defined scanning-related errors, with an explanation of their cause and proper handling recommendations.

Errors.badBrowser

`Atalasoftware.Controls.Capture.Errors.badBrowser`

Fired in: Any unsupported browser

During: `Atalasoftware.Controls.Capture.WebScanning.initialize`

Cause: Web Capture detected that it is running in a browser or operating system it does not support. For a list of supported browsers and operating systems, see the Kofax Web Capture *Technical Specifications*.

How to Handle: Make sure your application displays the `msg` parameter to the handler, or your own equivalent message. If you also display the value of the `params` parameter, which will be a string, it would help a technical support specialist identify the browser causing the problem.

errors.noTwain

`Atalasoftware.Controls.Capture.Errors.noTwain`

Fired In: All browsers.

During: `Atalasoftware.Controls.Capture.WebScanning.initialize`

Cause: Support for the TWAIN protocol itself not found on the client computer.

Background: This error is extremely unlikely to happen on a typical end-user PC running Windows 8, because retail editions of Windows all include a copy of the TWAIN manager. However, a user on Windows Server and perhaps some other Server editions can be missing TWAIN which will cause this error. Ref:

Using scanners in Windows Server with TWAIN drivers might require the installation of Desktop Experience Pack.

How to Handle: You could just display the error string (the value of the `msg` parameter of the `onScanError` handler) or display your own message that TWAIN was not found on the computer.

errors.noPlugin

`Atalasoftware.Controls.Capture.Errors.noPlugin`

Fired In: All browsers

During: `Atalasoftware.Controls.Capture.WebScanning.initialize`

Cause: The required Web Scanning plugin is either not installed or is disabled.

Background: If the plugin is not installed, A notification displays.

errors.oldPlugin

`atalasoftware.Controls.Capture.Errors.oldPlugin`

Fired In: All browsers

During: `Atalasoftware.Controls.Capture.WebScanning.initialize`

Cause: The Web Scanning plugin is installed and enabled but Web Capture is designed to work with a newer version. For example, Web Capture might require plugin version 1.55, but detect that the browser has plugin version 1.42 installed. That would cause this error to be fired during initialization.

How to Handle: Similar to handling noPlugin above, but there will never be any prompting by the browser so you must present the user with a button or hyperlink to the correct plugin deployment package on your server. The filename of the appropriate download is passed to your error handler as `params.filename`.

errors.licensingError

`Atalasoftware.Controls.Capture.Errors.licensingError`

Fired In: All browsers

During: Asynchronously, after `Atalasoftware.Controls.Capture.WebScanning.initialize()`

Cause: When queried, the server did not return the expected JSON licensing information in a timely fashion.

How to Handle: If you see this during development, it suggests some server configuration problem, the `handlerUrl` passed to `WebScanning.initialize` isn't right, the server is actually off-line or not accessible, or (maybe, even) the licensing isn't right for Web Capture on the server.

Assuming you resolve any logical problems during development, if this error occurs after deployment it almost certainly represents a typical "server not responding" error, with all the usual causes.

Other errors

Other errors are possible, and additional errors may be added in future updates to DotImage. We recommend that you defend against that possibility by displaying the text of the error (the `msg` parameter to the `onScanError` handler) to the user, and offering them as much flexibility as possible - for example, by linking to a troubleshooting & support page that you can revise based on experience.

Handler: `onScanClientReady()`

This is a handler which can be passed to `WebScanning.initialize` alongside the `onScanError` handler. See the Code Example at the end of this section.

Called In: All browsers.

During: `Atalasoftware.Controls.Capture.WebScanning.initialize` OR at some later time if the scanning control needed to be downloaded and installed.

Cause: The client-side scanning control or plugin has just been successfully initialized and is operational. Note that this does not mean that any scanners were detected, working or otherwise, so the Scan button is not necessarily enabled.

Background: Alongside the `onScanError` handler, you can provide an `onScanClientReady` handler that will be called when client scanning services have been successfully initialized. Remember that the `onScanClientReady` handler may be called an arbitrarily long time after the `WebCapture.initialize` call.

Or this handler may never be called. Some possible causes:

- The user declines to install/approve the WebCapture plugin.
- Plugin installation fails.
- The browser or OS is unsupported.

How to Handle: This handler is a good place to clear any initialization error messages or prompts as discussed above. See the code example at the end of this section.

Code Example – Scan Error Handling

See also: [Initialize the Control on the Client.](#)

JavaScript

```
function scanErrorHandler(msg, params)
{
    appendStatus(msg);
    switch (msg) {
        case Atalasoft.Controls.Capture.Errors.badBrowser:
            promptHTML(
                msg + " <br />(" + params + ")");
            break;

        case Atalasoft.Controls.Capture.Errors.activeX:
            promptText(
                "The ActiveX Scanning Control needs to be installed or updated.\n" +
                "When prompted, please allow the Kofax Scanning Control to install
                itself.");
            break;

        case Atalasoft.Controls.Capture.Errors.noPlugin:
            promptHTML(
                "The Kofax Web Scanning plugin is not available. "+
                "Please follow any prompts to install it, or <a href='/"+params.filename
                +"'>Click Here</a><br />"+
                "If you are not prompted to install, the plugin may "+
                "be installed but disabled - please enable it.");
            break;

        case Atalasoft.Controls.Capture.Errors.oldPlugin:
            promptHTML(
                "The Kofax Web Scanning plugin is out of date.<br />"+
                "To download and install the latest version "+
                "<a href='/"+params.filename+"'>Click Here</a>");
            break;

        case Atalasoft.Controls.Capture.Errors.noTwain:
            promptText(
                "TWAIN is not installed on this computer.\n"+
                "Contact your system administrator.");
            break;

        default:
            promptText(msg);
            break;
    }
}

function scanClientReady() {
    promptText(""); // Clear the prompt box
}
```

```
// Initialize Web Scanning and Web Viewing
Atalasoft.Controls.Capture.WebScanning.initialize({
  // designate error handler:
  onScanError: scanErrorHandler,
  onScanClientReady: scanClientReady,
  // etc...
});
```

Set scanning options

In the `Atalasoft.Controls.WebScanning.initialize` method one of the parameters that can be passed as an argument in `scanningOptions`. These are various settings that apply to the scanner and to the way images from the scanner are processed.

Not all settings can be applied to every scanner. If a setting is used on a scanner that does not support it, the unsupported setting is simply ignored.

applyVRS

An option to specify if VRS should run or not in the client.

Default value: true.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { applyVRS: false }
});
```

autoRotate

Detects the orientation of the text in an image - right-side up, upside-down, sideways - and rotates the image so the text is upright.

If VRS is disabled, `autoRotate` is always disabled. If VRS is enabled, `autoRotate` is enabled by default but you can disable it with this option.

deskew

Deskew is scanning jargon for 'straighten' - to rotate the scanned image by a few degrees to correct for the paper being scanned slightly crooked. Different from **autoRotate**.

If VRS is disabled, `deskew` is always disabled. If VRS is enabled, `deskew` is enabled by default but you can disable it with this option.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize(
  scanningOptions: { applyVRS: true, deskew: false }
});
```

disableVRSIfInstalledOnWorkstation

Automatically disable VRS processing by WebCapture, in those cases where VRS is detected on the client workstation. The idea here is that if VRS is detected on the workstation, the user is probably using a VRS-equipped TWAIN driver, so there is no need to apply VRS processing twice to each image.

Default value: false.

Controls the scanning resolution. It would be very unusual to find a scanner that doesn't support 100, 200 and 300 DPI. 150 DPI is almost as widely supported. Nearly all flatbed scanners can scan anything from 50 DPI to 1200 DPI.

discardBlankPages

When this option is true, blank images are detected and discarded during scanning.

i In duplex scanning, front and back sides of pages are discarded independently.

i No ImageAcquired event is fired for such discarded images.

Default value: false.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { duplex: 1, discardBlankPages: true }
});
```

dpi

Controls the scanning resolution. Most of the scanners support 100, 200 and 300 DPI. 150 DPI is almost as widely supported. Nearly all flatbed scanners can scan anything from 50 DPI to 1200 DPI.

Default value: 200 DPI.

To set the Scanner DPI to 300:

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { dpi: 300}
});
```

If you specify "applyVRS: true", the following options are set by default (i.e. if you do not specify them):

- pixelType: 2 (Color)
- resultPixelType: 0 (B&W)
- deskew: true
- autoRotate: true
- discardBlankPages: false

This option is ignored by importFiles.

duplex

Set the Scanner duplex property. The possible values are:

0 = Simplex (front side only)

1 = Duplex (both sides)

-1 = Any (leave up to scanner)

Default value: 0 (Simplex).

All scanners support simplex scanning. Many scanners with an ADF (Automatic Document Feeder) can scan duplex, but many cannot.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { duplex: -1 }
});
```

feeder

This option selects between the ADF (Automatic Document Feeder) and the flatbed/glass or platen. Valid values are: 0 - Scan from platen, 1 - Scan from feeder, -1 - Don't care (up to scanner or user).

Default value: -1 (Don't care)

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { feeder: 0 }
});
```

orientation

This parameter tells the scanner the expected orientation of the paper being fed, in the sense of upright (short edge feed) or sideways/landscape (long edge feed).

-1 = Any (leave up to scanner)

0 = Portrait (paper is scanned 'upright' (short edge feed))

1 = Landscape (paper is scanned 'sideways' (long edge feed))

paperSize

To set the paper size being fed in to the scanner. This is not a complete list, but shows the most common values. The list depends on a particular scanner's capabilities. Values are directly mapped to paper size constants from the TWAIN specification. So if a customer knows his scanner supports some size, but it's not listed here, it could be added to the static list of sizes. As a better choice, the result from `getSupportedValues` could be added. For example 53 3.5" x 2" Business Card

Default value: 3 (8.5" x 11")

Value	Meaning / Dimensions
-1	Indicates 'no preference'
0	TWAIN defines this as meaning 'maximum scan area' but many scanners will treat this as 'default' or 'last size selected by the user.'
1	210mm x 297mm
2	182mm x 257mm (Same as JIS B5)
3	8.5" x 11.0"
4	8.5" x 14.0"
5	148mm x 210mm
6	250mm x 353mm (ISO B4)
7	125mm x 176mm (ISO B6)
8	unused
9	11.0" x 17.0"

Value	Meaning / Dimensions
10	10.5" x 7.25"
11	297mm x 420mm (ISO A3)
12	353mm x 500mm (ISO B3)
13	105mm x 148mm (ISO A6)
14	229mm x 324mm (ISO C4)
15	162mm x 229mm (ISO C5)
16	114mm x 162mm (ISO C6)

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
    handlerUrl: 'TestCaptureHandler.ashx',
    scanningOptions: { paperSize: 3 }
});
```

pixelType

To set the pixel type of the document getting scanned. This affects the pixel type that scanner is asked to produce. To specify the pixel type of the resulting image after all possible image preprocessing, `resultPixelFormat` should be used.

- 0 – Black and white
- 1 – Grayscale
- 2 – RGB 24 bits per pixel
- 3 – Indexed color images 8 bits per pixel
- 1 – Don't care

Default value: 0 (Black & white)

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
    handlerUrl: 'TestCaptureHandler.ashx',
    scanningOptions: { pixelType: 0 }
});
```

Every scanner capable of scanning paper documents can scan in Black & White (B&W) mode. Almost all scanners can scan grayscale and color. Many scanners, but certainly not all, can scan indexed color.

resultPixelFormat

This specifies the pixel format you want delivered to your application after scanning or importing, and post-processing. This is distinct from the `pixelType` parameter, which controls the pixel format requested from the scanner. If the resulting `PixelFormat` is not specified, it defaults to -1.

The pixel format used for scanning is:

1. `pixelType` if specified
2. otherwise the `pixelType` implied by `resultPixelFormat` if specified, (see tables below)
3. otherwise if `applyVRS` is true then Color
4. otherwise: B&W. `applyVRS`:

applyVRS:true Value	Name	Delivered Image	Default Scan
-2	PixelFormat.Auto	B&W and grayscale => BW; All color => BW or RGB24, chosen by VRS	Color
-1(default)	PixelFormat.Any	All => BW (binarized by VRS)	Color
0	PixelFormat.BW	1-bit B&W images	Color
1	PixelFormat.Grayscale	8-bit grayscale images	Grayscale
2	PixelFormat.Color	24-bit color images	Color

applyVRS:false Value	Name	Delivered Image	Default Scan
-2	PixelFormat.Auto	as scanned	BW
-1 (default)	PixelFormat.Any	as scanned	BW
0	PixelFormat.BW Color	1-bit B&W images	BW
1	PixelFormat.Grayscale	8-bit grayscale images	Grayscale
2	PixelFormat.BW Color	24-bit color images	Color

i Note that when VRS is disabled, **resultPixelFormat** can be effectively used in place of **pixelType** to control the scanner.

Example

```
var PixelType = Atalasoft.Controls.Capture.PixelType;
Atalasoft.Controls.Capture.WebScanning.initialize({
    // Deliver color images. Implies scanning color.
    scanningOptions: { resultPixelFormat: PixelType.Color }
});
```

showScannerUI

To show (true), or not show (false) the scanning device's user interface.

Default value: false.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
    handlerUrl: 'TestCaptureHandler.ashx',
    scanningOptions: { showScannerUI: true }
});
```

suppressBackgroundColor

Only has effect in Auto Color mode i.e. when **applyVRS** is true and **resultPixelFormat** is -2.

In that mode, if **suppressBackgroundColor** is true, solid-color background in color scans is treated as white. If there is no other color content on a scanned image, the image will be automatically converted to B&W.

This is useful when your scan batch may include invoices and other documents printed on colored paper, which you want converted to B&W, but you also expect some pages with color content which you want to be preserved as color.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: {
    resultPixelFormat: -2, // detect color & B&W pages automatically
    suppressBackgroundColor: true // treat solid color background as white
  }
});
```

tiff.jpegCompression

Controls use of JPEG compression when writing color and grayscale images in TIFF format.

Important: Uses the revised TIFF 6 form, not Wang/Microsoft variant - check your downstream processes for compatibility.

When true, JPEG compression is used when writing color or grayscale images to TIFF.

When false, some other compression for color and grayscale images in TIFF will automatically be chosen.

Default value: true.

Example

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  handlerUrl: 'TestCaptureHandler.ashx',
  scanningOptions: { tiff: { jpegCompression: true } }
  // Note the nested object-within-object construction
}
```

scanner

Specifies the name of the scanner to use.

Default value: The last scanner selected in the scanner list control. If no scanner has been selected or there is no designated scanner list control, the user's default scanner according to TWAIN is used.

Code Example: scanningOptions.scanner

```
Atalasoft.Controls.Capture.WebScanning.initialize({
  ...
  scanningOptions: { scanner: 'Canon DR-3010C', showScannerUI: true, showProgress:
    false },
  ...
})
```

This option is ignored during importFiles.

showProgress

Similar to showScannerUI, when this option is true, the scanner is asked to display a small progress dialog during scanning. These dialogs typically include a Cancel button. When set to false, the scanner is asked not to display a progress dialog during scanning.

Default value: true.

This option is ignored during importFiles.


threshold

Default value: -1

This scanning option specifies the threshold to be used when scanning to B&W (bitonal) images. The value range is -1 to 255. The threshold value T is applied roughly as follows:

A value of -1 means "let the scanner choose the threshold." Imagine that each pixel of the document is measured as 8-bit grayscale to give a value V, with 0=black and 255=white. In the returned bitonal image, that pixel is returned as white if $V > T$, and as black if $V < T$.

If $V = T$, it may be returned as black or as white, depending on the scanner.

 Almost all scanners support this.

This setting only has an effect when scanning with pixelType 0 (B&W).

barcodes


This scanning option controls bar code recognition during scanning. The results are available in the bar codes property on each image delivered to the onImageAcquired handler.

Code Example: bar code

```
Atalasoft.Controls.Capture.WebScanning.scanningOptions = {
  barcodes: { count: 1, symbology: [ 'Code 39', 'Code 128' ] }
...
function _onImageAcquired(eventName, imageProxy) {
  if (imageProxy.barcodes.length > 0) {
    // process barcodes
    alert('first bar codetype='+imageProxy.barcodes[0].symbology+',
      data='+imageProxy.barcodes[0].data);
  }
}
```

To enable bar code recognition, include bar codes as a property of the scanningOptions object, and set it to an object with one or more of the following properties:

- **property** {type} meaning if omitted
- **count** {integer} maximum number of symbols to recognize -1, meaning "all."
- **checksums** {Boolean} whether to check & strip optional checksums false.
- **symbology** {array} names of symbologies to recognize (see below) "all."

 Symbology names are not case-sensitive, so "qr code" and "QR cOdE" are equivalent.

Supported symbologies:

- EAN-13
- EAN-8
- UPC-A
- UPC-E
- Code 39
- Code 39 (Full ASCII)
- Code 128
- Interleaved 2 of 5

- Codabar
- Code 93
- Aztec
- POSTNET
- PDF417
- Data Matrix
- QR Code
- MicroPDF417
- Micro QR Code


The default bar 1code engine is the Honeywell Omniplanar/SwiftDecoder, which requires a license key, which must be set before using the engine. See Licensing above.

brightness

Default value: 0

This scanning option specifies how the brightness of scanned images should be adjusted by the scanner. Following TWAIN convention, the value range is -1000 to +1000.

- -1000: reduce brightness as much as possible
- $-1000 < n < 0$: reduce brightness
- 0: do not adjust brightness
- $0 < n < 1000$: increase brightness
- +1000: increase brightness as much as possible

 Not all scanners support this.

For some scanners, not all values are distinguished: -500 may have the same effect as -501 or -499.

Some scanners will ignore this value when scanning B&W.

This option is ignored during importFiles.

contrast

Default value: 0

This scanning option specifies how the contrast of scanned images should be adjusted by the scanner. Following TWAIN convention, the value range is -1000 to +1000.

- -1000: reduce contrast as much as possible
- $-1000 < n < 0$: reduce contrast
- 0: do not adjust contrast
- $0 < n < 1000$: increase contrast
- +1000: increase contrast as much as possible

i Not all scanners support this.

For some scanners, not all values are distinguished: -500 may have the same effect as -501 or -499.

Some scanners will ignore this value when scanning B&W.

This option is ignored during importFiles.

deliverables

This scanning option specifies some data formats that the application expects to request, for each scanned or imported image. This provides some performance-enhancing "hints" to Kofax Web Capture. The following code example shows the deliverables option value with all possible members filled in.

Code Example: deliverables

```
Atalasoft.Controls.Capture.WebScanning.scanningOptions = {
  deliverables: {
    localFile: { format: 'tif', jpegCompression: true, quality: 85 },
    thumbnail: { width: 128, height: 128, format: 'tif' },
    originalImageFile: {
      format: 'tif', [jpegCompression: true, quality: 85]
    }
  }
}
...
}
```

The deliverables property if defined must be null or an object as shown above. Neither member of deliverables is required - the object can be empty or contain any combination of localFile, thumbnail or originalImageFile. The additional deliverable data is attached to the image-proxy object that is handed to the ImageAcquired handler, just as bar codes, dpi, pixelType, etc. are properties of the image proxy already. Note that you can consider deliverables as a performance-enhancing 'hint' to the plugin, which simply makes the image-proxy.saveEncryptedLocal and image-proxy.thumbnail functions faster. More details below.

deliverables.localFile

If deliverables.localFile is defined, Kofax Web Capture will automatically save each incoming image to an encrypted local file. The file-id is stored in the *localFile* property of the image-proxy object. The format must be provided. It specifies a filetype (as a 3-letter extension). Exception: When a PDF file is imported this format value is ignored and the PDF file is saved verbatim, in PDF format.

The jpegCompression and quality properties are optional - if present, they are also passed when saving image to encrypted local file.

However, we recommend you not specify these options here, but instead, supply them as top-level scanning options, i.e. as scanopts.tiff.jpegCompression* and scanopts.jpeg.quality*.

deliverables.thumbnail

If deliverables.thumbnail is defined, Kofax Web Capture precomputes thumbnail for each incoming image.

The size of the thumbnail must be specified as width & height.

To obtain thumbnail proxy object call `imageProxy.thumbnail(w,h)`. To retrieve thumbnail base64 data call `thumbProxy.asBase64String(fmt)` and if your width and height match the precomputed thumbnail that data will be returned immediately.

If you call `image-proxy.thumbnail(w,h)` asking for a different size than you specified in `deliverables.thumbnail`, then you'll get a new thumbnail proxy and retrieving image data will require additional scanning service roundtrip.

deliverables .originalImageFile


Tells the service to store the original image obtained from a scanner to an encrypted local file. It's useful when image processing applied to scanned image makes it less readable for humans. For example, color dropout on forms images or even binarization. It makes OCR more robust, but makes human reading harder. With this feature, a client app could preserve both images and use them accordingly.

maxPages

Default value: -1

Requests that the scanner scan no more than the specified number of pages. It defaults to -1, meaning 'no limit'. Set to 1 to scan a single page.

This option is ignored by `importFiles`.

 This means physical pages - if scanning in duplex, `maxPages:1` tells the scanner to send 2 images.

Not all TWAIN scanners can do this! Numerous Kodak models always scan everything in the hopper once you start them.

evrsSettings

Settings can be loaded into EVRS for use in post-processing scanned images, by adding the property `evrsSettings` as a scanning option. The value of this property, if present, must be a string containing a valid XML-style EVRS operation string.

This option has effect only when **applyVRS** is **true**. If **applyVRS** is **false** the **evrsSettings** string is *ignored*. This option can be used to set the effective value of any EVRS parameter, overriding any default value or value normally used by Kofax Web Capture.

Any command/operation included in this string will cause EVRS to ignore any competing scanning options, as described in this table.

eVRS command	ignored options
<code>Do90DegreeRotation</code>	<code>autoRotate</code>
<code>DoBarcodeDetection</code>	<code>patchCodes</code>
<code>DoBinarization</code>	<code>resultPixelFormat</code>
<code>DoBlankPageDetection</code>	<code>discardBlankPages</code>
<code>DoColorDetection</code>	<code>resultPixelFormat</code>
<code>DoCropCorrection</code>	<code>autoCrop</code>
<code>DoEnhancedBinarization</code>	<code>resultPixelFormat</code>
<code>DoGrayOutput</code>	<code>resultPixelFormat</code>

DoSkewCorrectionAlt	deskew
DoSkewCorrectionPage	deskew

Code Example: scanningOptions.evrsSettings

```
Atalasoftware.Controls.Capture.WebScanning.initialize({
...
scanningOptions: { evrsSettings:
'_Do90DegreeRotation_2' +
'_DoGrayOutput_' +
'_LoadSetting_<PropertyName="CBinarize.Do_Adv_Clarify.Bool" Value="1" Comment="DEFAULT
0"/>'
},
...
}
```

jpeg.quality

Default value: 75

Specifies the 'quality' to use when doing JPEG compression, either in saved JPEG files, or in TIFF files saved with JPEG compression.

Lower quality = smaller files. Higher quality = larger files.

Valid quality values are from 1 to 100. We recommend using values of 60 and above, unless file size is at a premium. Quality is severely degraded around 50 and below. Around 85-90 and above, images begin to be indistinguishable from uncompressed images except under magnification.

The effect of the quality setting depends on the content, the medium and the human viewer. Quality should be selected using a representative sample of documents, displayed or printed and viewed in realistic end-use scenarios, preferably by several different people.

Remember that B&W images are unaffected by this setting, it applies only to grayscale and color images saved to files using JPEG compression.

Upload Options

The following are the supported properties of the **uploadOptions** object that can be included in the parameters to `Atalasoftware.Controls.Capture.WebScanning.initialize`.

Keep in mind that an upload is submitted to the server as a POST of a MIME message, per RFC2046 "Multipart Media Type." The uploaded file is included as a form-data part with name="file".

uploadOptions.formData object

Default value: null

Each 'own' property of the formData object is inserted in uploads as a form-data part, with its name and value. On the server, if you override the **HandleUpload** method, each property of the formData object will be available in the context.Request.Form dictionary. Numbers are converted to strings, strings are sent unquoted. Objects and arrays are sent (as strings) in JSON format. Unicode characters are handled correctly in both names and values.

Code Example: uploadOptions.formData

```
his._uploadOptions = {};
...
```

```


Atalsoft.Controls.Capture.WebScanning.initialize({
  ...
  uploadOptions: this._uploadOptions,
  ...
})
...
this._uploadOptions.formData = {
  application: 'Doc-U-droid 9000',
  operator: ''
  pages: [1, 3, 7]
};
// an upload initiated now will include three extra form-data parts.
// These fields are accessible on the server
// in the HandleUpload method, viz:
string app = context.Request.Form["application"];
string oper = context.Request.Form["operator"];
string pages = context.Request.Form["pages"];

```

uploadOptions.extraParts array of strings

Default value: undefined

This optional parameter is a list of additional parts to be inserted in the multipart body of each upload. Each string must conform to the rules in RFC2046 for a *body-part*. This option is retrieved from the uploadOptions object at the beginning of each upload, so its contents *at that moment* are included in the subsequent upload.

 The order of strings in the array is not necessarily preserved in the upload.

Code Example: uploadOptions.extraParts

```

this._uploadOptions = {};
...
Atalsoft.Controls.Capture.WebScanning.initialize({
  ...
  uploadOptions: this._uploadOptions,
  ...
})
...
this._uploadOptions.extraParts = [
  'Content-Disposition: form-data; name="application"\n'+
  '\n'+
  'Doc-U-droid 9000'
];
// an upload initiated now will include the 'application' field
// above, in the multipart MIME message sent to the server.

```

Connect to the Web Document Viewer

To display the documents scanned with the Web Capture WebScanning Control in the same page, or possibly another page in a browser simply do the following:

1. Add the Web Document Viewer resources to the application.
2. Add the following div tag to the same page:

Code Snippet

```
<div>
  <div class="atala-document-toolbar" style="width: 670px;"></div>
  <div class="atala-document-container" style="width: 670px; height: 500px;"></div>
</div>
```

3. In the html/asp/jsp page add the following script. The values of serverurl and handlerUrl should be changed to the locations of your WebDocViewer and Capture handlers, respectively.

Code Snippet

```
<script type="text/javascript">
  // Show status and error messages
  function appendStatus(msg)
  {
    $('#status').append('<p>'+msg+'</p>');
  }

  // Initialize Web Scanning and Web Viewing
  $(function() {
    try {
      var viewer = new Atalasoftware.Controls.WebDocumentViewer({
        parent: $('.atala-document-container'),
        toolbarparent: $('.atala-document-toolbar'),
        serverurl: 'WebDocViewer.ashx'
      });

      Atalasoftware.Controls.Capture.WebScanning.initialize({
        handlerUrl: 'TestCaptureHandler.ashx',

        onScanError: function(msg, params) { appendStatus(msg); },

        onScanStarted: function(eventName, eventObj) { appendStatus('Scan
        Started'); },

        onScanCompleted: function(eventName, eventObj)
        { appendStatus('Scan Completed: ' + eventObj.success); },

        onUploadError: function(msg, params) { appendStatus(msg); },
        onUploadStarted: function(eventName, eventObj)
        { appendStatus('Upload Started'); },
        onUploadCompleted: function(eventName, eventObj) {
          appendStatus('Upload Completed: ' + eventObj.success);
          if (eventObj.success) {
            viewer.OpenUrl('atala-capture-upload/' +
            eventObj.documentFilename);
          }
        },
        scanningOptions: { pixelType: 0 }
      });

      Atalasoftware.Controls.Capture.CaptureService.initialize({
        handlerUrl: 'TestCaptureHandler.ashx',
        onError: function(msg, params) { appendStatus(msg + ': ' +
        params.statusText); }
      });
    }
    catch (error) {
      appendStatus('Thrown error: ' + error.description);
    }
  });
</script>
```

```
</script>
```

Licensing

Kofax Web Capture requires confirmation that it is licensed, and that its built-in copy of VRS is also licensed if you wish to use it. Kofax Web Capture and bar code engine licensing are integrated into a single object.

Normally this licensing information will come from the server. Kofax Web Capture can accept it in two different ways that might be called 'push' licensing, and 'call-back' licensing.

For details, see the `params.license` property under `Atlassoft.Controls.Capture.WebScanning.initialize`.

File Formats and File Options

Several methods take a file format as a parameter, followed by an optional options parameter.

Examples are `encryptedLocalFileAsBase64String` and `imageProxy.asBase64String`.


Valid values for the format parameter are described in this table:

Value	Result
"bmp" or ".bmp"	Windows BMP file
"gif" or ".gif"	standard GIF file
"jpg" or ".jpg"	standard JPEG (technically, JFIF)
"png" or ".png"	standard PNG file
"tif" or ".tif" TIFF file	TIFF file Bitonal images are written to TIFF with CCITT G4 compression. Color and grayscale are written uncompressed, unless the <code>tiff.jpegCompression</code> option is true.

Use VirtualReScan (VRS)

The Web Capture Web Scanning Control includes the award-winning VirtualReScan Technology (VRS), a "de Facto Standard" for image processing. VRS requires additional licensing.

By default, VRS processing is applied to each scanned image: All images are auto-rotated and deskewed, and non-B&W images are converted to B&W ('binarized').

 The specific image processing steps performed by VRS may change in future versions of Web Capture.

To turn on or off VRS processing in the client an optional argument must be passed into the `Atalasoftware.Controls.Capture.WebScanning.initialize` method on the page in which the control has been added. See Client API Reference.

Example - Disabling VRS JavaScript

```
Atalasoftware.Controls.Capture.WebScanning.initialize({
    handlerUrl: 'TestCaptureHandler.ashx',

    onScanError: function(msg, params) { appendStatus(msg); },
    onScanStarted: function(eventName, eventObj) { appendStatus('Scan
Started'); },
    onScanCompleted: function(eventName, eventObj) { appendStatus('Scan
Completed: ' + eventObj.success); },

    onUploadError: function(msg, params) { appendStatus(msg); },
    onUploadStarted: function(eventName, eventObj) { appendStatus('Upload
Started'); },
    onUploadCompleted: function(eventName, eventObj) {
        appendStatus('Upload Completed: ' + eventObj.success);
        if (eventObj.success) {
            viewer.OpenUrl('atala-capture-upload/' + eventObj.documentFilename);
        }
    },
    scanningOptions: { pixelType: 1, applyVRS: false }
});

Atalasoftware.Controls.Capture.CaptureService.initialize({
    handlerUrl: 'TestCaptureHandler.ashx',
    onError: function(msg, params) { appendStatus(msg + ': ' + params.statusText); }
});
```

In the `scanningOptions` argument, `applyVRS` is set to `false` to turn VRS off in the Web Scanning Control.

Test your application

Obviously you should test scanning, ideally with several scanners. Yes, we try to hide all the scanning issues and make it "just work". Nonetheless, it can be beneficial to learn about the problems your end-users will have setting up and using scanners, to get a sense of the little idiosyncrasies every scanner has, and to understand the physical details of the task you are asking users to carry out.

Test in Edge, Firefox and Chrome

Always test with all the browsers you intend to support. The Web Capture plugin may operate differently and require subtle differences in coding in different browsers.

Test for error conditions

Scanning and uploading documents is fast and simple when everything works. Your users' efficiency and satisfaction will primarily depend on how you handle errors and failures.

Verify that your application behaves in a reasonable way and guides users successfully when:

1. These browsers are not supported: Safari, or Opera, or a non-Windows OS.
2. The client PC has no devices in the TWAIN device list.
3. The selected scanner is turned off or disconnected.
4. A scan is canceled mid-scan.
5. Web Capture throws an error, especially those documented in Handling Errors.

Troubleshoot scanning problems

My scanner appears in the list as WIA-something - what does this mean?

Many scanners support Microsoft's proprietary scanner protocol, called WIA. Microsoft Windows enables WIA devices to appear as TWAIN devices. However, these pseudo-TWAIN devices are not native TWAIN drivers, and sometimes have important limitations. If you have any problems using a WIA driver through TWAIN, see if the scanner vendor offers a native TWAIN driver.

My scanner appears twice in the scanner list, once with a WIA-prefix and once without - what does this mean?

This means your scanner supports the Microsoft WIA scanner protocol as well as having a native TWAIN driver. Basically you are seeing two different drivers that can both talk to your scanner. In general we recommend using the native (non-WIA) driver, but you are welcome to try them both and see which one works better for you.

Scanner does not appear in device list.

Things to check:

- Is the scanner connected and powered on.
- Does the scanner support TWAIN? The popular Fujitsu ScanSnap models do not.
- Is a TWAIN driver for the scanner installed? Most do not auto-install.
- Test the driver+scanner combination outside the browser with IrfanView, see "Scanning fails" below.

Scan fails with "unable to open" or "connection failure"

Scan fails before scanning any pages

- Is the scanner connected and powered on?
- If there has been a recent crash or error related to scanning?
Try cycling the power on the scanner and then re-try the scan, up to two times.
- Verify that the scanner is working outside the browser, through TWAIN. Note the TWAIN name of the device.

To verify that a scanner has a working TWAIN driver, we sometimes use IrfanView - this is a free scanning application with TWAIN support.

- If IrfanView can scan from your scanner, then you have a working scanner with a working TWAIN driver.

In this case, scanning failures are most likely caused by the browser sandboxing the scanner driver. Try moving your Web site into the trusted zone.

- If IrfanView cannot find and scan from your scanner, then you don't have the basic prerequisite of a working TWAIN scanner.

The ultimate fall-back for this kind of problem is to get support from the scanner vendor.

Uploads fail with '598' status

If uploads fail with '598' status codes, this indicates the client-side code timed-out waiting for the upload to complete. You can increase the `Params.serverTimeout`: Integer value, try to speed up your connection, reduce your upload sizes (see below), or (if it's actually the problem) speed up your server.

Uploads fail with JSON parse error or server response status 404

If uploading fails with a JSON parse error or server response 404, once you check that the URL being used for upload is valid and correct, consider whether the upload might be exceeding the server's upload size limit. Some suggestions:

- Increase the server's upload limit.
- Scan to grayscale instead of color, or better yet to B&W. If you use VRS, we recommend specifying `resultPixelFormat` instead of `pixelType`. This allows VRS to scan in color and use sophisticated algorithms to convert to grayscale or B&W. Note that this may cause slower scanning with some scanners.
- If you have licensed VRS, try setting `resultPixelFormat`: -2 to tell VRS to automatically classify color and non-color images, and convert non-color images to B&W. If you are scanning pages on colored paper, take a look also at `suppressBackgroundColor`.
- If you are scanning with a DPI higher than 200, experiment with scanning at 200 DPI to see if the results are acceptable.

Documents do not display in viewer after scan & upload

First check the upload: Are the documents being uploaded?

1. Attach handlers for `onUploadStarted`, `onUploadCompleted` and `onUploadError` (see Client API Reference) - is `onUploadCompleted` being fired, and not `onUploadError`?
2. Does each new upload appear in the upload folder on the server? (See notes about the upload folder in Getting Started with Web Capture).

Second, check the viewer:

1. Is the code to invoke the viewer being called, and is the correct URL being given to the viewer? An alert box is an easy way to check this.
2. Can you enter the URL or its fully-qualified equivalent into a browser manually and get the expected file?

Scan quality is poor

If images or graphics looks bad, could this be because the scans are being converted to black & white? See question below.

Are you setting the resolution? See Setting Scanning Options. A very low resolution - anything below 100 DPI or so - will produce blurry or ... well, 'low-res' images.

As a very rough guide, for black & white scans of text:

100 DPI	legible but visibly rough & pixelated, like a poor quality fax or a 1970's video game.
---------	--

150 DPI	modest fax quality, still some visible defects and pixelation but highly legible in typical on-screen viewing.
200 DPI	high quality fax. A full page viewed on screen looks good, letters look slightly fuzzy or 'haloed'.
300 DPI	good quality, minor defects usually visible only under magnification.

All scans are converted to B&W, even my bunny pictures

This is the default behavior for WebCapture: It applies several VRS clean-up operations to each scan, including binarization, which converts the image to 1-bit per pixel black & white.

To avoid this, specify the `resultPixelFormat` as grayscale (1) or color (2). If you are using VRS, you can also specify `resultPixelFormat: -2` which tells VRS to automatically classify each image as color or non-color, and to convert only non-color images to B&W.

I ask for duplex scanning, but only front sides are scanned.

It sounds a little silly, but the first thing to check is that 'duplex scanning' is a listed feature of the scanner.

Assuming the scanner claims to support duplex (both sides) scanning, the most common reason for it to fail is using the scanner through a WIA driver (choosing WIA-something in the scanner list). The WIA drivers have historically had problems with duplex scanning.

If a WIA-driver is being used, the solution is to find, install and use a native TWAIN driver for the scanner.

Logging

There are client and server side logging results.

Server logging can be enabled in configuration files located in the installation folders: `Kofax.WebCapture.Host.exe.log.config` and `Kofax.WebCapture.ScanWorker.exe.log.config`. By default logging is turned off.

Client side logging can be enabled in the browser console by setting `Atalasoftware.TraceEnabled = true;`

Uninstall Web Capture MSI

Here are the steps to uninstall the scanning service on a Windows client machine.

1. Open **Add or Remove Programs**.
2. Locate and right click the Kofax WebCapture item.
3. Select **Uninstall**.

Client API reference

[Atalasoftware.Controls.Capture.WebScanning](#)

This object is responsible for communicating with the client-side scanner, controlling scanning, and uploading scanned documents to the Web server.

Atalsoft.Controls.Capture.WebScanning.initialize(params)

This method must be called to initialize the WebScanning component. The params object must contain a handlerUrl property, the other properties are optional.

As a side-effect, initialize attempts to initialize TWAIN scanning on the client, and to collect a list of available TWAIN scanners. If it is successful, it will populate a scanner-list control and enable a scan button, provided that they exist with the appropriate classes. See [Connect to UI controls](#).

i The scanner initialization process is asynchronous and may not have finished when the initialize function returns. In fact it may never complete, for example if the user declines to install and activate the plugin MSI.

params.handlerUrl: string

The URL of the request handler. This is normally a relative URL, for example: TestCaptureHandler.

params.serverTimeout: Integer

This is the number of seconds to wait for the server response after starting an upload. After this number of seconds, the upload is considered to have failed, it is canceled, and an error is signaled by calling onUploadError.

Default value: 20

params.onScanClientReady: function()

This handler is called when scanning initialization is complete. Not successful, just complete: Scanning initialization has either succeeded fully, or failed. Normally in case of failure the onScanError handler will have been called.

See [Handle errors](#)

params.onScanError: function(msg, params)

This handler is called when an error occurs during scanning or scan initialization.

params.onScanStarted: function(eventName, eventObj)

This handler is called when a scan is started. See [Handle events](#)

params.onScanCompleted: function(eventName, eventObj)

This handler is called when a scan is completed.

params.onUploadStarted: function(eventName, eventObj)

This handler is called when a document upload is starting.

params.onUploadCompleted: function(eventName, eventObj)

This handler is called a document upload has completed.

params.onUploadError: function(msg, params)

This handler is called when an error occurs during uploading.

params.scanningOptions: object

This object contains any scanner settings to be used for scanning, as described in Setting Scanning Options.

params.license:object

Provides Kofax Web Capture, VRS, and bar code engine licensing information.

key a Kofax Web Capture and VRS license authentication string. Required if object set.

barcode a bar code engine license key. Optional.

Currently the only bar code engine key accepted is for the Honeywell/SwiftDecoder engine.


Code Example: licensing object

```
Atalasoftware.Controls.Capture.WebScanning.initialize({
...
license: { key: '~}qs~z}af4 @# ?)%?!+F#*(#(&^H4f`gw4f`gw', barcode: 12345 },
...
})
```

Default value: If no license object is provided to `WebScanning.initialize`, Kofax Web Capture asks the server for license information. Specifically, the initialize method starts an AJAX GET to the `handlerUrl` with `?cmd=getlicense` appended. For licensing to be successful, GET must return a JSON encoded string that decodes to a licensing object as described above. Typically the `WebCaptureRequestHandler` on the server takes care of this automatically

params.onShutdownfunction()

This optional handler is called when the page is unloaded or reloaded, prior to any internal Kofax Web Capture cleanup or shutdown.

 Any scan or import that is in progress is aborted before this handler is called.

If `WebScanning.Initialize` did not succeed, it is possible this handler will not be called.

And if you are somehow using Kofax Web Capture in the Opera browser, this handler will never be called because Opera famously does not support thepage-unload event.

params.onImageAcquiredfunction(eventName, imageProxy)

See `onImageAcquired` handler.

params.uploadOptionsobject

This property must be an object, which contains (additional) upload settings, as described in *Upload Options*.

params.localizationobject

Default value: `{}` - which selects the default *American English* localization. This object provides a translation table for all potentially user-visible strings used by Kofax Web Capture.

Identifier	Approximate English
activex	The Scanning ActiveX Control needs to be installed, updated, or enabled, or ActiveX controls need to be allowed to install and run.
ajax	Could not create an XMLHttpRequest object needed for uploading.
badBrowser	Scanning requires Chrome, Firefox, or Edge running on Windows, or Safari running on macOS
badLicense	license is invalid or expired
badVrsLicense	VRS license is missing or invalid
contentTypeError	The server could not retrieve content types.
contentDescError	The server could not retrieve the content description.

Identifier	Approximate English
docClassIndexFieldError	The server could not retrieve the index fields for the document class.
doorOpen	scanner reports cover or door open
doubleFeed	scanner reports a double feed
driverCrash	a fatal exception occurred in the device driver
dsmFail	TWAIN DataSource Manager failure
dsOpen	unable to open TWAIN device
fileFail	file not found or cannot be written or cannot be read
ieVersion	IE version not supported:
importError	The server could not import the document.
importFilesPrompt	Choose Files to Import' (title of file-open dialog of importFiles)
internalError	unexpected internal error
licensingError	The server did not return valid licensing information
minVersion	minimum version needed:
not32Bit	Browser is not 32-bit:
noPlugin	The scanning plugin needs to be installed. If it is installed it may need to be enabled. In your browser look for Add-ons, Plugins, Kofax scanning.
notSupported	Not currently supported:
notWindows	Platform is not Windows:
noTwain	The TWAIN Manager needs to be installed.
oldPlugin	The scanning plugin needs to be updated to a newer version.
outOfMemory	memory allocation failed (out of memory)
paperJam	scanner reports a paper jam
pluginCreate	plugin not created
pluginVersion	plugin is version:
scanFail	Unable to start scan
scanMore	Would you like to scan more pages? (user prompt during multipage scan from flatbed)
serverNotResponding	The server is not responding.

Atalasoft.Controls.Capture.WebScanning.scan(options)

Initiates a scan with the specified scanning options (see [Set scanning options](#)).

If you pass in nothing, null or undefined, it uses the scanning options stored in `Atalasoft.Controls.Capture.WebScanning.scanningOptions`.

This method is called (with no parameters) when the user clicks the designated scan button.

Atalasoft.Controls.Capture.WebScanning.abortScan()

Aborts the current background operation in progress, if any. If there is no current background operation, it does nothing.

Background Operation Started by:

- Scanning WebScanning.scan
- Importing files WebScanning.importFiles
- Querying supported values WebScanning.getSupportedValues
- Displaying the scanner settings dialog WebScanning.showSettingsDialog

Atalasoft.Controls.Capture.WebScanning.importFiles([options])

Begin a background process to import files with the specified options. The *options* object has the same valid properties as the *scanningOptions* parameter to the *scan* method above, however, this method ignores options that control the scanner. If *options* is omitted, **null** or **undefined**, importFiles uses the scanning options stored in **Atalasoft.Controls.Capture.WebScanning.scanningOptions**.

The user is prompted to select one or more files on the local machine, with a standard multi-select File Open dialog. The supported file formats are those listed in File Formats and File Options above, plus PDF. If the user cancels the File Open dialog, this is treated as an import of zero (0) files.

The title of the dialog is a localizable string named `importFilesPrompt`.

The selected files are read image-by-image and processed as if they were being scanned - post-processing options are applied to each image - except for PDF files, which are passed through verbatim.

Files are processed in an order determined by Windows, and not necessarily in the order they appear in the File Open dialog nor the order of selection. If the order of processing is important, the user must do separate Import operations.

importFiles calls the onScanStarted handler before doing anything else, then calls the onImageAcquired handler with each successfully imported image or PDF file, and finally calls the onScanCompleted handler when finished, whether successful or not.

The *eventObj* parameter to onScanCompleted has a property (success) that tells you if the import completed successfully. See the onImageAcquired handler for details of how PDF files are imported.

If the user attempts to import any files of unsupported type (such as .doc or .psd) the unsupportedFileFormat error is fired to the onScanError handler and the import proceeds, completely ignoring all those files.

If a file of supported type cannot be imported (e.g. corrupted data, access error), an appropriate error is fired to the onScanError handler and the import process is aborted (completes unsuccessfully).

The images and files imported by importFiles are not retained by the control or uploaded automatically to the server.

This method is called (with no parameters) when the user clicks the designated import button, if any.

Atalasoft.Controls.Capture.WebScanning.dispose: function (success, error)

Closes scanning session, unbinds events, removes default generated UI, if any.

Atalasoft.Controls.Capture.WebScanning.isInitialized: function()

Indicates whether WebScanning component is initialized.

Atalasoft.Controls.Capture.WebScanning.getVersion():function(){}

Returns the current Kofax Web Capture version.

Atalasoft.Controls.Capture.WebScanning.LocalFile.splitToFiles: function (fid, [params], callback)

Splits specified file to the set of blobs of the specific size(source stream is not decoded to image, so chunks are just blobs, not a valid images)

```
params: { size: number, removeSource: bool }
```

Callback is called with array of new files identifiers in order of split. this function is useful for transferring huge files that could not be mapped into memory in a 32 bit process.(32-bit browser could refuse to load files bigger than 200-300mb).

Atalasoft.Controls.Capture.WebScanning.setBarcodeLicense(licenseKey)

Code Example: bar code licensing:

```
Atalasoft.Controls.Capture.WebScanning.setBarcodeLicense(licenseKey)
```

The licenseKey is a string, which is decoded/decrypted and passed to the Honeywell engine.

Atalasoft.Controls.Capture.WebScanning.getSupportedValues(params,callback)

Begins a background operation to query the scanner for supported values of scanning parameters. When done, it invokes *callback(vals)* where *vals* is an object describing the supported values, described below. If anything goes wrong, it invokes *callback({})*. Before querying the scanner, the current scanning options are set as specified by *params*. If *params* is **null**, the current scanning options are used.

The scanner used is: *params.scanner*, if that exists and is a string. Otherwise

Atalasoft.Controls.Capture.WebScanning.scanningOptions.scanner is used if it exists and is a string. Otherwise the scanner most recently selected in the UI or otherwise the default scanner, as reported by TWAIN.


Errors during this operation will normally result in an asynchronous call to the onScanError handler.

The supported-values object

Each property in the object has the name of a Kofax Web Capture scanning-parameter property e.g. *pixelType*, *dpi*, and so on. Each property value is one of the following:

- A enumeration, represented by an array of valid values, for example [0, 1, 3].
- A range represented by an object with *min*, *max* and *step* properties, for example { *min*: 50, *max*: 2400, *step*: 1 }.

In this release of Kofax Web Capture, a range value is returned only for the value of *dpi*, and only for scanners that describe their resolution values this way.

 A few models of scanner may provide incorrect or misleading information through this query, such as a flatbed scanner that lists the value 1 for **feeder**, implying that you can scan from its (non-existent) ADF.

i Qquerying the scanner's capabilities requires opening the scanner, which may fail (scanner offline, unplugged, etc.) which may display an error box to the user. For TWAIN-friendly user-interface design, make 'choose your scanner' or 'change scanner' into a separate dialog or screen, and call `getSupportedValues` when the user OK's a new scanner choice.

i Completing the `getSupportedValues` operation can take several seconds.

Code Example: `getSupportedValues`:

```
Atalasoft.Controls.Capture.WebScanning.getSupportedValues(null, gotValues);  
// if successful might call gotValues with an object like this:  
{ pixelType: [0, 1, 2],  
  dpi: {min: 50, max: 1200, step: 1},  
  duplex: [0, 1],  
  feeder: [0, 1],  
  paperSize: [0, 1, 2, 4, 7, 9],  
  orientation: [0 1]  
}
```

Atalasoft.Controls.Capture.WebScanning.getSupportedValues(null , gotValues);

Starts a background operation to display the settings-only dialog (custom user interface) of the scanner, if the scanner supports this feature. This shows a version of the scanner's UI that is only for choosing settings, without a Scan button.

When the operation completes, successfully or not, `callback(status)` is called, where `status` is an object.

`status.complete` is true if the dialog was successfully displayed and closed by the user, false if the dialog could not be displayed or the operation was aborted.

If anything goes wrong, the `onScanError` handler will be called, asynchronously, with details.

The scanner used is: `params.scanner` if that exists and is a string. Otherwise

`Atalasoft.Controls.Capture.WebScanning.scanningOptions.scanner`, if it exists and is a string. Otherwise the scanner most recently selected in the UI, or otherwise the default scanner, as reported by TWAIN.

Atalasoft.Controls.Capture.WebScanning.LocalFile.setEncryptionKey: function (key, callback)

Sets the encryption key for subsequent encrypted local file load/saves. The basis is an arbitrary string, which is used to generate the symmetric encryption key. The basis is immediately discarded, and the generated key is moved into the Windows secure cryptographic storage.

Required for any local file operations.

Atalasoft.Controls.Capture.WebScanning.LocalFile.fromBase64String: function (str, callback)

Saves the specified base64-encoded binary data to an encrypted local file, and returns a unique file-identifier for that local file.

`str`: base64 string representing data to store.

Atalasoft.Controls.Capture.WebScanning.LocalFile.asBase64String: function (fid [,fmt [, options[, callback]])

Decrypts a locally-saved file and returns it as a base64-encoded string. If `fmt` is specified, returns the data in the specified file format. If no encryption key has been set, throws an exception.

Atalasoftware.Controls.Capture.WebScanning.LocalFile.remove: function (fid, callback)

Immediately deletes a local file and any data associated with it. This is irreversible and non-recoverable.

Atalasoftware.Controls.Capture.WebScanning.LocalFile.list: function (callback)

Returns an array of the identifiers of all the existing saved local files associated with this instance of Kofax Web Capture.

Atalasoftware.Controls.Capture.WebScanning.LocalFile.removeAll: function (callback)

Immediately deletes all local files associated with (created by) this instance of Kofax Web Capture. Has no effect on local files created in other instances of the client application, or files created by previous incarnations of Kofax Web Capture.

Atalasoftware.Controls.Capture.WebScanning.LocalFile.globalPurgeByAge: function (hours, callback)

Immediately deletes all local files written more than hours ago. Deletes ALL local files meeting the age criterion, no matter how or when they were created, or by which instance of Kofax Web Capture.

Encrypted Local Files

- Local files are stored in this folder: `...\Users\\AppData\Local\Kofax\WebCapture\Persistent`
- The file names follow this pattern: `('T'|'U')<sessionid>-<filenumber>'.elf'`
- T indicates a 'trusted' file - Kofax Web Capture generated the contents.
- U indicates an untrusted file - contents came from outside the plugin e.g. via `LocalFile.fromBase64String`.
- The session-id is generated each time the plugin is instantiated, and should be unique on any given computer for ~13 years.
- The file number is generated sequentially within each session, starting from 1.
- `.elf` stands for Encrypted Local File.

Atalasoftware.Controls.Capture.WebScanning.scanningOptions

This property holds the current scanning options as described in Setting Scanning Options. These options are used when the user clicks the scan button, or if `Atalasoftware.Controls.Capture.WebScanning.scan()` is called.

Initially this holds the `scanningOptions` object passed to `Atalasoftware.Controls.Capture.WebScanning.initialize(params)`, but your code can dynamically edit this object to change the settings for a subsequent scan.

Atalasoftware.Controls.Capture.WebScanning.getProfile()

This method returns a string, containing the current scanning options. The string is in JSON format, but you should not rely on that.

Atalasoftware.Controls.Capture.WebScanning.setProfile(s)

This method loads the scanning options from a string previously produced by `Atalasoftware.Controls.Capture.WebScanning.getProfile()`.

 This is not a merge - all scanning options not set in the strings are cleared.

Atalasoftware.Controls.Capture.CaptureService

This object is responsible for returning information from KIC, such as the content-types and content-type document descriptions, and for importing uploaded documents into Kofax Capture.

.initialize(params)

This method must be called to initialize the CaptureService component. The params object must contain a handlerUrl, the other items are optional.

As a side-effect, initialize starts a process that attempts to communicate with the KIC service and obtain the content-type list and content-type document description list. If this process succeeds, it will populate the appropriate controls on the page, if they exist with the correct classes.

- **params.handlerUrl: string:** The URL, normally relative, of the KIC extended Web request handler on the server.
Example: 'TestCaptureHandler.ashx'
- **params.onError: function(msg, params):** This event is called if KIC returns an error. See [Handle errors](#).
- **params.onImportCompleted: function(params):** This event is called when a document has finished importing. See [Handle events](#).
- **params.onBeforeImport: function(boolean):** This parameter must return a function that returns true or false. The function passed in through this parameter will be run prior to importing a document to KC.
- **params.onTrackStatusReceived: function(params):** This event is called when track status of a document imported into KC is requested.
- **params.contentType: String:** Use this parameter when the client is not bound to any UI elements in a page to set the batch class or repository name for KIC. See [Connect controls with no UI](#).
- **params.defaultContentType: String:** Use this parameter to specify one content type to be the content type first displayed and selected when the list is populated.
- **params.onContentTypesCreated: function(params):** The event is fired after the content type selection box has finished being populated.
- **params.contentTypeDescriptionName: String:** Use this parameter when the client is not bound to any UI elements in a page to set the KIC document class/form type pair.
- **params.onContentTypeDescriptionsCreated: function(params):** The event is fired after the content type description selection box has been populated.
- **params.removedContentTypes: String:** Use this parameter to filter the list of content types displayed in the content type selection drop down. The list (comma separated) specified in the parameter will be removed from the list. See [Filter selection lists](#).
- **params.removedContentTypeDescriptions: String:** Use this parameter to filter the list of content type descriptions displayed in the content type document list selection drop down. The list (comma separated) specified in the parameter will be removed from the list.
- **params.LoosePages: String:** This parameter enables importing loose pages in to KC through KIC only. Set to "true" to enable. Default is false. To specify a loose pages selection text in the atala-contenttype-document-list selection set the parameter to "true, <any text>", if unspecified a blank entry is created. See [Import loose pages](#).

- **params.displayedLoosePagesForContentType: String:** Use this parameter in conjunction with the LoosePages parameter. Will only apply to clients connecting to a KIC service. This take a comma separated list of batch class names which should have the loose pages option included in the content type description name drop down list.
- **params.indexFields: String:** Use this parameter when the client is not bound to the indexfield UI div to set the indexfields for the specified KC/KIC document class.
- **params.displayedIndexFields: String:** Use this parameter when the client is bound to the index field dive element to inclusively filter the list of index fields displayed.
- **params.onIndexFieldImportValidationError: function(params):** Use this parameter to deal with index field validation errors.
- **params.onIndexFieldTypeValidationStatus: function(params):** Use this parameter to customize how index field validation gets dealt with in the client.
- **params.onIndexFieldCompleted: function(params):** Use this parameter to customize event behavior when an index field label, and input field have been created.
- **params.onIndexFieldsCompleted: function(params):** Use this paramter to customize event behavior for after all index field labels and input fields have been created.
- **params.batchFields: String:** Use this parameter when the client is not bound to the indexfield UI div to set the batch fields for the specified KC/KIC batch class.
- **params.displayedBatchFields: String:** Use this parameter when the client is bound to the index field dive element to inclusively filter the list of batch fields displayed.
- **params.onBatchFieldImportValidationError: function(params):** Use this parameter to deal with batch field validation errors.
- **params.onBatchFieldTypeValidationStatus: function(params):** Use this parameter to customize how batch field validation gets dealt with in the client.
- **params.onBatchFieldCompleted: function(params):** Use this parameter to customize behavior after each batch field label, and input field have been created.
- **params.onBatchFieldsCompleted: function(params):** Use this parameter to customize behavior after all of the batch fields labels and input fields have been created.

.setIndexFieldValues(String)

This function can be used to set index fields outside of the initialization parameters. Currently this works as it would if used in the initialization parameters, and is intended to be used when no UI for index fields is present.

Example

```
var indexFields = "Required: filled in";
Atalasoftware.Controls.Capture.CaptureService.setIndexFieldValues(indexFields);
```

.setBatchFieldValues(String)

This function can be used to set batch fields outside of the initialization parameters. Currently this works as it would if used in the initialization parameters, and is intended to be used when no UI for batch fields is present.

Example

```
var batchFields = "BatchField1: SomeText";
Atalasoftware.Controls.Capture.CaptureService.setBatchFieldValues(batchFields);
```


Chapter 5

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 Kofax Web Capture.

The Web Document Viewer online documentation is available at <https://atalasoft.github.io/web-document-viewer>. The offline version can be downloaded from the public GitHub repository at <https://github.com/Atalasoftware/web-document-viewer/tree/master/docs>.

Chapter 6

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 be viewed at arbitrary magnification with little or no loss of information.

The Atalasoftware 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 Atalasoftware 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 Atalasoftware 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 `Concat`s 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 Atalasoftware 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 Atalasoftware 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 (such as 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	Δ	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	Υ
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	ο
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	€	ı	161	Ÿ
Ç	162	'	£	163	≤
¤	164	/	¥	165	∞
ı	166	f	§	167	♣
¨	168	◆	©	169	♥
ª	170	♠	«	171	↔
¬	172	←	•	173	↑
®	174	→	–	175	↓
º	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	⌘	Á	193	Ɔ
Â	194	⌘	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	Σ
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
đ	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 Kofax Web Capture 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 in the process is replaceable if need be.

In addition, there is an extra assembly for interacting with Kofax Web Capture that contains an `AtalaImageCompressor` object for handling all `AtalaImage` types.

See [Integrate with Kofax Web Capture](#) 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 Kofax Web Capture, 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 Atalasoftware 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 implements at least the `PdfRenderable` interface and optionally the `PdfRenderableContainer` and `PdfResourceConsumer` interfaces. All shapes must be serializable.

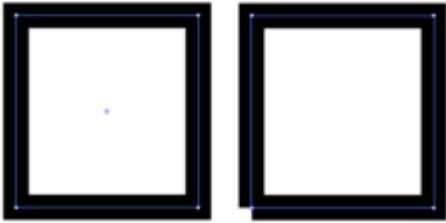
PdfPath

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 connect 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.



i 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:

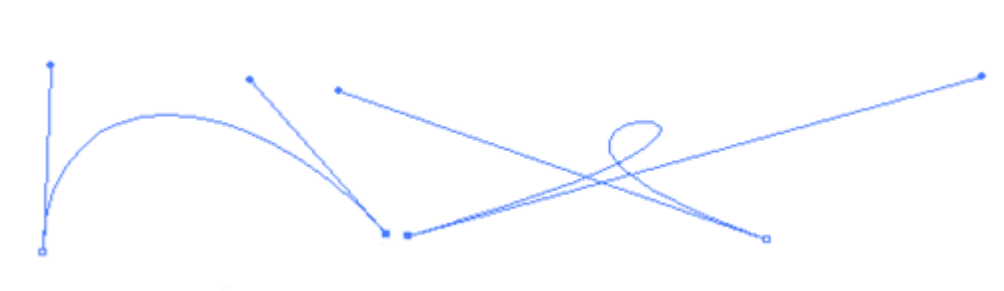
```
return path.MoveTo(0, 0).LineTo(wide, 0).LineTo(wide, wide).LineTo(0,
wide).Close();
```

Curves in PDF are represented by cubic Bézier functions. A Bézier is represented by four points, a start point and an end point (P₀ and P₃) and two control points (P₁ and P₂) and is represented by the following formula:

$$B(t) = (1 - t)^3 P_0 + 3t(1 - t)^2 P_1 + 3t^2(1 - t) P_2 + t^3 P_3$$

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 of 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.

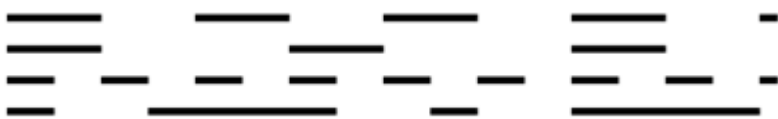
The thickness of a line is in PDF units and defaults to 1.0. When set to 0, the PDF viewer is instructed to render the line in the thinnest possible way. Since this is device-dependent, the final output will not be consistent from device to device and this should be avoided (consider the difference between 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]

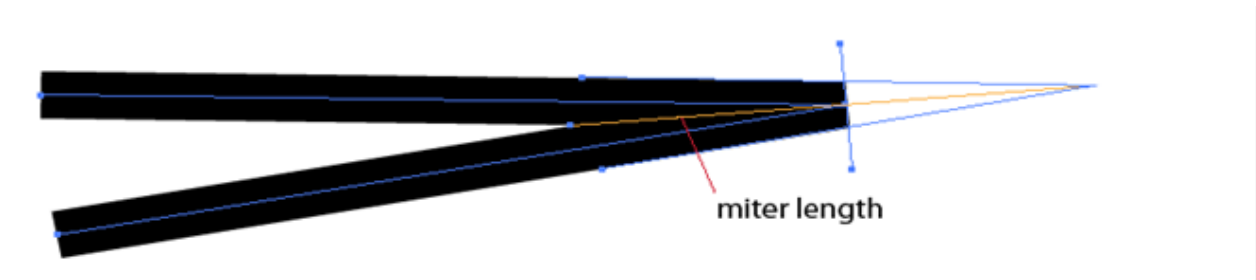


In the sample dash patterns, a single entry implies alternating dash and gaps of the same length. In the bottom example, you can see how complex dashes patterns can be made. Each pattern above has a phase of 0, meaning that the PDF viewer starts the pattern at the beginning. If the phase were 0.5, the 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:

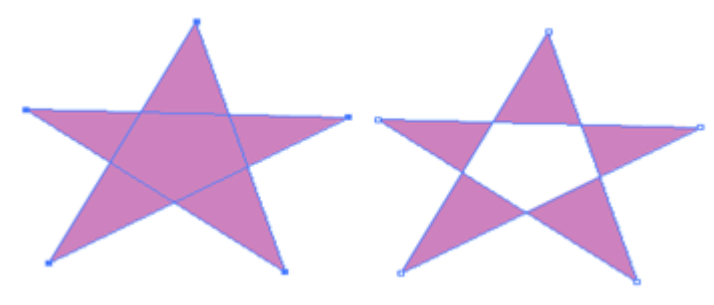
$$\frac{\text{miter length}}{\text{line width}} = \frac{1}{\sin\left(\frac{\theta}{2}\right)}$$

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 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 `Atalasoftware.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();
```

 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 coordinates 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.

This is a PdfTextLine.

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.

PdfClippedTextLine will clip the

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.

**PdfTextBox will
format text to fit
in the box.**

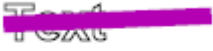






PdfStyledTextBox is similar to PdfTextBox except that it accepts a StyleTextInput object which can be used to add new styled text to the box. Typically this will be used for font changes or color changes.

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.

FillText	
StrokeText	
FillThenStrokeText	
Invisible	
FillTextAndClip	
StrokeTextAndClip	
FillStrokeTextAndClip	
Clip	

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");
}

```

The code produces the following output.

<i>Person</i>	<i>Age</i>	<i>Favorite Color</i>
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)));

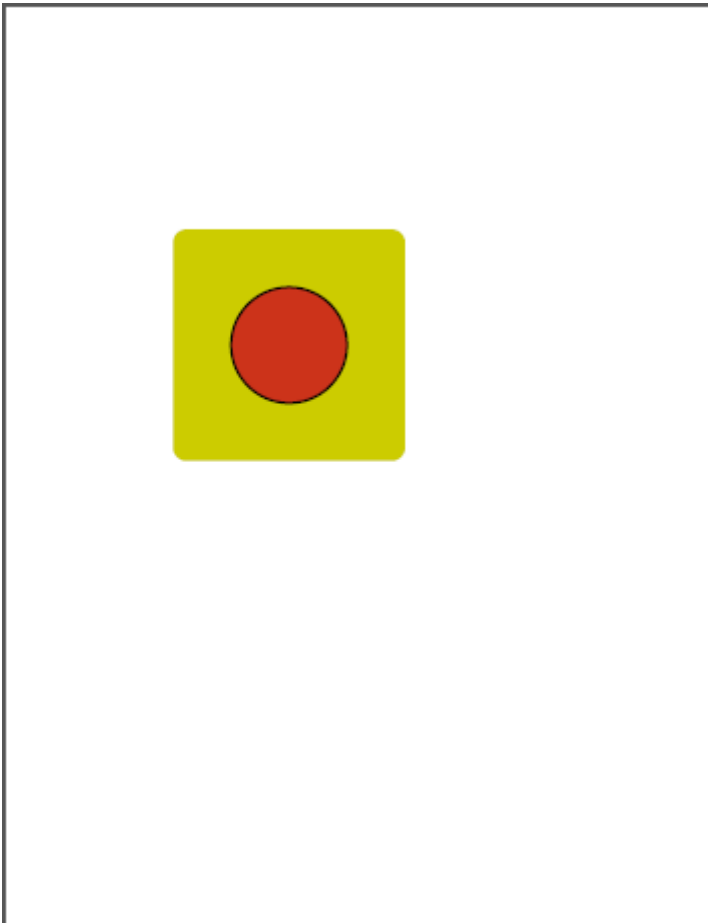
string resourceName = doc.Resources.Templates.Add(template);

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:



i 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. 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);

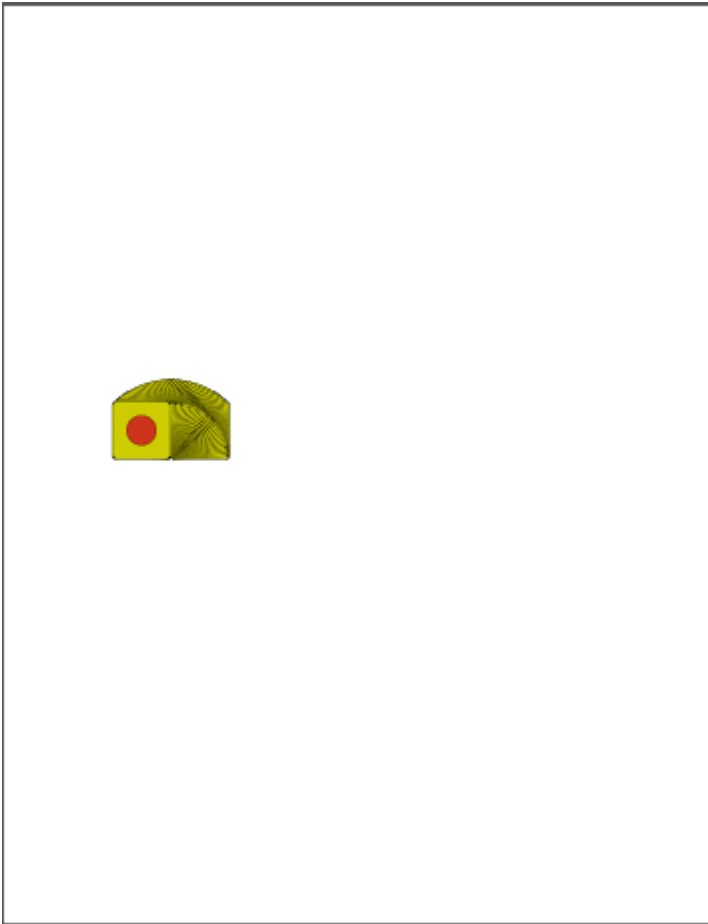
    DrawingTemplate template = new DrawingTemplate(new PdfBounds(0, 0, 204, 204));
    template.DrawingList.Add(new PdfRoundedRectangle(new PdfBounds(2, 2,
    template.Bounds.Width - 4, template.Bounds.Height - 4),
    12, PdfColorFactory.FromRgb(.8, .8, 0), PdfColorFactory.FromRgb(0, 0, 0), 4));
    template.DrawingList.Add(new 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 Postnet bar code accepts a text string with either 5, 9, or 11 digits. The bar code is 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 given 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 it 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, colorspace, 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 Atalsoft 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 Atalsoft PDF Generating toolkit also includes the ability to import pages from the Atalsoft `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 Web Capture

In addition to the main assembly, there is an additional assembly, Atalasoftware.PdfDoc.Bridge. This assembly provides a bridge between Kofax Web Capture 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 Atalasoftware.Jpeg2000Encoder 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 Jpeg2000Encoder(),
    null);

string imName = doc.Resources.Images.AddImage(atalaImage);
string imName1 = doc1.Resources.Images.AddImage(atalaImage);
```

In this example, doc1 is created with the Atalasoftware.Jpeg2000Encoder 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 `PdfAuthoredDocument.Save()`, 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 PdfDestination 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.

i 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("Atalasoftware", Color.Blue,
    FontStyle.Regular,
    new PdfURIAction(new Uri("http://www.atalasoftware.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", selfPrint);
```

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.

i 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

 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

- CaretAnnotation
- 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.

i 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.

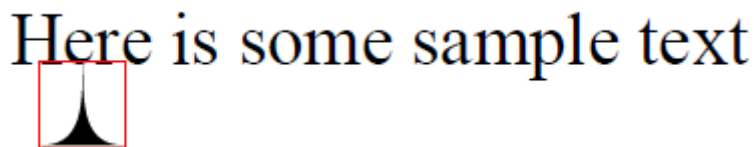
Property name	Property type	Description
InsetArea	PdfBounds	A rectangle that specifies margins around the caret symbol. The rectangle needs to be fully contained within the Bounds rectangle.

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

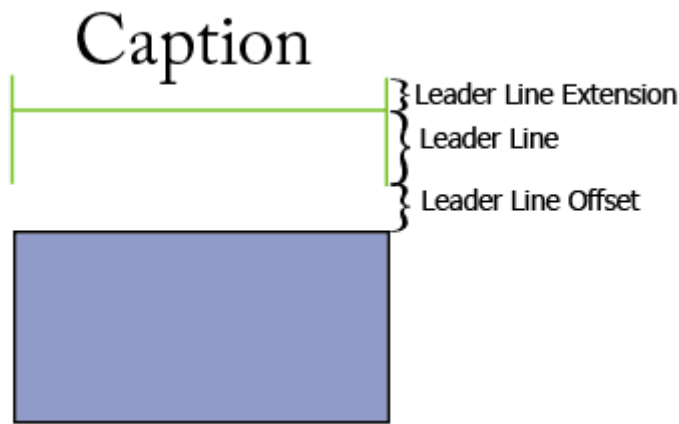
An EllipseAnnotation is identical to a RectangleAnnotation except that it is rendered as an ellipse that fits within the Bounds property.

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. They 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 its 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
<code>AutoGenerateBasicAppearance</code>	bool	If set to true, the annotation will autogenerate a simple appearance upon being rendered. If the <code>Regions</code> collection is empty, it will generate a single rectangle outlined with the annotation's <code>Color</code> . If the <code>Regions</code> collection is not empty, it will generate a single <code>PdfPath</code> with each quadrilateral outlined in the annotation's <code>Color</code> .
<code>DefaultTextAppearance</code>	<code>PdfTextAppearance</code>	This property, if set, will represent how the <code>OverlayText</code> will appear on the annotation. If not set, the text, if any, will appear in Helvetica 12 point.
<code>IsOverlayTextRepeated</code>	bool	If set to true, the <code>OverlayText</code> string will be repeated over the surface of the redacted area when the redaction is applied.
<code>OverlayText</code>	string	Gets or sets text that will be rendered on the redaction area after the redaction has been applied.
<code>RedactionInteriorColor</code>	<code>IPdfRgbColor</code>	An RGB color that will be used to render the interior area of the redaction after it has been applied. If <code>RedactionTemplate</code> is set, this will be ignored.
<code>RedactionTemplate</code>	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

i 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 convert 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 PopupAnnotation 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.

i 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");
```

i 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 <code>Bounds</code> 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 markups 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
<code>AutoGenerateBounds</code>	bool	If set to true (default), the annotation will use the <code>StartPt</code> property, the <code>Contents</code> property, and the font information to calculate the <code>Bounds</code> property at render time. Lines will be split at "\r" or "\n" characters.
<code>AutoGenerateInsetArea</code>	bool	If set to true and if <code>AutoGenerateBounds</code> is true, then the <code>InsetArea</code> will be calculated as if it were the bounds and the <code>Bounds</code> will be calculated by expanding the <code>InsetArea</code> by the margins.
<code>LeftRightMargin</code>	double	If <code>AutoGenerateBounds</code> is true, this value will be used to create margins on the left and right edges. Must be non-negative.
<code>StartPoint</code>	<code>PdfPoint</code>	If <code>AutoGenerateBounds</code> 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.
<code>TopBottomMargin</code>	double	If <code>AutoGenerateBounds</code> is true, this value will be used to create margins on the top and bottom edges. Must be non-negative.

i 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` and `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";
```

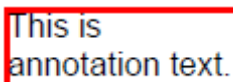
```

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.

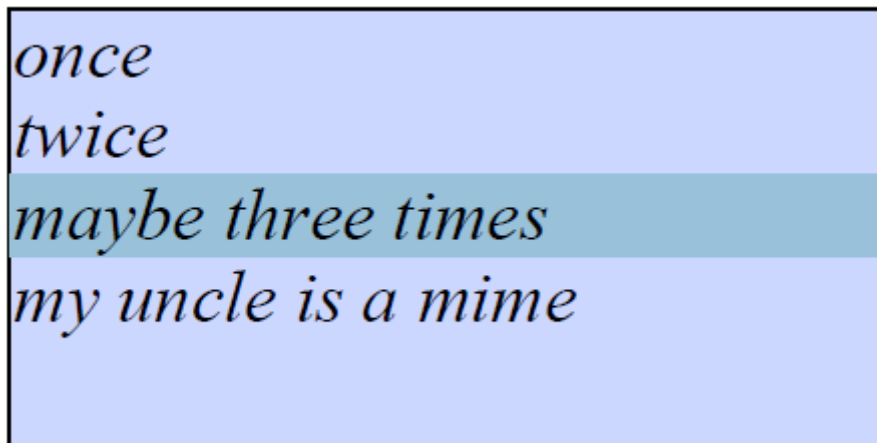
Text of items in the list will be rendered using the DefaultTextAppearance property.

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.

i 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, "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");

```

i **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("threechoiceready.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 to 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
<code>AutoGenerateBasicAppearance</code>	bool	If set to true, before rendering the widget will generate a basic appearance for the text box.
<code>DefaultRichTextStyleString</code>	string	Gets or sets a default rich text string to be used to define the style of the <code>RichTextValue</code> of the widget. Note that if you use <code>RichTextValue</code> , you need to also set the <code>Vlue</code> property to a plain text version of the rich text.
<code>DefaultTextValue</code>	string	Gets or sets the default value for the widget.
<code>IsColumns</code>	bool	If set to true, the <code>MaxLength</code> property will be used to define columnar layout of the text. Note that <code>IsColumns</code> only makes sense if <code>IsPassword</code> , <code>IsScrollable</code> , and <code>IsFileSelection</code> are all false.
<code>IsFileSelection</code>	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");
```

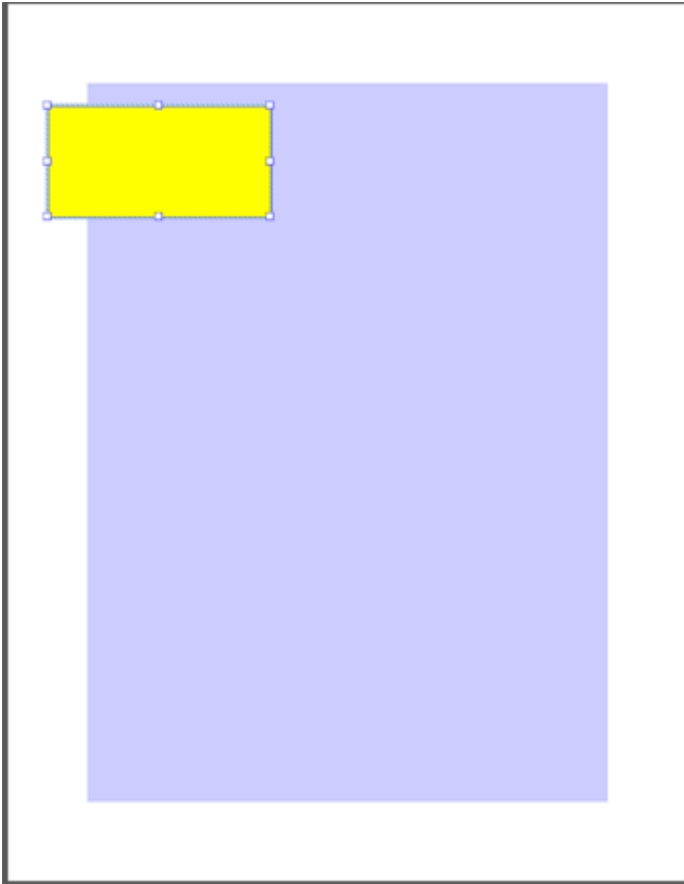
Use annotations

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

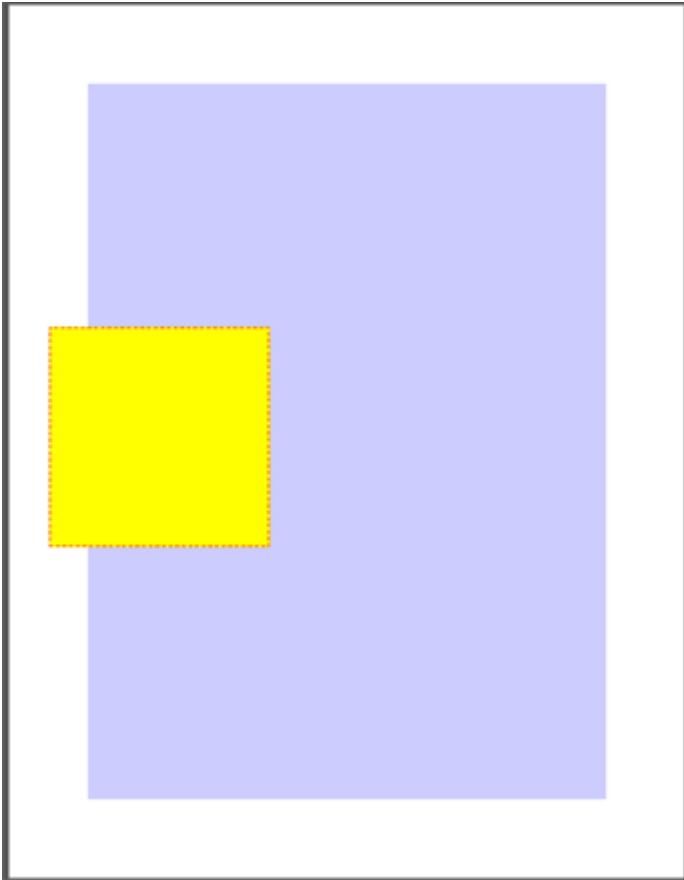
This C# sample creates a page with a light blue rectangle and a yellow rectangle annotation with an orange dashed border.

```
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)));

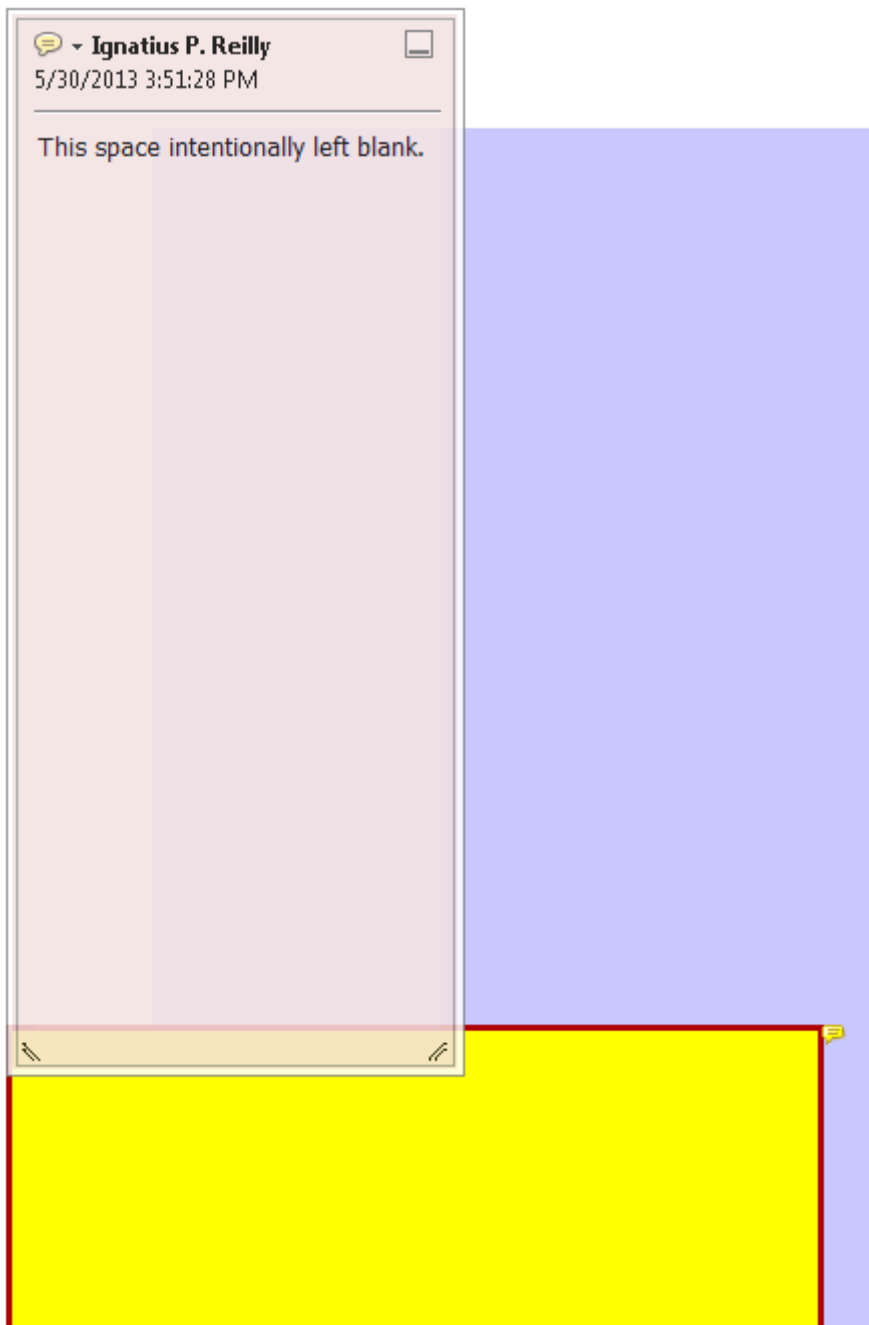
RectangleAnnotation rectAnnot = new RectangleAnnotation(new PdfBounds(36, 300, 200,
200));
rectAnnot.InternalColor = PdfColorFactory.FromRgb(1, 1, 0);
rectAnnot.Color = PdfColorFactory.FromRgb(1, .5, 0);
rectAnnot.Border = new AnnotationBorder(0, 0, 1.5, new double[] { 4, 1 });
page.Annotations.Add(rectAnnot);

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

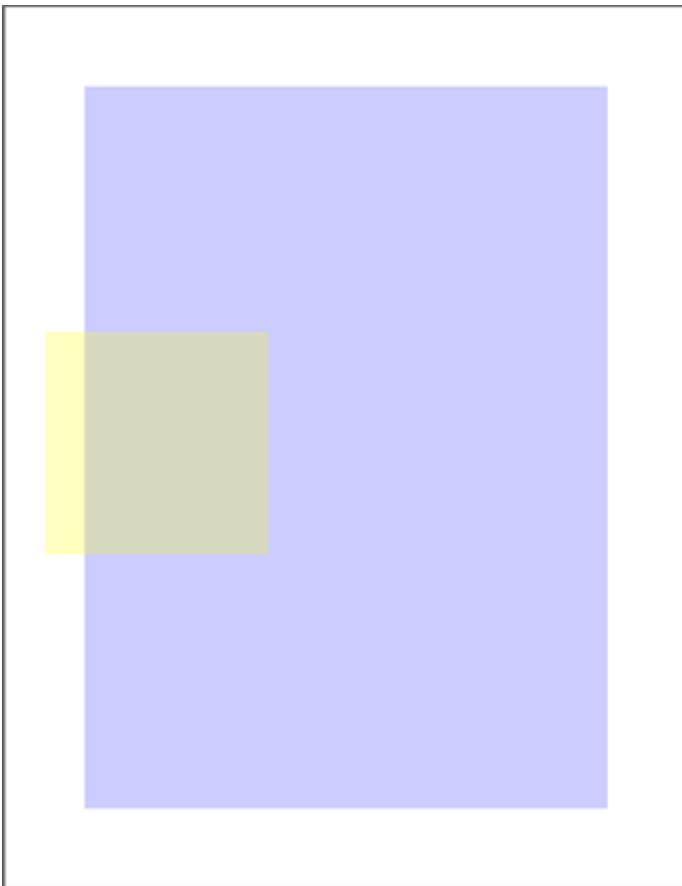
This C# sample shows how to set transparency in a rectangle annotation.

```
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

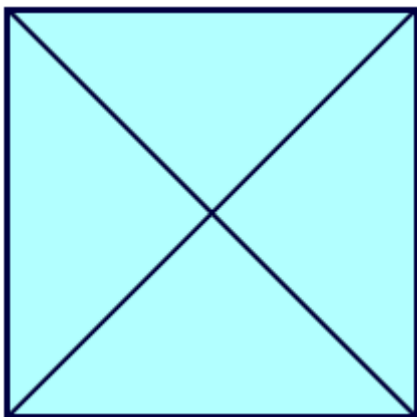
This C# sample demonstrates how to create an annotation with a custom "Normal" appearance. For simple skinning, you should create exactly one appearance and put it in the Normal collection under the name `AppearanceSet.DefaultAppearanceName`. This creates a rectangle with an x.

```
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.

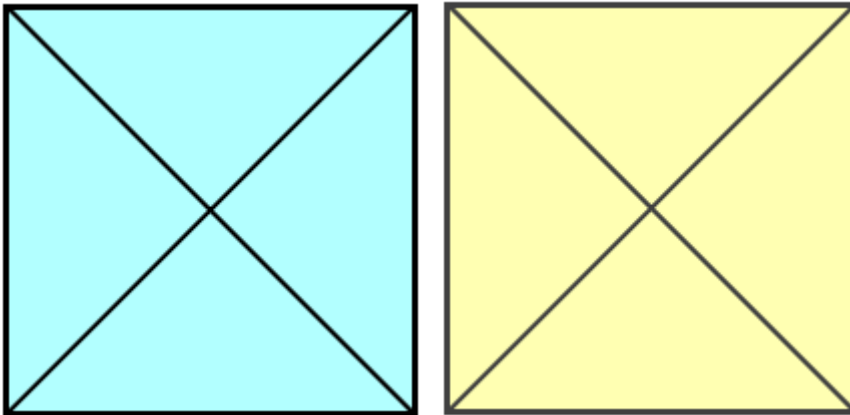
```
public string MakeAppearance(PdfBounds bounds, IPdfColor outline, IPdfColor fill,
    GlobalResources resources)
{
    DrawingTemplate template = new DrawingTemplate(bounds);
    bounds = bounds.Expand(-1);
    template.DrawingList.Add(new PdfRectangle(bounds, outline, 1, fill));
    PdfPath path = 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);

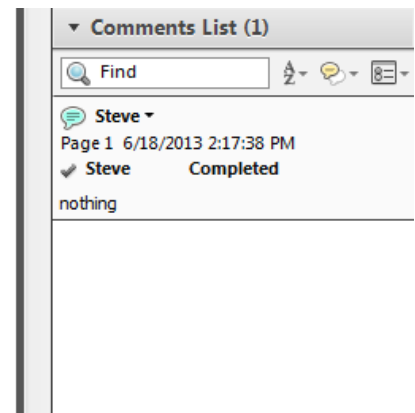
StickyNoteAnnotation sticky = new StickyNoteAnnotation(new PdfBounds(144, 600, 72, 72),
    "Completed set by steve hawley", new PdfBounds(156, 420, 100, 100));
sticky.Color = PdfColorFactory.FromRgb(1, 1, .8);
sticky.IconName = StickyNoteAnnotation.CommentIconName;
GeneralReview generalReview = new 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");

```



Steve - (1 Status)
nothing



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");
```

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.

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");
```


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.

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");
```

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.

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");
```

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.

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.

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");
```

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 incidunt 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.

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 Bowdler was here, scing vestibulum. Bowdler was here, suere at venenatis dapibus suscipit ing 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.

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 or 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);

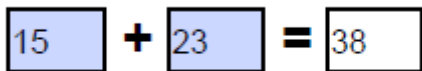
PdfTextLine tl = new PdfTextLine("Helvetica-Bold", 20, "+", new PdfPoint(114, 506));
page.DrawingList.Add(tl);

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");
```

 $15 + 23 = 38$

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");
```

Jane	Smith
First Name	Last Name



Jane Smith

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 as AcroForms. A PDF Form is a hierarchical collection of fields that represent the form data as well as some information to indicate 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`.

i 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. PdfForm 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 objects 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 loop 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");
```

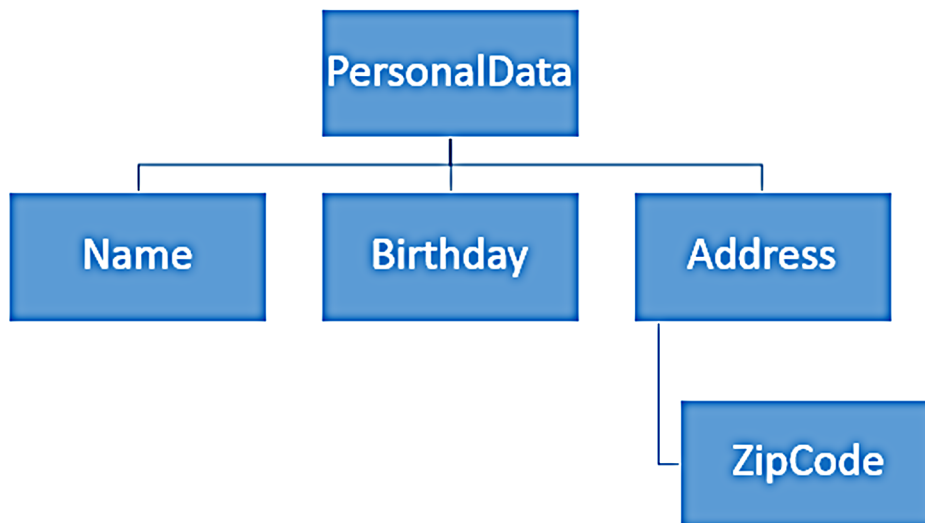
i In this example, the reset button was added to the page, but not to the form. This will prevent it from being subjected to the reset. This could also have been accomplished by putting the button in the form and adding it to the Fields property of the PdfResetFormAction. Since PushButtonFields and PushButtonWidgetAnnotation 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 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 ProblemSelector and 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;
};
```

 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
<code>RebuildCrossReferenceTable</code>	<code>bool</code>	<code>false</code>	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 <code>RepairedStoredStream</code> , if non-null will be set to the <code>StoredStream</code> 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 <code>PdfParseException</code> .
<code>StoredStreamProvider</code>	<code>IStoredStreamProvider</code>	<code>TempFileStreamProvider</code>	This object is used to create a <code>StoredStream</code> object that is used to hold the contents of a PDF file that had to have its cross-reference table rebuilt.
<code>RepairedStoredStream</code>	<code>StoredStream</code>	<code>null</code>	After repairing a cross reference table, this property will be non-null and will contain the <code>Stream</code> 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.
CreateBlankPageIfNoPagesFound	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.
CorrectInvalidDataStreamLengths	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.
DuplicateNameTreeEntryRepairAction	DuplicateNameTreeEntryRepairAction	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, by 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://127.0.0.1`. 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 `X509Certificate2` 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 <ul style="list-style-type: none"> • 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 PdfDocumentSignatureInformation 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	<p>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.</p>
PagesOfAnnotations	ReadOnlyCollection <IList<BaseAnnotation>>	<p>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.</p> <p>If document forbids adding or removing fields or annotations, each sub-collection will also be read-only.</p> <p>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.</p> <div data-bbox="1079 1302 1451 1738" style="background-color: #e6f2ff; padding: 10px; border: 1px solid #add8e6;"> <p>i 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).</p> </div>

Property name	Property type	Description
Form	PdfForm	<p>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).</p> <p>If a document forbids adding or removing fields or widget annotations, each collection of child fields will be read-only.</p> <div style="background-color: #e6f2ff; padding: 5px; border: 1px solid #add8e6;"> <p>i When adding or removing a widget annotation from the document via PagesOfAnnotations, it is imperative that the parallel change be made in Form.</p> </div>

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 viewer 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 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 digest method to the CmsInformation constructor, it will use SHA256 by default.

 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 Atalsoft 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 in 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

i 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.

i 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
}
```

i 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 Atalasoftware.dll and uses PdfDecoder from Atalasoftware.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.

PdfActionException	PDF/A does not support AdditionalActions in widget annotations.	Clear AdditionalActions property in widget annotations or set IgnoreUnsupportedAnnotsAndActions = true.
PdfActionException	PDF/A does not support ClickActions in widget annotations.	Clear ClickActions-property in widget annotations or set IgnoreUnsupportedAnnotsAndActions = true.
PdfActionException	PDF/A does not support AdditionalActions in form fields.	Clear AdditionalActions-property in form fields or set IgnoreUnsupportedAnnotsAndActions = true.
PdfAnnotationException	PDF/A does not support [annotation type] annotations.	Set IgnoreUnsupportedAnnotsAndActions = true.
PdfAnnotationException	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.
PdfFontException	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.
PdfFontException	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.
PdfImageAndColorSpaceException	PDF/A standard does not support images with a mask. Use ConvertIncompatiblePagesToImages to convert the page to an image.	Set ConvertIncompatiblePagesToImages = true.
PdfImageAndColorSpaceException	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.
PdfImageAndColorSpaceException	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.

PdfImageAndColorSpaceException	Page contains unimported images that may contain prohibited parameters for PDF/A-1b.	Set ConvertIncompatiblePagesToImages = true.
PdfImageAndColorSpaceException	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 PdfRenderer.ImageExtractor property.
PdfImageAndColorSpaceException	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 PdfRenderer.ImageExtractor property.
PdfImageAndColorSpaceException	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 PdfRenderer.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 = ObjectCannotBeSavedEventArgs.SaveObjectAction.Skip;
                break;
            case "SoundAnnot":
                obj.Action =
                ObjectCannotBeSavedEventArgs.SaveObjectAction.ThrowException;
                break;
        }
    };
    doc.Save("pdf2_0.pdf", options, null);
}
```

Chapter 7

BarcodeReader

The Kofax Web Capture BarcodeReader add-on provides advanced bar code image recognition for your .NET applications. This component is very easy to use and designed specifically for Microsoft .NET.

 Licensing is runtime royalty free for desktop applications.

Features

- Recognizes all bar codes in an image
- Returns the string value of each bar code recognized
- 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 bar codes
- Returns the coordinates of the start and end lines, can be used to construct a polygon encompassing the bar code area
- Detects the type of bar code recognized
- Integrates with Kofax Web Capture 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 bar codes
- Deploys as a single managed assembly alongside Kofax Web Capture 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
PDF417

2D Barcodes

QR Code

Deployment

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

Assembly	Description
Atalasoftware.Shared.dll	Shared classes such as licensing management
Atalasoftware.dotImage.Lib.dll	Web Capture low level library assembly
Atalasoftware.dotImage.dll	Web CaptureKofax Web Capture class library assembly
Atalasoftware.dotImage.Barcoding.Reading.dll	Barcode Recognition Engine

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 bar codes located within an image.

The following examples demonstrate how to read bar codes from an Atalasoftware.Imaging.AtalaImage object.

The steps involved in reading a bar code are 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 code read from the image:

Reading a bar code

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 bar codes
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 bar codes 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 bar codes
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 bar codes read. Maybe the image was scanned upside down. Recheck by
        scanning the opposite direction.
        options.Direction = Directions.West;
        bars = br.ReadBars(options);
    }
}
```