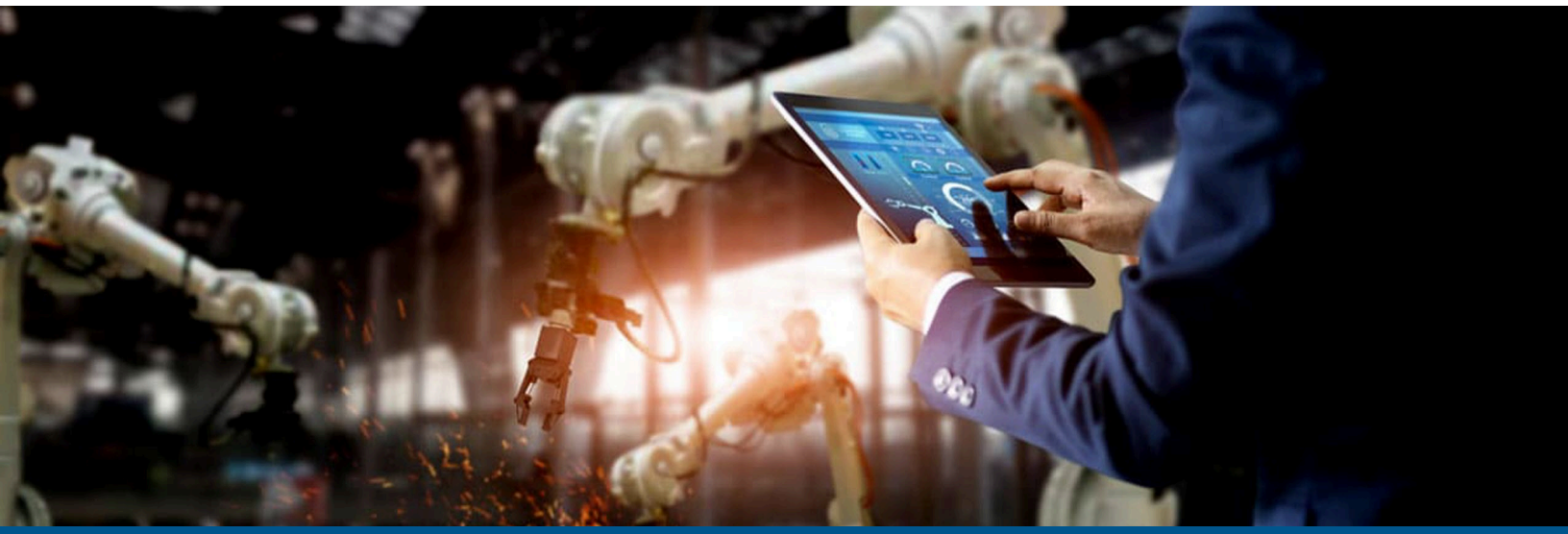

Kofax RPA
開発者ガイド - レガシー API

バージョン: 11.5.0

日付: 2023-10-02

KOFAX

目次

はじめに

ロボットはAPI (Java または .Net) を介して RoboServer で実行されます。独自のアプリケーションで API を直接使用することも、Management Console を使って ロボットを使用して間接的に API を使用することもできます。

> このガイドには、現在レガシー API と呼ばれている非推奨の API が含まれています。

このガイドは 3 つの部分で構成されています:

- 「Java プログラマー ガイド」(Java プログラムで使用されるレガシー API に関する説明が記載されています。)
- 「NET プログラマー ガイド」(C# プログラムを含む .NET アプリケーションで使用されるレガシー API に関する説明が記載されています。)

Java および .NET API リファレンス ドキュメントは、オフライン ドキュメント フォルダにあります。Java API ドキュメントは、オンライン ドキュメント サイトでも入手できます。詳細については、『Kofax RPA インストール ガイド』を参照してください。

関連ドキュメント

Kofax RPA のドキュメント セットには次の場所からアクセスできます。[1]

https://docshield.kofax.com/Portal/Products/RPA/11.5.0-nlfihq5gwr/RPA.htm

このガイドの他に、ドキュメント セットには次の項目が含まれています。

Kofax RPA リリース ノート
その他の Kofax RPA ドキュメントからは入手できない最新の詳細やその他の情報が含まれています。

Kofax RPA 技術仕様
サポートされるオペレーティング システムおよびその他のシステム要件に関する情報が含まれています。

Kofax RPA インストール ガイド
Kofax RPA およびそのコンポーネントを開発環境にインストールする方法について説明します。

[1] オンラインのドキュメント セットにアクセスするにはインターネットに接続する必要があります。インターネットに接続せずにアクセスする方法については、『インストール ガイド』を参照してください。

Kofax RPA アップグレード ガイド
Kofax RPA やそのコンポーネントを新しいバージョンにアップグレードする手順が含まれています。

Kofax RPA 管理者ガイド
Kofax RPA での管理タスクについて説明します。

Kofax RPA のヘルプ
Kofax RPA の使用方法について説明しています。ヘルプは、『Kofax RPA ユーザー ガイド』という PDF 形式のドキュメントとしても提供されています。

Kofax RPA ロボット ライフサイクル マネジメントのベスト プラクティス ガイド
Kofax RPA 環境でロボット ライフサイクル マネジメントを使用しながらパフォーマンスを最適化し、成功を確実にするために推奨される方法とテクニックを提供します。

Kofax RPA ロボット構築の開始ガイド
Kofax RPA を使用してロボットを構築するプロセスを実行するためのチュートリアルを提供します。

Kofax RPA Document Transformation スタート ガイド
OCR、抽出、フィールドの書式設定、検証などを含む Kofax RPA 環境の Document Transformation 機能を使用する方法について説明します。

Kofax RPA Desktop Automation サービス設定ガイド
リモート コンピュータで Desktop Automation を使用するために必要な Desktop Automation サービスを設定する方法について説明します。

Kofax RPA Integration API documentation (Kofax RPA 統合 ***API*** 文書***)***
Kofax RPA へのプログラムでのアクセスを提供する Kofax RPA Java API および Kofax RPA .NET API についての情報が含まれています。Java API 文書は、オンラインおよびオフラインの Kofax RPA 文書から入手できますが、.NET API 文書はオフラインのみとなります。

Kofax RPA API は、元の製品名である「RoboSuite」に対する詳細な参照を含んでいます。RoboSuite の名前は下位互換性を確保するために残されています。API ドキュメントの中では、RoboSuite という用語は Kofax RPA と同じ意味で使われています。

トレーニング

Kofax は、Kofax RPA ソリューションを最大限に活用するために、教室でのトレーニングとコンピュータでのトレーニングを提供しています。利用可能なトレーニング オプションとスケジュールの詳細については、https://learn.kofax.com/ の Kofax 教育ポータルを参照してください。

また、https://smarthub.kofax.com/ の Kofax Intelligent Automation SmartHub にアクセスして、追加のソリューション、ロボット、コネクタなどを見つけることもできます。

Kofax 製品のヘルプの入手

[Kofax Knowledge Portal (Kofax ナレッジ ポータル)] リポジトリにある記事の内容は定期的に更新され、Kofax 製品の最新情報について参照できます。製品に関してご不明の点がある場合は、Knowledge Portal (ナレッジ ポータル) で情報を検索することをお勧めします。

[Kofax Knowledge Portal] にアクセスするには、https://knowledge.kofax.com にアクセスしてください。

[Kofax Knowledge Portal] は Google Chrome、Mozilla Firefox、または Microsoft Edge 向けに最適化されています。

[Kofax Knowledge Portal] は以下の内容を提供します。

- 強力な検索機能で必要な情報をすぐに見つけることができます。
  [Search (検索)] ボックスに目的の語句を入力し、検索アイコンをクリックしてください。
- 製品情報、設定の詳細、リリース情報などのドキュメント。
  記事を見つけるには、Knowledge Portal のホームページにアクセスし、製品に該当するソリューション ファミリを選択するか、[View All Products (すべての製品を表示)] ボタンをクリックします。

Knowledge Portal のホームページからは、次の操作を実行できます。

- Kofax Community (Kofax コミュニティ) へのアクセス (全カスタマー)。
  [Resources (リソース)] メニューで、**[Community (コミュニティ)]** リンクをクリックします。
- Kofax Customer Portal (Kofax カスタマー ポータル) へのアクセス (一部のカスタマーのみ)。
  [Support Portal Information (サポート ポータルの情報)] ページに移動し、**[Log in to the Customer Portal (カスタマー ポータルにログイン)]** をクリックします。
- Kofax Partner Portal (Kofax パートナー ポータル) へのアクセス (一部のパートナーのみ)。
  [Support Portal Information] ページに移動し、**[Log in to the Partner Portal (パートナー ポータルにログイン)]** をクリックします。
- Kofax サポート コミットメント、ライフサイクル ポリシー、電子フルフィルメントの詳細、セルフサービス ツールへのアクセス。
  [Support Details (サポートの詳細)] ページに移動し、適切な記事を選択します。

第 1 章

Java プログラマー ガイド

この章では、Kofax RPA レガシー Java API を使用してロボットを実行する方法について説明します。このガイドは、簡単なロボットの作成方法に関する知識があり、Java プログラミング言語に精通していることを前提としています。

`printStackTrace` メソッドは Kofax Kapow バージョン 9.6 以降では非推奨です。

Kofax RPA 製品のドキュメント サイトの Application Programming Interface セクションで、特定の Java クラスに関する情報を見つけることができます。https://docshield.kofax.com/Portal/Products/RPA/11.5.0-nlfihq5gwr/RPA.htm。この情報は、オフライン ドキュメント フォルダでも入手できます。詳細については、『Kofax RPA インストール ガイド』を参照してください。

Java の基本

Management Console で実行されるロボットは、Java API を使用して実行されます。これにより、リクエストを `RoboServer` に送信し、特定のロボットを実行するように指示します。これは、Management Console がクライアントとして機能し、`RoboServer` がサーバーとして機能する、クラシックなクライアント/サーバー設定です。

API を使用することにより、任意の Java ベースのアプリケーションを `RoboServer` のクライアントにすることができます。データベースにデータを保存するロボットを実行することに加えて、ロボットがクライアント アプリケーションに直接データを返すようにすることもできます。ここではいくつかの例を示します:

- 複数のロボットを使用して、複数のソースからの結果をリアルタイムで集約する統合検索を実行します。
- アプリケーションのバックエンドのイベントに応じてロボットを実行します。たとえば、新しいユーザーがサインアップしたときにロボットを実行して、バックエンドに直接統合されていない Web ベースのシステムにアカウントを作成します。

このガイドでは、コア class とそれらを使用してロボットを実行する方法を紹介します。また、ロボットに入力を提供し、`RoboServer` での実行をコントロールします。

Java API は、Kofax RPA インストール フォルダ内の `/API/legacy/robosuite-java-api/lib/robosuite-api.jar` にある JAR ファイルに配置されています。詳細については、インストール ガイド の「Kofax RPA の重要なフォルダ」を参照してください。このガイドのすべての例は `/API/robosuite-java-api/examples` にも含まれています。Java API の隣には、API の外部依存関係を構成する 5 つの追加の JAR ファイルがあります。ロボットの実行などの最も基本的な API タスクは、これらのサードパーティ ライブラリを使用せずに実行できますが、一部の高度な機能ではこれらのライブラ

リを 1 つ以上使用する必要があります。このガイドの例では、そのようなライブラリがいつ必要になるかを指定しています。

Java API JAR ファイルを使用するには JAXP バージョン 1.5 が必要であるため、Xalan-Java 2.7.2 などのレガシーの実装は Java JAR ファイルでは機能しません。

最初の例

以下は、`NewsMagazine.robot` という名前のロボットを実行するために必要なコードで、デフォルトプロジェクトの Tutorials フォルダにあります。ロボットは、Return Value ステップ アクションを使用して結果を書き込みます。これにより、API を使用してプログラムで出力を簡単に処理できます。他のロボット (通常 Management Console によるスケジュールで実行) は、データベース データ登録ステップアクションを使用してデータを直接データベースに保存します。この場合、ロボットによって収集されたデータは API クライアントに返されません。

次の例では、NewsMagazine ロボットが実行され、出力がプログラムで処理されます。

入力なしでロボットを実行します:

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
 import com.kapowtech.robosuite.api.java.rql.*;
 import com.kapowtech.robosuite.api.java.rql.construct.*;

 /**
 * Example that shows you how to execute NewsMagazine.robot from tutorial1
 */
 public class Tutorial1 {

     public static void main(String[] args) throws ClusterAlreadyDefinedException {

         RoboServer server = new RoboServer("localhost", 50000);
         boolean ssl = false;
         Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);

         Request.registerCluster(cluster); // you can only register a cluster once per
 application

         try {
             Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
             request.setRobotLibrary(new DefaultRobotLibrary());
             RQLResult result = request.execute("MyCluster");

             for (Object o : result.getOutputObjectsByName("Post")) {
                 RQLObject value = (RQLObject) o;
                 String title = (String) value.get("title");
                 String preview = (String) value.get("preview");
                 System.out.println(title + ", " + preview);
             }
         }
     }
 }
```

次の表に、関係するクラスとその責任を示します。

`RoboServer`	これは、ロボットを実行できる RoboServer を識別する単純な値オブジェクトです。各 RoboServer を Management Console でアクティブにして、使用する前に KCU を割り当てる必要があります。

`Cluster`	クラスタとは、単一の論理ユニットとして機能する RoboServer のグループです。
`Request`	この class は、ロボット リクエストを作成するために使用されます。リクエストを実行する前に、リクエスト class のあるクラスタを登録する必要があります。
`DefaultRobotLibrary`	ロボット ライブラリは、リクエストで特定されたロボットをどこで見つけるかを RoboServer に指示します。後の例では、さまざまなロボット ライブラリの種類と、それらをいつどのように使用するかについて説明します。
`RQLResult`	この class には、ロボットの実行結果が含まれます。結果には、ログおよびサーバー メッセージを含む値応答が含まれます。
`RQLObject`	戻り値アクションを使用してロボットから返される各値には、`RQLObject` としてアクセスできます。

次の行は API に RoboServer が localhost ポート 50000 で実行されていることを知らせます。

```
RoboServer server = new RoboServer("localhost", 50000);
```

次のコードは、単一の RoboServer のあるクラスタを定義します。クラスタはリクエスト class に登録され、このクラスタでリクエストを実行できるようになります。各クラスタは 1回のみ登録できます。

クラスタの登録:

```
boolean ssl = false;
 Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);
 Request.registerCluster(cluster);
```

以下のコードは、リクエストに設定されたロボットライブラリを参照して、 `Library:/Tutorials.Library:/` にある `NewsMagazine.robot` というロボットを実行するリクエストを作成します。ここでは、`DefaultRobotLibrary` が使用され、 `RoboServer` サーバー用のローカル ファイル システムでロボットを探すよう指示しています。ロボット ライブラリの使用方法の詳細については ロボット ライブラリ を参照してください。

```
Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
 request.setRobotLibrary(new DefaultRobotLibrary());
```

次の行は、`MyCluster` という名前のクラスタ (以前に登録されたクラスタ) でロボットを実行し、ロボットが完了すると結果を返します。デフォルトでは、ロボットが API 例外を生成すると、`execute` が例外をスローします。

```
RQLResult result = request.execute("MyCluster")
```

ここで、抽出された値を処理します。まず、Post という名前のタイプのすべての抽出値を取得し、それらを反復処理します。各 `RQLObject` について、Post タイプの属性にアクセスし、結果を出力します。属性とマッピングについては、後のセクションで説明します。

```
for (Object o : result.getOutputObjectsByName("Post")) {
     RQLObject value = (RQLObject) o;
     String title = (String) value.get("title");
     String preview = (String) value.get("preview");
     System.out.println(title + ": " + preview);
 }
```

ロボット入力

API を介して実行されるほとんどのロボットは、検索キーワードやログイン資格情報などの入力を通じてパラメータ化されます。ロボットへの入力は、RoboServer のリクエストの一部で、リクエストの `createInputVariable` メソッドを使って提供されます。

暗黙的な **RQLObjectBuilder** を使用した入力**:**

```
Request request = new Request("Library:/Input.robot");
 request.createInputVariable("userLogin").setAttribute("username", "scott")
 .setAttribute("password", "tiger");
```

この例では、`Request` が作成され、 `createInputVariable` が `userLogin` という名前の入力変数を作成するために使用されます:。次に、`setAttribute` が入力変数のユーザー名とパスワードの属性を設定するために使用されます。

上記の例は一般的な略記法ですが、RQLObjectBuilder を使用してより詳細に表現することもできます:

明示的な **RQLObjectBuilder** を使用した入力**:**

```
Request request = new Request("Library:/Input.robot");
 RQLObjectBuilder userLogin = request.createInputVariable("userLogin");
 userLogin.setAttribute("username", "scott");
 userLogin.setAttribute("password", "tiger");
```

2 つの例は同じです。1 つ目は、匿名の `RQLObjectBuilder` でカスケード メソッド呼び出しを使用するため、より短くなります。

RoboServer がこのリクエストを受信すると、次のことが発生します:

- RoboServer は `Input.robot` をロードします (リクエスト用に設定された `RobotLibrary` から) 。
- RoboServer は口ボットに `userLogin` という名前の変数があり、この変数が入力としてマークされていることを確認します。
- RoboServer は、 `setAttribute` を使用して設定された属性が変数の `userLogin` のタイプと互換性があるかを検証します。その結果、タイプにはユーザー名とパスワードという名前の属性が必要であり、両方ともテキスト ベースの属性である必要があります (次のセクションでは、API と Design Studio 属性のマッピングを説明します) 。
- すべての入力変数に互換性がある場合、RoboServer は口ボットの実行を開始します。

口ボットが複数の入力変数を必要とする場合、それらをすべて作成して口ボットを実行する必要があります。設定する必要があるのは必須属性のみで、API を使用して設定しない必須属性はすべて null 値になります。Facebook と Twitter への口グインを必要とする口ボットがある場合、このような入力を定義できます。

```
Request request = new Request("Library:/Input.robot");
 request.createInputVariable("facebook").setAttribute("username", "scott")
 .setAttribute("password", "facebook123");
 request.createInputVariable("twitter").setAttribute("username", "scott")
 .setAttribute("password", "twitter123");
```

属性タイプ

Design Studio で新しいタイプを定義するとき、各属性のタイプを選択します。一部の属性には、ショート テキスト、ロング テキスト、パスワード、HTML、XML などのテキストを含めることができ、口ボッ

ト内で使用する場合、これらの属性にテキストを保存する必要がある場合があります。XML 属性にテキストを保存する場合、テキストは有効な XML ドキュメントである必要があります。この検証は、タイプがロボット内で使用される場合に発生しますが、API はタイプについて何も認識しないため、同じ方法で属性値を検証しません。そのため、API には 8 つの属性タイプがあり、Design Studio には 19 種類のタイプがあります。この表は、API と Design Studio 属性タイプのマッピングを示しています。

API から **Design Studio** へのマッピング

API 属性タイプ	**Design Studio** の属性タイプ
テキスト	ショート テキスト、ロング テキスト、パスワード、HTML、XML、プロパティ、言語、国、通貨、キーの再検索
整数	整数
ブール値	ブール値
数	数
文字	文字
日付	日付
セッション	セッション
バイナリ	バイナリ、画像、PDF

API 属性タイプは、次の方法で Java にマップされます。

属性の **Java** タイプ

API 属性タイプ	**Java Class**
テキスト	`java.lang.String`
整数	`java.lang.Long`
ブール値	`java.lang.Boolean`
数	`java.lang.Double`
文字	`java.lang.Character`
日付	`java.util.Date`
セッション	`com.kapowtech.robosuite.api.construct.Session`
バイナリ	`com.kapowtech.robosuite.api.construct.Binary`

`RQlObjectBuilder` の `setAttribute` メソッドは、オーバーロードされているため、適切な Java class が引数として使用されている場合、API を介して属性を設定する際に属性タイプを明示的に指定する必要はありません。オブジェクトの属性に可能な Design Studio 属性タイプをすべて設定する方法を次の例に示します。

`setAttribute` の推奨使用法**:**

```
Request request  = new Request("Library:/AllTypes.robot");
 RQLObjectBuilder inputBuilder = request.createInputVariable("AllTypes");
 inputBuilder.setAttribute("anInt", new Long(42L));
 inputBuilder.setAttribute("aNumber", new Double(12.34));
 inputBuilder.setAttribute("aBoolean", Boolean.TRUE);
 inputBuilder.setAttribute("aCharacter", 'c');
```

```
inputBuilder.setAttribute("aShortText", "some text");
inputBuilder.setAttribute("aLongText", "a longer test");
inputBuilder.setAttribute("aPassword", "secret");
inputBuilder.setAttribute("aHTML", "<html>text</html>");
inputBuilder.setAttribute("anXML", "<tag>text</tag>");
inputBuilder.setAttribute("aDate", new Date());
inputBuilder.setAttribute("aBinary", new Binary("some bytes".getBytes()));
inputBuilder.setAttribute("aPDF", (Binary) null);
inputBuilder.setAttribute("anImage", (Binary) null);
inputBuilder.setAttribute("aProperties", "name=value\nname2=value2");
inputBuilder.setAttribute("aSession", (Session) null);
inputBuilder.setAttribute("aCurrency", "USD");
inputBuilder.setAttribute("aCountry", "US");
inputBuilder.setAttribute("aLanguage", "en");
inputBuilder.setAttribute("aRefindKey", "Never use this a input");
```

上記の例では、新しい `Long(42L)` と新しい `Double(12.34)` を明示的に使用していますが、自動ボクシングのためには 42L と 12.34 で十分です。また、Java コンパイラーはオーバーロードされた `setAttribute` メソッドを呼び出す判断ができないため、null 値をキャストする必要があることに注意してください。ただし、未構成の属性は自動的に null になるため、null を明示的に設定する必要はありません。

API を使用して入力を作成するときに、属性と `AttributeType` を明示的に指定することができます。このアプローチは推奨されませんが、まれに必要になる場合があり、次のようになります。

`setAttribute` の誤った使用法:

```
Request request  = new Request("Library:/AllTypes.robot");
 RQLObjectBuilder inputBuilder = request.createInputVariable("AllTypes");
 inputBuilder.setAttribute(new Attribute("anInt", "42", AttributeType.INTEGER));
 inputBuilder.setAttribute(new Attribute("aNumber", "12.34", AttributeType.NUMBER));
 inputBuilder.setAttribute(new Attribute("aBoolean", "true", AttributeType.BOOLEAN));
 inputBuilder.setAttribute(new Attribute("aCharacter", "c", AttributeType.CHARACTER));
 inputBuilder.setAttribute(new Attribute("aShortText", "some text",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aLongText", "a longer test",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aPassword", "secret", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aHTML", "<html>bla</html>",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("anXML", "<tag>text</tag>",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aDate", "2012-01-15 23:59:59.123",
   AttributeType.DATE));
 inputBuilder.setAttribute(new Attribute("aBinary",
   Base64Encoder.encode("some bytes".getBytes()), AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("aPDF", null, AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("anImage", null, AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("aProperties", "name=value\nname2=value2",
   AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aSession", null, AttributeType.SESSION));
 inputBuilder.setAttribute(new Attribute("aCurrency", "USD", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aCountry", "US", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aLanguage", "en", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aRefindKey", "Never use this a input",
   AttributeType.TEXT));
```

すべての属性値は、文字列の形式で提供する必要があります。文字列値は、指定された属性タイプに基づいて適切な Java オブジェクトに変換されます。これは、Kofax RPA Java API で他の汎用 API を上に構築する場合にのみ役立ちます。

実行パラメータ

`createInputVariable` メソッドに加えて、リクエストにはロボットが `RoboServer` でどのように実行されるかを制御する多くのメソッドが含まれています。

リクエストに応じた実行制御メソッド

`setMaxExecutionTime(int seconds)`	ロボットの実行時間を制御します。この時間が経過すると、ロボットは RoboServer で停止します。ロボットが実行を開始するまでタイマーは開始されないため、ロボットが RoboServer で待機している場合、これは考慮されません。
`setStopOnConnectionLost(boolean)`	true (デフォルト) の場合、RoboServer がクライアント アプリケーションへの接続が失われたと発見すると、ロボットは停止します。この値を false に設定するには、それなりの理由が必要です。この値を処理するコードが記述されていない場合、アプリケーションは期待どおりに動作しません。
`setStopRobotOnApiException(boolean)`	true (デフォルト) の場合、最初の API 例外が発生した後、ロボットは RoboServer で止まります。デフォルトでは、ロボットのほとんどのステップは、ステップの実行に失敗すると API 例外を発生させます。ステップの [エラー処理] タブでこの値を設定します。 false に設定すると、ロボットは API 例外に関係なく実行を続けます。ただし、アプリケーションが `RequestExecutor` ストリーミング実行モードを使用しない場合は例外が `execute()` でスローされます。false に設定するときは注意してください。
`setUsername(String), setPassword(String)`	RoboServer 資格情報を設定します。RoboServer は認証を要求するように設定できます。このオプションを有効にすると、クライアントは資格情報を提供するか、RoboServer がリクエストを拒否します。
`setRobotLibrary(RobotLibrary)`	ロボット ライブラリは、リクエストで特定されたロボットをどこで見つけるかを RoboServer に指示します。さまざまなライブラリ タイプとその使用法に関するその他の例については、ロボット ライブラリ を参照してください。
`setExecutionId(String)`	このリクエスト用に `executionId` を設定することができます。指定しない場合は、RoboServer で自動的に生成されます。実行 ID はログ記録に使用され、ロボットをプログラムで停止するためにも必要です。ID は (経時的に) グローバルに一意である必要があります。2 つのロボットが同じ実行 ID を使用している場合、ログの一貫性が失われます。

`setProject(String)`	このメソッドは、ロギングの目的でのみ使用されます。Management Console はこのフィールドを使用してログメッセージをプロジェクトにリンクします。そのため、ログ ビューはプロジェクトでフィルタリングできます。 アプリケーションが `RepositoryRobotLibrary` を使用していない場合、この値を設定して RoboServer ロギングシステムにこのロボットが属するプロジェクト (存在する場合) を通知する必要があります。

ロボット ライブラリ

Design Studio で、ロボットはプロジェクトにグループ化されます。ファイル システムを見ると、これらのプロジェクトが Library という名前のフォルダによって識別されていることがわかります (詳細については Kofax RPA のヘルプ にある「ライブラリとロボット プロジェクト」トピックを参照) 。

`RoboServer` の実行リクエストを作成するとき、ロボットの URL でロボットを識別します:

```
Request request = new Request("Library:/Input.robot");
```

ここでは、`Library:/` がロボット ライブラリへのシンボリック リファレンスで、そこで RoboServer がロボットを探す必要があります。`RobotLibrary` はビルダーで指定されます:

```
request.setRobotLibrary(new DefaultRobotLibrary());
```

3 つの異なるロボットライブラリの実装が利用可能で、選択は展開環境によって異なります。

ロボット ライブラリ

ライブラリ タイプ	説明
`DefaultRobotLibrary`	このライブラリは設定アプリケーションで定義されている現在のプロジェクト フォルダでロボットを探して、RoboServer を設定します。 RoboServer が複数ある場合は、すべての RoboServer でロボットを展開する必要があります。 このロボット ライブラリはキャッシュされないため、ロボットは実行のたびにディスクからリロードされます。このアプローチはロボットが頻繁に変更される開発環境でライブラリを使用可能にしますが、本番環境には適していません。

ライブラリ タイプ	説明
`EmbeddedFileBasedRobotLibrary`	このライブラリは、RoboServer に送信される実行リクエストに埋め込まれています。このライブラリを作成するには、ロボットとすべての依存関係 (タイプ、スニペット、リソース) を含む zip ファイルを作成する必要があります。Design Studio にある **[ツール]** > **[ロボット ライブラリ ファイルの生成]** メニューを使います。 ライブラリはリクエストごとに送信されるため、大規模なライブラリにはオーバーヘッドが追加されますが、ライブラリは RoboServer でキャッシュされ、最高のパフォーマンスを提供します。 1 つの強みは、ロボットとコードを単一のユニットとして展開できることです。これにより検証環境から本番環境へのクリーンな移行が可能になります。ただし、ロボットが頻繁に変更される場合は、頻繁に再デプロイする必要があります。 次のコードを使用して、リクエスト用の組み込みロボット ライブラリを設定できます。 `Request request = new Request("Library:/Tutorials/ NewsMagazine.robot"); RobotLibrary library = new EmbeddedFileBasedRobotLibrary (new FileInputStream ("c:\\embeddedLibrary.robotlib")); request.setRobotLibrary(library);`

ライブラリ タイプ	説明

`RepositoryRobotLibrary`	これは最も柔軟なロボット ライブラリです。
	このライブラリはロボット ライブラリとして Management Console の組み込みリポジトリを使用します。このライブラリを使用すると、RoboServer は Management Console と通信し、ロボットとその依存関係を含むロボット ライブラリを送信します。
	キャッシングは Management Console と RoboServer の内部で、ロボットごとに行われます。Management Console 内部で、生成されたライブラリはロボットとその依存関係に基づいてキャッシュされます。RoboServer で、キャッシュはタイムアウトに基づいているため、Management Console にそれぞれのキャッシュに問い合わせる必要はありません。さらに、RoboServer と Management Console の間で読み込むライブラリは、HTTP パブリック/プライベート キャッシュを使用して、帯域幅をさらに削減します。
	`NewsMagazine.robot` が Management Console にアップロードされた場合、ロボットの実行時にリポジトリ ロボット ライブラリを使用できます。
	`Request request = new Request("Library:/Tutorials/NewsMagazine.robot");`
	`RobotLibrary library = new RepositoryRobotLibrary("http://localhost:50080", "Default Project", 60000);`
	`request.setRobotLibrary(library);`
	このコマンドは、RoboServer がローカルの Management Console からロボットをロードするように指示し、ロボットの新しいバージョン (タイプとスニペット) が変更されたかどうかを Management Console で確認する前に1分間キャッシュします。
	さらに、`Library:/` プロトコルを介してロードされたリソースにより、RoboServer は Management Console からリソースを直接要求します。

Java Advanced

このセクションでは、出力ストリーミング、ロギング、SSL 構成、並列実行などの高度な API 機能について説明します。

負荷分散とフェールオーバー

`RequestExecutor` の内部では、エグゼキューターに RoboServer の配列が与えられます。Executor が構築されると、それぞれ RoboServer に接続しようとします。接続されると、それぞれ RoboServer に ping リクエストを送信し、サーバーの設定方法を検出します。

負荷分散エグゼキューター:

```
RoboServer prod = new RoboServer("prod.kapow.local", 50000);
 RoboServer prod2 = new RoboServer("prod2.kapow.local", 50000);
```

```
Cluster cluster = new Cluster("Prod", new RoboServer[]{ prod, prod2}, false);
Request.registerCluster(cluster);
```

RoboServer の未使用の実行スロットの数に基づいて、負荷はクラスタ内の各オンライン RoboServer に分散されます。次のリクエストは常に、使用可能なスロットが最も多い RoboServer に配信されます。使用可能な実行スロットの数は、最初の ping 応答を通じて取得され、エグゼキューターは起動したロボットと完了したロボットを追跡します。RoboServer の実行スロットの数は、Management Console > 管理 > RoboServer セクションの [最大同時ロボット数] の設定によって決まります。

RoboServer がオフラインになると、ping リクエストに正常に応答するまでロボット実行リクエストを受信しません。

1 つのクライアント ルール

デフォルトでは、API 接続は 20 件の接続に制限されています。ただし、最高のパフォーマンスを確保するには、RoboServer の特定のクラスタを使用する API クライアントを 1 つだけにすることをお勧めします。同じ RoboServer に対してロボットを実行している JVM が多すぎる場合は、パフォーマンスが低下します。

以下は推奨されませんが、より大きなボリュームを処理する必要がある環境では、common.conf ファイルの `kapow.max.multiplexing.clients` システムプロパティを調整することにより、接続制限を設定できます。

実行ログ

リクエストを実行すると、ロボットがエラーを生成した場合、execute メソッドは例外をスローします。他のタイプのエラーと警告は、`ExecutorLogger` のインターフェイスを通じて報告されます。前の例では、ロボットの実行時に `ExecutionLogger` が提供されませんでした。これは `System.out` に書き込むデフォルトの実装です。

以下に、RoboServer のいずれか 1 つがオフラインになった場合の `ExecutorLogger` からの報告の例を示します。この例では、クラスタはオンラインではない RoboServer で設定されます。

ExecutorLogger、オフライン サーバーの例:

```
RoboServer rs = new RoboServer("localhost", 50000);
Cluster cluster = new Cluster("name", new RoboServer[]{rs}, false);
Request.registerCluster(cluster);
```

この例を実行すると、次の内容がコンソールに書き込まれます。

ExecutorLogger、オフライン **RoboServer** コンソール出力:

```
RoboServer{host='localhost', port=50000} went offline.
Connection refused
```

アプリケーションが `System.out` に直接書き込む必要がない場合は、クラスタを登録するときに、異なる `ExecutorLogger` 実装を提供できます。

DebugExecutorLogger を使用:

```
Request.registerCluster(cluster, new DebugExecutorLogger());
```

この例は、`DebugExecutorLogger()` を使用し、これは `System.out` に書き込みますが、API デバッグが有効になっている場合のみです。エラーメッセージの処理方法を制御するために `ExecutorLogger` の独自の実装を提供できます。詳細については、`ExecutorLogger` の JavaDoc を確認してください。

データ ストリーミング

ロボットの実行結果をリアルタイムで表示する必要がある場合は、ロボットが実行を終了して `RQLResult` へアクセスするのを待つ代わりに、API を使用してすぐに抽出した値を返却できます。

API は、API がロボットから返された値を受け取るたびにコールバックを受け取る可能性を提供します。これは、`RobotResponseHandler` インターフェイスを通じて行います。

AbstractFailFastRobotResponseHandler を使用した応答ストリーミング:

```
public class DataStreaming {

    public static void main(String[] args) throws ClusterAlreadyDefinedException {

        RoboServer server = new RoboServer("localhost", 50000);
        Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, false);
        Request.registerCluster(cluster);

        try {
            Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
            RobotResponseHandler handler = new AbstractFailFastRobotResponseHandler()
{
                public void handleReturnedValue(RobotOutputObjectResponse response,
Stoppable stoppable) throws RQLException {
                    RQLObject value = response.getOutputObject();
                    Long personId = (Long) value.get("personId");
                    String name = (String) value.get("name");
                    Long age = (Long) value.get("age");
                    System.out.println(personId + ", " + name + ", " + age);
                    }
                };
            request.execute("MyCluster", handler);
        }
    }
}
```

上記の例では、リクエストの 2 番目の execute メソッドを使用しています。ロボットを実行するクラスタの名前に加えて `RobotResponseHandler` を想定しています。この例では、デフォルトのエラー処理を提供する `AbstractFailFastRobotResponseHandler` を拡張して `RobotResponseHandler` を作成し、ロボットが返す値のみを処理します。

`handleReturnedValue` メソッドは、RoboServer からAPI が戻り値を受け取るたびに呼び出されます。この例で使用されている `AbstractFailFastRobotResponseHandler` は、非ストリーミング実行メソッドと同じ方法で例外をスローします。これは、ロボットによって生成された API 例外に応じて例外がスローされることを意味します。

`RobotResponseHandler` は、3 つのカテゴリにグループ化できるいくつかのメソッドがあります。

ロボットのライフ サイクル イベント

RoboServer で、実行の開始時と終了時など、ロボットの実行状態が変化したときに呼び出されるメソッド。

ロボット データ イベント

ロボットが API にデータまたはエラーを返すときに呼び出されるメソッド。

追加のエラー処理
RoboServer または API の内部エラーのため呼び出されるメソッド。

RobotResponseHandler - ロボットのライフ サイクル イベント

メソッド名	説明
`void requestSent(RoboServer roboServer, ExecuteRequest request)`	`RequestExecutor` がリクエストを実行するサーバーを見つけたら呼び出されます。
`void requestAccepted(String executionId)`	RoboServer がリクエストを受け入れキューに入れた時に呼び出されます。
`void robotStarted(Stoppable stoppable)`	RoboServer がロボットの実行を開始するときに呼び出されます。RoboServer の負荷が大きいか、複数の API クライアントで使用されていない場合、通常、ロボットがキューに入れられた直後に発生します。
`void robotDone(RobotDoneEvent reason)`	ロボットの実行が RoboServer で完了したときに呼び出されます。`RobotDoneEvent` は、エラーのために実行が正常に終了したか、停止したかを指定するために使用されます。

RobotResponseHandler - ロボット データ イベント

メソッド名	説明
`void handleReturnedValue(RobotOutputObjectResponse response, Stoppable stoppable)`	ロボットが Return Value アクションを実行すると呼び出され、値はソケットを介して API に返されます。
`void handleRobotError(RobotErrorResponse response, Stoppable stoppable)`	ロボットが API 例外を発生させたときに呼び出されます。通常の状況では、ロボットは最初のAPI例外の後に実行を停止します。この動作は `Request.setStopRobotOnApiException(false)` を使用してオーバーライドできます。この場合、このメソッドは複数回呼び出されます。このアプローチは生成されたエラーに関係なく、データ ストリーミング ロボットの実行を継続する場合に役立ちます。
`void handleWriteLog(RobotMessageResponse response, Stoppable stoppable)`	ロボットがログの書き込みステップを実行するときに呼び出されます。これは、ロボットから追加のログ情報を提供するのに役立ちます。

RobotResponseHandler - 追加のエラー処理

メソッド名	説明
`void handleServerError(ServerErrorResponse response, Stoppable stoppable)`	RoboServer がエラーを生成すると呼び出されます。たとえば、サーバーがビジー状態でリクエストを処理できない場合、または RoboServer 内部でエラーが発生した場合に発生する可能性があり、ロボットの起動を防ぎます。
`handleError(RQLException e, Stoppable stoppable)`	API 内でエラーが発生した場合、または最も一般的には、クライアントが RoboServer への接続を失った場合に呼び出されます。

メソッドの多くには `Stoppable` オブジェクトが含まれており、特定のエラーまたは返された値に応じて停止するために使用できます。

一部のメソッドでは `RQLException`をスローでき、帰結が得られる場合があります。ハンドラーを呼び出すスレッドは、`Request.execute()` を呼び出すスレッドです。スローされた例外は、呼び出しスタックをオーバーロードする可能性があります。`handleReturnedValue`、`handleRobotError` または `handleWriteLog` に応じて例外をスローした場合、`Stoppable.stop()` を呼び出すのはあなたの責任です。そうしない場合、ロボットは `Request.execute()` への呼び出しが完了しても実行し続ける可能性があります。

データ ストリーミングは、次の使用例のいずれかで最もよく使用されます。

- 結果がリアルタイムでユーザーに表示される Ajax ベースの Web アプリケーション。データがストリーミングされない場合、ロボットの実行が完了するまで結果を表示できません。
- クライアントがロボットの実行中にすべてをメモリに保持することができない程大量のデータを返すロボット。
- 抽出された値がロボットの実行と並行して処理されるように最適化する必要があるプロセス。
- カスタム形式でデータベースにデータを保存するプロセス。
- API 例外のカスタム処理を無視または必要とする ロボット (次の例を参照) 。

AbstractFailFastRobotResponseHandler を使用した応答とエラーの収集:

```
public class DataStreamingCollectErrorsAndValues {

 public static void main(String[] args) throws ClusterAlreadyDefinedException {

  RoboServer server = new RoboServer("localhost", 50000);
  Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, false);
  Request.registerCluster(cluster);

  try {
    Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
    request.setStopRobotOnApiException(false);  // IMPORTANT!!
    request.setRobotLibrary(new DefaultRobotLibrary());
    ErrorCollectingRobotResponseHandler handler =
      new ErrorCollectingRobotResponseHandler();
    request.execute("MyCluster", handler);

    System.out.println("Extracted values:");
    for (RobotOutputObjectResponse response : handler.getOutput()) {
        RQLObject value = response.getOutputObject();
        Long personId = (Long) value.get("personId");
        String name = (String) value.get("name");
        Long age = (Long) value.get("age");
        System.out.println(personId + ", " + name + ", " + age);
     }

     System.out.println("Errors:");
     for (RobotErrorResponse error : handler.getErrors()) {
         System.out.println(error.getErrorLocationCode() + ", " +
           error.getErrorMessage());
     }
  }
}

private static class ErrorCollectingRobotResponseHandler extends
       AbstractFailFastRobotResponseHandler {

   private List<RobotErrorResponse> _errors =
```

```
        new LinkedList<RobotErrorResponse>();
   private List<RobotOutputObjectResponse> _output =
        new LinkedList<RobotOutputObjectResponse>();
   public void handleReturnedValue
     (RobotOutputObjectResponse response, Stoppable stoppable)
     throws RQLException {
    _output.add(response);
}

@Override
public void handleRobotError(RobotErrorResponse response,
    Stoppable stoppable) throws RQLException {
  // do not call super as this will stop the robot
  _errors.add(response);
}

public List<RobotErrorResponse> getErrors() {
     return _errors;
}

public List<RobotOutputObjectResponse> getOutput() {
   return _output;
  }
 }
}
```

上記の例は、返された値とエラーを収集する `RobotResponseHandler` の使い方を示します。このタイプのハンドラは、エラーが発生した場合でもロボットが実行を継続する必要がある場合に役立ちます。これは、Web サイトが不安定で、時々タイムアウトする場合に役立ちます。ロボット エラー (API 例外) のみがハンドラーによって収集されることに注意してください。`RoboServer` への接続が失われた場合、`Request.execute()` は引き続き `RQLException` をスローし、RoboServer はロボットを停止します。

詳細については、`RobotResponseHandler` の JavaDoc を参照してください。

SSL

API は、`RQLService` を通じて RoboServer と通信し、これは特定のネットワーク ポートで API リクエストをリッスンする RoboServer コンポーネントです。RoboServer を開始すると、暗号化された SSL サービス、プレーン ソケット サービス、またはその両方 (2 つの異なるポートを使用) を使用するかどうかを指定します。クラスタ内のすべての RoboServer が同じ `RQLService` を実行している必要があります (ただし、ポートは異なる場合があります)。

ポート 50043 で SSL を含む `RQLService` RoboServer を開始したとみなします:

```
RoboServer -service ssl:50043
```

次のコードが使用されます。

SSL 構成

```
RoboServer server = new RoboServer("localhost", 50043);
 boolean ssl = true;
 Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, ssl);
 Request.registerCluster(cluster);
```

クラスタを SSL クラスタとして作成し、それぞれ RoboServer が使用する SSL ポートを指定する必要があります。これで、RoboServer とAPI のすべての通信は暗号化されます。

この例を機能させるには、`not-yet-commons-ssl-0.3.17.jar` が `classpath` アプリケーションで必要になります。Kofax RPA インストール内部で、API .jar ファイルの隣にあります。

SSL では、データの暗号化に加えて、リモート パーティの身元を確認することができます。このタイプの確認は、インターネット上で非常に重要です。多くの場合、API クライアントと RoboServer は同じローカル ネットワーク上にあるため、互いの ID を確認する必要はほぼありませんが、必要になる場合もあるため API はこの機能をサポートしています。

本人確認はほとんど使用されないため、このガイドでは説明しません。興味がある場合は、Java API に含まれている SSL の例を参照してください。

並列実行

リクエストの両方の実行メソッドはブロックされています。つまり、各ロボットの実行にはスレッドが必要です。前のセクションの例では、メイン スレッドでのロボットの直接実行を示しました。これは、一度に 1 つのロボットしか順番に実行できないため、通常は好ましくありません。

次の例では、2 つのチュートリアル ロボットを並行して実行します。この例では、マルチスレッド用の `java.util.concurrent` ライブラリが使用されています。

マルチスレッドの例

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
 import com.kapowtech.robosuite.api.java.rql.*;
 import com.kapowtech.robosuite.api.java.rql.construct.*;
 import com.kapowtech.robosuite.api.java.rql.engine.hotstandby.*;

 import java.util.concurrent.*;

 public class ParallelExecution {

     public static void main(String[] args) throws Exception {

         RoboServer server = new RoboServer("localhost", 50000);
         Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server},
          false);
         Request.registerCluster(cluster);

         int numRobots = 4;
         int numThreads = 2;
         ThreadPoolExecutor threadPool = new ThreadPoolExecutor(numThreads,
           numThreads, 10, TimeUnit.SECONDS, new LinkedBlockingQueue());
         for (int i = 0; i < numRobots; i++) {
           Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
           request.setRobotLibrary(new DefaultRobotLibrary());
           threadPool.execute(new RobotRunnable(request));
         }
         threadPool.shutdown();
         threadPool.awaitTermination(60, TimeUnit.SECONDS);
     }

     // --------------------------------------------------------------
    // Inner classes
     // --------------------------------------------------------------
    static class RobotRunnable implements Runnable {

         Request _request;

         RobotRunnable(Request request) {
             _request = request;
```

```
        }

        public void run() {

            try {
                RQLResult result = _request.execute("MyCluster");
                System.out.println(result);
            }
        }
    }
}
```

上記の例では、2 つのスレッドの `ThreadPoolExecutor` と 4 つの `RobotRunnables` が作成され、スレッド プールで実行します。スレッド プールには 2 つのスレッドがあるため、2 つの口ボットがすぐに実行を開始します。残りの 2 台の口ボットは、`LinkedBlockingQueue` で配置され、最初の 2 つの口ボットが実行を終了し、スレッド プールのスレッドが使用可能になった後に、順番に実行されます。

リクエストは変更が可能であり、条件の発生を避けるため、リクエストは execute メソッド内で複製されることに注意してください。リクエストは変更が可能であるため、別々のスレッドで同じリクエストを変更しないでください。

リポジトリ統合

Management Console では、スケジュールされた口ボット、および REST サービスとして実行される口ボットの実行に使用される RoboServer のクラスタの指定も行います。API では `RepositoryClient` を使って Management Console からクラスタ情報が取得できます。詳細については `RepositoryClient` ドキュメントを参照してください。

リポジトリ統合:

```
public class RepositoryIntegration {
     public static void main(String[] args) throws Exception {

      RepositoryClient client = RepositoryClientFactory.createRepositoryClient
        ("http://localhost:50080", null, null);
      Request.registerCluster(client, "Cluster 1");

      Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
      request.setRobotLibrary(new DefaultRobotLibrary());
      RQLResult result = request.execute("MyCluster");
      System.out.println(result);
    }
}
```

上記の例は、localhost でデプロイされた Management Console に接続する `RepositoryClient` を作成する方法を示しています。この例を機能させるには、クラスパスに `commons-logging-1.1.1.jar`、`commons-codec-1.4.jar`、`commons-httpclient-4.1.jar` を含める必要があります。

認証が有効になっていないため、ユーザー名とパスワードに null が渡されます。`RepositoryClient` を登録するとき、Management Console 上に存在するクラスタ名を指定します。その後、Management Console に照会を行ってこのクラスタに設定された RoboServer のリストを取得し、2 分ごとに Management Console 上でクラスタ設定が更新されているかどうかを確認します。

この統合により、Management Console ユーザー インターフェイスを使って、動的に変更できるクラスタを Management Console 上で作成することができます。API を使用する Management Console クラ

スタは 1 つである必要があり、OneClient ルールに違反するため、ロボットのスケジューリングには使用しないでください。

内部処理

このセクションでは、クラスタを登録してリクエストを実行する際に内部で何が起こっているのかを説明します。

リクエストでクラスタを登録すると、裏で `RequestExecutor` が作成されます。この `RequestExecutor` は、クラスタ名をキーとしてマップに保存されます。リクエストが実行されると、提供されたクラスタ名を使用して、関連付けられた `RequestExecutor` を探し、リクエストを実行します。

通常の実行:

```
public static void main(String[] args) throws InterruptedException,
       RQLException {

   RoboServer server = new RoboServer("localhost", 50000);
   Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
   Request.registerCluster(cluster);
   Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
   request.setRobotLibrary(new DefaultRobotLibrary());
   RQLResult result = request.execute("MyCluster");
   System.out.println(result);
}
```

次に、`hiddenRequestExecutor` を使用して同じ例を記述します。

内部処理の実行中:

```
public static void main(String[] args) throws InterruptedException,
    RQLException {

  RoboServer server = new RoboServer("localhost", 50000);
  Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
  RequestExecutor executor = new RequestExecutor(cluster);

  Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
  request.setRobotLibrary(new DefaultRobotLibrary());
  RQLResult result = executor.execute(request);
  System.out.println(result);
 }
```

`RequestExecutor` はデフォルトで非表示になっているため、管理する必要はありません。クラスタごとに一つの `RequestExecutor` しか作成できないので、直接使用する場合はアプリケーション全体でその参照を保存する必要があります。`Request.registerCluster(cluster)` を使用することは `RequestExecutor` およびライフ サイクル ルールを無視できることを意味します。

`RequestExecutor` は、必要な状態とロジックが含まれており、負荷分散とフェイルオーバー機能を提供します。`RequestExecutor` を直接使用すると、いくつかの追加機能も使用できます。

RequestExecutor の機能

`RequestExecutor` がリポジトリに接続されていない場合、`addRoboServer(..)` と `removeRoboServer(..)` を呼び出すことで、RoboServer を動的に追加または削除できます。これらのメソッドは、`RequestExecutor` の内部で使用される分配リストを変更します。

`RequestExecutor.getTotalAvailableSlots()` は、内部分配リスト内にあるすべての RoboServer で未使用の実行スロットの数を返します。

これらのメソッドを使用すると、使用可能な実行スロットの数が少なくなった場合に、RoboServer を `RequestExecutor` に動的に追加することができます。

`RequestExecutor` を作成するとき、オプションとして `RQLEngineFactory` を提供できます。`RQLEngineFactory` では、RoboServer に接続する時に使用される `RQLProtocol` をカスタマイズすることができます。これは、クライアント証明書を使用してセキュリティを強化する場合など、まれな状況でのみ必要です。詳細については、『Kofax RPA 管理者ガイド』の証明書の章を参照してください。

Web アプリケーション

`RequestExecutor` には、RoboServer へのリクエストの送受信、および既知の RoboServer への定期的な ping 送信に使用される内部スレッドの数が含まれています。これらのスレッドはすべてデーモンとしてマークされています。つまり、メインス レッドが存在するときに JVM の停止を妨げません。デーモンスレッドの詳細については、スレッド JavaDoc を参照してください。

`RequestExecutor` を Web アプリケーション内で使用する場合は、JVM の寿命は Web アプリケーションよりも長く、Web コンテナの実行中に Web アプリケーションをデプロイおよびアンデプロイできます。つまり、Web アプリケーションは、作成したスレッドを停止する役割があります。Web アプリケーションがスレッドを停止しない場合、Web アプリケーションをアン デプロイするとメモリ リークが発生します。実行中のスレッドが参照するオブジェクトは、スレッドが停止するまでガベージ コレクションできないため、メモリ リークが発生します。

Web アプリケーション内で `RequestExecutor` を使用する場合は、コードはこれらの内部スレッドをシャットダウンする必要があります。これは、コードが明示的に `RequestExecutor` を作成した場合、`Request.shutdown()` または `RequestExecutor.shutdown()` を呼び出すことによって行われます。

この例は、Web アプリケーションがアンデプロイされたときに `ServletContextListener` を使って API を正しくシャットダウンする例を示しています。アプリケーション web.xml でコンテキスト リスナーを定義する必要があります。

Web アプリケーションの適切なシャットダウン:

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
 import com.kapowtech.robosuite.api.java.rql.*;
 import com.kapowtech.robosuite.api.java.rql.construct.*;

 import javax.servlet.*;

 public class APIShutdownListener implements ServletContextListener {
     public void contextInitialized(ServletContextEvent servletContextEvent) {
         RoboServer server = new RoboServer("localhost", 50000);
         Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server},
```

```
            false);
        try {
            Request.registerCluster(cluster);
        }
        catch (ClusterAlreadyDefinedException e) {
            throw new RuntimeException(e);
        }
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        Request.shutdown();
    }
}
```

`contextDestroyed` は Web コンテナがアプリケーションをアンデプロイするときに呼び出されます。`Request.shutdown()` は、非表示の `RequestExecutor` のすべての内部スレッドが確実に正しく停止するために呼び出されます。

`contextInitialized` は未チェックの例外を投げることができないので、`ClusterAlreadyDefinedException` を `RunTimeException`.でラップする必要があります。Java ウェブコンテナのクラスローダ階層のため、アプリケーションを 2 回デプロイした場合にこの例外が発生する可能性があります。API .jar ファイルが個々のアプリケーション クラス ローダーではなく、共通のクラス ローダーによってロードされた場合にのみ発生します。

API デバッグ

API は、デバッグの目的で追加情報を提供できます。API デバッグを有効にするには、システム プロパティ `DEBUG_ON` を設定する必要があります。このプロパティの値は、API のパッケージ/クラス名である必要があります。

たとえば、API と `RoboServer` の間のデータ送信に関心がある場合は、パッケージ `com.kapowtech.robosuite.api.java.rql.io` のデバッグ情報を要求できます。開発中に、コードでシステム プロパティを直接設定してこれを行います:

デバッグの有効化:

```
System.setProperty("DEBUG_ON", "com.kapowtech.robosuite.api.java.rql.io");
 RoboServer server = new RoboServer("localhost", 50000);
 Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
 Request.registerCluster(cluster);
```

本番環境でアプリケーションをデバッグする場合は、コマンド ラインからシステム プロパティを定義できます。

デバッグの有効化:

```
java -DDEBUG_ON=com.kapowtech.robosuite.api.java.rql.io
```

複数のパッケージからデバッグする場合は、パッケージ名をコンマで区切ります。パッケージ名の代わりに、引数 `ALL` を指定して、すべてのパッケージからのデバッグを出力できます。

リポジトリ API

リポジトリ API を使用すると、Management Console のリポジトリのクエリを実行でき、プロジェクト、ロボット、およびロボットを呼び出すために必要な入力のリストを取得します。また、ロボット、タイプ、およびリソースファイルをプログラムで展開することもできます。

依存関係

リポジトリ API を使用するには、プロジェクトの `classpath` への Kofax RPA インストール フォルダにある `API/robosuite-java-api/lib` フォルダからすべての .jar ファイルを追加します。

Java 8 以降を使用します。

リポジトリ クライアント

リポジトリとの通信は、`RepositoryClient` を通じて `com.kapowtech.robosuite.api.java.repository.engine` で行われます。

RepositoryClient を作成:

```
public static void main(String[] args) {

    String username = "admin";
    String password = "admin";
    try
    {
        RepositoryClient client = RepositoryClientFactory.
                createRepositoryClient("http://localhost:50080/",
                        username, password);
        Project[] projects = client.getProjects();
        for (Project project : projects) {
            System.out.println(project.getName());
        }
    }
    catch(
    RepositoryClientException e)

    {
        e.printStackTrace();
    }
}
```

ここでは、`RepositoryClient` がユーザー名とパスワードを使用して `http://localhost:50080/` で Management Console のリポジトリに接続するように設定されています。

一度 `RepositoryClient` が作成されると、 `getProjects()` メソッドは、プロジェクトのリストのリポジトリを照会するために使用されます。`RepositoryClient` メソッドのいずれかを呼び出すときに エラーが発生した場合、`RepositoryClientException` がスローされることに注意してください。

`RepositoryClient` には次のメソッドがあります。

RepositoryClient のメソッド:

メソッド署名	説明
`void deleteResource(String projectName, String resourceName, boolean silent)`	プロジェクトからリソースを削除します。silent が true の場合、リソースが存在しなくてもエラーは生成されません。`resourceName` 引数はリソースのフル パスを使用します。
`void deleteRobot(String projectName, String robotName, boolean silent)`	プロジェクトからロボットを削除します。`robotName` 引数はロボットのフル パスを使用します。
`void deleteSnippet(String projectName, String snippetName, boolean silent)`	プロジェクトからスニペットを削除します。`snippetName` 引数はスニペットのフル パスを使用します。
`void deleteType(String projectName, String modelName, boolean silent)`	プロジェクトからタイプを削除します。`modelName` 引数は、タイプのフルパスを使用します。
`void deployLibrary(String projectName, EmbeddedFileBasedRobotLibrary library, boolean failIfExists)`	ライブラリをサーバーにデプロイします。ロボット、タイプ、リソースは、`failIfExists` が true でない場合に上書きされます。
`void deployResource(String projectName, String resourceName, byte[] resourceBytes, boolean failIfExists)`	リソースをプロジェクトにデプロイします。指定された名前のリソースがすでに存在する場合、`failIfExists` を false に設定することで上書きできます。`resourceName` 引数はリソースのフル パスを使用します。
`void deployRobot(String projectName, String robotName, byte[] robotBytes, boolean failIfExists)`	ロボットをプロジェクトにデプロイします。指定された名前のロボットがすでに存在する場合、`failIfExists` を false に設定することで上書きできます。`robotName` 引数はロボットのフル パスを使用します。
`void deploySnippet(String projectName, String snippetName, byte[] snippetBytes, boolean failIfExists)`	スニペットをプロジェクトにデプロイします。指定された名前のスニペットがすでに存在する場合、`failIfExists` を false に設定することで上書きできます。`snippetName` 引数はスニペットのフル パスを使用します。
`void deployType(String projectName, String typeName, byte[] typeBytes, boolean failIfExists)`	タイプをプロジェクトにデプロイします。指定された名前のタイプがすでに存在する場合、`failIfExists` を false に設定することで上書きできます。`typeName` 引数は、タイプのフルパスを使用します。
`Project[] getProjects()`	このリポジトリに存在するプロジェクトを返します。
`Cluster[] getRoboServerClusters()`	リポジトリを実行中の Management Console に登録されているクラスタおよびオンライン (有効な) RoboServer のリストを返します。
`Cluster[] getRoboServerClusters(boolean onlineRoboServer)`	Management Console に登録されているクラスタと RoboServer のリストを返します。`onlineRoboServer` フラグを使用して、クラスタのリストにオンラインの RoboServer を含めるかどうか、またはすべての RoboServer を含めるかどうかをマークします。
`Cluster addRoboServer(String clusterName, int portNumber, String host)`	新しい RoboServer をクラスタに追加します。
`Robot[] getRobotsInProject(String projectName)`	プロジェクトで使用可能なベーシック エンジン ロボットのフルパスを返します。

メソッド署名	説明
`RobotSignature getRobotSignature(String projectName, String robotName)`	ロボットのフルパス、このロボットの実行に必要な入力変数、およびロボットが返すまたは保存する可能性のあるタイプのリストとともに、ロボットの署名を返します。
`RepositoryFolder getProjectInventory(String projectName)`	リポジトリからフォルダとファイルのツリー全体を返します。
`RepositoryFolder getFolderInventory(String projectName, String folderPath)`	指定したプロジェクトのサブ フォルダのフォルダとファイルをリポジトリから返します。
`RepositoryFolder getFileInventory(String projectName, String folderPath, String fileName, RepositoryFile.Type fileType)`	Management console からファイルと参照ファイルを取得します。ファイル インベントリは、参照を取得するために `RepositoryFolder` でラップされることに注意してください。
`void deleteFile(RepositoryFile file)`	指定されたファイルをリポジトリから削除します。
`Date getCurrentDate()`	Management Console の現在の日付と時刻を返します。
`byte[] getBytes(RepositoryFile file)`	リポジトリ内の指定されたファイルのサイズをバイト単位で返します。
`void updateFile(RepositoryFile file, byte[] bytes)`	リポジトリ内の指定されたファイルを新しいバイトで更新します。
`void moveFile (RepositoryFile sourceFile, String destFolderPath)`	指定されたファイルをリポジトリから `destFolderPath` で指定されているフォルダに移動します。
`void renameRobot(RepositoryFile robotFile, String newName)`	指定したロボット ファイルの名前を変更します。
`void deleteFolder(String projectName, String folderPath)`	リポジトリ内の指定されたフォルダを削除します。
`void deleteRoboServer(String clusterName, RoboServer roboServer)`	RoboServer を削除します。
`Map<String, String> getInfo()`	Management Console およびリポジトリ API についての情報を返します このメソッドは、次のマッピングを返します: • 「アプリケーション」はメジャー バージョン、マイナー バージョン、およびドット バージョンを含む Management Console のバージョンです。たとえば、`11.5.0` • 「リポジトリ」は、リポジトリ API で使用される最新の DTD の ID です。たとえば、`//Kapow Technologies//DTD Repository 1.5//EN` • 「Rql」は、次のような Robot Query Language API で使用される最新の DTD の ID です。`//Kapow Technologies//DTD RoboSuite Robot Query Language 1.13//EN`
`pingRepository()`	リポジトリ サーバーに対して ping を実行すると、成功した場合は `null` を返され、それ以外の場合はエラー文字列を返されます。

メソッド署名	説明
`deployConnector(String projectName, String connectorName, byte[] connectorBytes, boolean failIfExists, AdditionalInfo additionalInfo)`	リポジトリにコネクタを展開します。
`deleteConnector(String projectName, String connectorName, boolean silent, AdditionalInfo additionalInfo)`	リポジトリからコネクタを削除します。
`getRobotsByTag(String projectName, String tag)`	指定したタグを持つロボットを取得します。

フル パスはプロジェクト フォルダからの相対パスです。

プロキシ サーバーは、`RepositoryClient` で指定されているフォルダに移動します。認証なしの標準 HTTP プロキシ サーバーがサポートされています。認証付きの NTLM プロキシ サーバーもサポートされています。

詳細については `RepositoryClient` JavaDoc を確認してください。

リポジトリ クライアントを使用したデプロイ

次の例は、`RepositoryClient` を使用して、ローカル ファイル システムからロボットとタイプを展開する方法を示しています。

RepositoryClient を使用したデプロイ:

```
String user = "test";
 String password = "test1234";
 RepositoryClient client = new RepositoryClient("http://localhost:50080", user,
     password);
 try {
     FileInputStream robotStream = new FileInputStream
      ("c:\\MyRobots\\Library\\Test.robot");
     FileInputStream typeStream = new FileInputStream
      ("c:\\MyRobots\\Library\\Test.type");

     // Use the Kapow Java APIs StreamUtil to convert InputStream to byte[].
     // For production we recommend IOUtils.toByteArray(InputStream i)
        in the commons-io library from apache.
     byte[] robotBytes = StreamUtil.readStream(robotStream).toByteArray();
     byte[] typeBytes = StreamUtil.readStream(typeStream).toByteArray();

     // we assume that no one has deleted the Default project
     client.deployRobot("Default project", "Test.robot", robotBytes, true);
     client.deployType("Default project", "Test.type", typeBytes, true);
 }
 catch (FileNotFoundException e) {
     System.out.println("Could not load file from disk " + e.getMessage());
 }
 catch (IOException e) {
     System.out.println("Could not read bytes from stream " + e.getMessage());
 }
 catch (FileAlreadyExistsException e) {
     // either the type or file already exist in the give project
     System.out.println(e.getMessage());
```

```
}
```

リポジトリ REST API

リポジトリ API は、実際にはデータを投稿できる Restful サービスと URL のグループです。

リポジトリから情報を取得するすべてのリポジトリ クライアント メソッドは、XML をリポジトリに送信し、リポジトリは XML で応答します。すべてのデプロイメソッドは、バイトをリポジトリ (URL でエンコードされた情報) にポストし、リポジトリは確認のために XML を返します。送受信される XML の形式は、www.kapowtech.com にある DTD によって管理されています。

以下は、すべての XML ベースのリクエストの例です。すべてのメッセージは、次の宣言で始まる必要があります。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE repository-request PUBLIC "-//Kapow Technologies//
DTD Repository 1.5//EN" "http://www.kapowtech.com/robosuite/
repository_1_5.dtd">
```

Management Console が `http://localhost:8080/ManagementConsole` でデプロイされたら、`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=xml` にリクエストが投稿される必要があります

スニペット

API 全体で多数の XML スニペットが使用されており、例では次のスニペットが使用されています。DTD を調べて、データの構造を理解することをお勧めします。

リクエストを送信するとき、多くの場合ファイルを記述する必要があります。同様に、応答にはファイルに関するデータが含まれます。次の表は、例で短縮されているスニペットを示しています。Design Studio と Management Console.のプログラム同期をアシストするためにコンストラクトが 1.5 DTD に追加されました。

スニペット名	コード
リポジトリ ファイル リクエスト	`<repository-file-request> <project-name>Default project</project-name> <name>ExName</name> <type>snippet</type> <path>subfolder</path> <last-modified>2019-02-01 19:26:12.321</last-modified> <last-modified-by>username</last-modified-by> <checksum>a342ddaf</checksum> </repository-file-request>`
リポジトリ ファイル	`<repository-file><name>filename</name> <type>ROBOT</name><last-modified>2019-02-01 19:26:12.321</last-modified><last-modified-by>username</last-modified-by><checksum>a342ddaf</checksum><dependencies><dependency><name>exsnippet</name><type>snippet</type></dependency> </dependencies></repository-file>`

REST 操作

メソッド	リクエスト例	応答例
delete-file (robot)	<repository-request> <delete-file file-type="robot" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>	<repository-response><delete-successful/></repository-response>
delete-file (type)	<repository-request> <delete-file file-type="type" silent="false"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>	<repository-response><error type="file-not-found">Could not find a Type named InputA.type in project 'Default project'</error></repository-response>
delete-file (snippet)	<repository-request> <delete-file file-type="snippet" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>	<repository-response><delete-successful/></repository-response>
delete-file (resource)	<repository-request> <delete-file file-type="resource" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>	<repository-response><delete-successful/></repository-response>
get-projects	<repository-request> <get-projects/> </repository-request>	<repository-response><project-list><project-name>Default project</project-name></project-list></repository-response>
get-robots-in-project	<repository-request> <get-robots-in-project> <project-name>Default project</project-name> </get-robots-in-project> </repository-request>	<repository-response><robot-list><robot><robot-name>DoNothing.robot</robot-name><version>10.7</version><last-modified>2019-10-11 18:24:12.648</last-modified></robot></robot-list></repository-response>
get-robot-signature	<repository-request> <get-robot-signature> <project-name>Default project</project-name> <robot-name>DoNothing.robot</robot-	<repository-response><robot-signature><robot-name>DoNothing.robot</robot-name><version>10.7</version><last-modified>2019-10-11 18:24:12.648</last-modified><input-object-list><input-object><variab

メソッド	リクエスト例	応答例
	name> </get-robot-signature> </repository-request>	le-name>InputA</variable-nam e><type-name>InputA</type-nam e><input-attribute-list><inp ut-attribute><attribute-nam e>aString</attribute-name><att ribute-type>Short Text</att ribute-type></input-attrib ute><input-attribute><attrib ute-name>anInt</attribute-nam e><attribute-type>Integer</att ribute-type></input-attribute ><input-attribute><attribute -name>aNumber</attribute-nam e><attribute-type>Number</att ribute-type></input-attribute ><input-attribute><attribute -name>aSession</attribute-nam e><attribute-type>Session</ attribute-type></input-attrib ute><input-attribute><attrib ute-name>aBoolean</attribute -name><attribute-type>Boolea n</attribute-type></input-att ribute><input-attribute><att ribute-name>aDate</attribute -name><attribute-type>Date</ attribute-type></input-attrib ute><input-attribute><attrib ute-name>aCharacter</attrib ute-name><attribute-type>Cha racter</attribute-type></inp ut-attribute><input-attribute ><attribute-name>anImage</att ribute-name><attribute-typ e>Image</attribute-type></inp ut-attribute></input-attrib ute-list></input-object><inp ut-object><variable-name>Inp utB</variable-name><type-nam e>InputB</type-name><input-att ribute-list><input-attribute required="true"><attribute -name>aString</attribute-nam e><attribute-type>Short Tex t</attribute-type></input-att ribute><input-attribute requir ed="true"><attribute-name>anI nt</attribute-name><attribute -type>Integer</attribute-typ e></input-attribute><input-att ribute required="true"><attrib ute-name>aNumber</attribute- name><attribute-type>Number</ attribute-type></input-attrib ute><input-attribute requir

メソッド	リクエスト例	応答例
		ed="true"><attribute-name>aSession</attribute-name><attribute-type>Session</attribute-type></input-attribute><input-attribute required="true"><attribute-name>aBoolean</attribute-name><attribute-type>Boolean</attribute-type></input-attribute><input-attribute required="true"><attribute-name>aDate</attribute-name><attribute-type>Date</attribute-type></input-attribute><input-attribute required="true"><attribute-name>aCharacter</attribute-name><attribute-type>Character</attribute-type></input-attribute><input-attribute required="true"><attribute-name>anImage</attribute-name><attribute-type>Image</attribute-type></input-attribute></input-attribute-list></input-object></input-object-list><returned-type-list><returned-type><type-name>OutputA</type-name><returned-attribute-list><returned-attribute><attribute-name>aString</attribute-name><attribute-type>Short Text</attribute-type></returned-attribute></returned-attribute-list></returned-type></returned-type-list><stored-type-list/></robot-signature></repository-response>
get-clusters	<repository-request> <get-clusters/> </repository-request>	<repository-response><clusters><cluster name="Cluster 1" ssl="false"><roboserver host="localhost" port="50000"/></cluster></clusters></repository-response>
get-current-date	<repository-request> <get-current-date/> </repository-request>	<repository-response> <current-date>2019-02-01 19:26:12.321</current-date> </repository-response>
get-bytes	<repository-request> <get-bytes> <repository-file-request>EXAMPLE</repository-file-request> </get-bytes> </repository-request>	<repository-response> <file-content> <file-bytes></file-bytes> </file-content> </repository-response>

メソッド	リクエスト例	応答例
get-project-inventory	<repository-request> <get-project-inventory> <project-name>Default project</project-name> </get-project-inventory> </repository-request>	<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>
get-folder-inventory	<repository-request> <get-folder-inventory> <project-name>Default project</project-name> <path>subfolder</path> </get-folder-inventory> </repository-request>	<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>
get-file-inventory	<repository-request> <get-file-inventory> <project-name>Default project</project-name> <path>subfolder</path> <name>robotname</name> <type>robot</type> </get-file-inventory> </repository-request>	<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>
update-file	<repository-request> <update-file> <repository-file-request>...</repository-file-request> <file-bytes></update-file> </repository-request>	<repository-response> <update-successful/> </repository-response>
get-clusters	<repository-request> <get-clusters online-roboserver='true'/> </repository-request>	<repository-response> <clusters> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> </clusters> </repository-response>

メソッド	リクエスト例	応答例
add-roboserver	<repository-request> <add-roboserver> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> <roboserver host='localhost' port='50001' primary='true'/ > </add-roboserver> </ repository-request>	<repository-response> <clusters> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> <roboserver host='localhost' port='50001' primary='true'/> </cluster> </clusters> </repository-response>
delete-roboserver	<repository-request> <add-roboserver> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> <roboserver host='localhost' port='50001' primary='true'/> </cluster> <roboserver host='localhost' port='50001' primary='true'/ > </add-roboserver> </ repository-request>	<repository-response> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> </ repository-response>
delete-folder	<repository-request> <delete-folder> <project-name>Default project</project-name> <path>path/to/empty/folder</ path> </delete-folder> </ repository-request>	<repository-response> <delete-successful/> </repository-response>
move-file	<repository-request> <move-file> <repository-file-request>...</repository-file-request> <path>new/ destination/path</path> </ move-file> </repository-request>	<repository-response> <update-successful/> </repository-response>
Rename-robot	<repository-request> <rename-robot> <repository-file-request>...</repository-file-request> <file-name>newnameofrobot</file-name> </rename-robot> </ repository-request>	<repository-response> <update-successful/> </repository-response>

ロボット、タイプ、スニペット、およびリソース名は、フル パスとして指定する必要があります。フル パスはプロジェクト フォルダからの相対パスです。

展開は、raw byte をポスト (オクテット ストリームがポスト本体として送信される) を次の URL に送信することによって行われます。以下は、リポジトリが `http://localhost:8080/ManagementConsole` でデプロイされている例です。

展開操作のメソッド:

操作	URL
deploy robot	http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployRobot&projectName=Default project&fileName=DoNothing.robot&failIfExists=true
deploy type	http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployType&projectName=Default project&fileName=InputA.type&failIfExists=true
deploy Snippet	http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deploySnippet&projectName=Default project&fileName=A.snippet&failIfExists=true
deploy resource	http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployResource&projectName=Default project&fileName=resource.txt&failIfExists=true
deploy library	http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployLibrary&projectName=Default project&fileName=NA&failIfExists=true

認証が Management Console で有効になっている場合、URL `http://localhost:8080/ManagementConsole/secure/RepositoryAPI` は basic 認証によって保護されています。これにより、次の方法で資格情報を URL に含めることができます。 `http://username:password@localhost:8080/ManagementConsole/secure/RepositoryAPI`。

Management Console API:

Management Console では、ロボットの実行をキューに入れることができます。ロボットを RoboServer で直接実行するのではなく、ロボットを Management Console のキューに配置できます。実行 ID の設定、データベース接続の定義、最大実行時間の設定、API 例外時のロボットの強制停止などの一部の機能は、ロボットの実行をキューに入れたときは制御できないことに注意してください。キューイングの利点は次のとおりです。

- 特定のリソース (デバイスなど) が利用できない場合でも、ロボットはキューに登録されます。
- 複数のバージョンの RoboServer が利用可能な場合、ロボットは正しい RoboServer にルーティングされます。
- アプリケーションの構築中にクラスタを管理する必要がなくなります。クラスタ設定は、Management Console でプロジェクト レベルで管理されます。

Java API の設定

QueuedRequests 用に Java API を設定するには、次の手順を実行します。

1. コードをコンパイルします。
2. コードの実行時に、.jar ファイルの場所を クラスパス に追加します。
3. キューを作成します。
   以下の詳細情報と例を参照してください。

ロボットの実行をキューに入れる

`QueuedRequest` クラスを使用して、Management Console で実行されるロボットをキューに入れます。詳細については、『Kofax RPA Java API Documentation』(Kofax RPA Java API ドキュメント) を参照してください。

> コードは大文字と小文字が区別されます。

JAVA API の例:

```
RepositoryHttpClientProvider repository = new RepositoryRobotLibrary("http://
localhost:50080/", "Default project", 60000, "admin", "admin");
QueuedRequest request = new QueuedRequest("myfolder/myrobot.robot", "Default project",
 repository);
request.setPriority(QueuedRequest.Priority.HIGH);
RQLResult result = request.execute();
```

コンストラクタ

コンストラクタ	説明
`QueuedRequest(String robotURL, String projectName, RepositoryHttpClientProvider httpClientProvider)`	指定された口ボットを使用して新しい `QueuedRequest` を作成します。 有効なエントリは、口ボット URL、プロジェクト名、およびリポジトリを指定する 3 つのパラメータで構成されます。 プロパティはリクエストを拡張します。プロパティ パラメータについては、次の表を参照してください。
`QueuedRequest(String robotURL, String projectName, RepositoryHttpClientProvider httpClientProvider, Priority priority, long timeout)`	指定された口ボット URL、優先度、およびタイムアウト時間を使用して、新しい `QueuedRequest` を作成します。

プロパティ

プロパティ	説明
`getPollingIntervalMillis()` / `setPollingIntervalMillis(int pollingIntervalMillis)`	口ボットの実行ステータスを更新します。また、戻り値と API 例外エラーをポーリングによって取得します。この関数は、ポーリングの間隔をミリ秒単位で設定または取得します。間隔が短いと、Management Console の負荷が高くなる可能性がありますが、このリクエストの更新レートが高くなります。デフォルト値は `1000` (1 秒) です。
`getRobotURL()` / `setRobotURL(String robotName)`	実行する口ボットの名前。口ボットがフォルダにある場合は、パスも含まれます。 > このプロパティに含まれる URL は、実際の URL ではありません。この名前は下位互換性のために付けられました。

プロパティ	説明
`getPriority()` / `setPriority(Priority priority)`	キュー内のリクエストの優先度: `MINIMUM`、`LOW`、`MEDIUM`、`HIGH`、または `MAXIMUM`。デフォルト値は `MEDIUM` です。
`setStopRobotOnApiException(boolean stopOnError)`	設定すると、最初の API 例外がクライアントに返された後、ロボットが RoboServer 上で停止します。デフォルト値は `true` です。
`getTimeout()` / `setTimeout(long timeout)`	ロボットが RoboServer の実行を待機してキューにとどまる最大時間 (秒単位)。デフォルト値は `600` (10 分) です。

メソッド

メソッド	説明
`createInputVariable(String name)`	指定された名前で新しい入力オブジェクトを作成し、オブジェクトの構築に使用できる RQLObjectBuilder を返します。
`createInputVariable(String name, RQLObject rqlObject)`	指定された名前の新しい入力オブジェクトを `rqlObject` から作成します。これは、ある口ボットからの応答を別の口ボットへの入力として使用する必要がある場合に役立ちます。
`createOAuthInputVariable(String name, String userName, String applicationName)`	指定されたユーザー名とアプリケーション名に基づいて、Management Console で OAuth の詳細を検索する OAUTH 入力タイプの入力変数を作成します。Management Console で、アプリケーションと OAuth ユーザーが、実行を要求する口ボットと同じプロジェクトにリンクされていることを確認してください。
`execute()`	口ボットを実行のためにキューに配置し、実行の終了を待機します。
`execute(RobotResponseHandler handler)`	口ボットを実行のためにキューに配置し、リクエストの処理のさまざまなポイントでハンドラを呼び出します。

第 2 章

.NET プログラマー ガイド

この章では、Kofax RPA レガシー .NET API を使用して、ロボットを実行する方法について説明します。このガイドは、簡単なロボットの作成方法に関する知識があり、C# プログラミング言語に精通していることを前提としています。

特定の .NET class に関する情報は、オフライン ドキュメント フォルダにあるコンパイル済みヘルプ `robosuite-dotnet-api.chm` で見つけることができます。詳細については、『Kofax RPA インストール ガイド』を参照してください。

.NET の基本

.NET API を使用することで、任意の .NET ベースのアプリケーションを RPA インスタンスのクライアントにすることができます。.NET API は .NET Standard 2.0 をターゲットにしており、.NET (Core) 2.0 ~ 6.0、または .NET Framework 4.7.2 以降でアプリケーションを構築します。

データベースにデータを保存するロボットを実行することに加えて、ロボットがクライアント アプリケーションに直接データを返すようにすることもできます。ここではいくつかの例を示します:

- 複数のロボットを使用して、複数のソースからの結果をリアルタイムで集約する検索を実行します。
- アプリケーションのバックエンドのイベントに応じてロボットを実行します。たとえば、新しいユーザーがサインアップしたときにロボットを実行して、バックエンドに直接統合されていない Web ベースのシステムにアカウントを作成します。

このガイドでは、コア class とそれらを使用してロボットを実行する方法を紹介します。また、ロボットに入力を提供し、RoboServer でロボットでの実行を制御する方法についても説明します。

.NET API は .dll ファイルで、Kofax RPA インストール フォルダ (インストール ガイド の 「Kofax RPA の重要なフォルダ」トピックを参照) の `/API/legacy/robosuite-dotnet-api/lib/robosuite-dotnet-api.dll` にあります。このガイドのすべての例は、`API/legacy/robosuite-dotnet-api/examples` にあります。`Newtonsoft.JSon.dll` は .NET API ファイルに含まれる必須のサードパーティ ライブラリです。

最初の例

以下は、`NewsMagazine.robot` という名前のロボットを実行するために必要なコードで、デフォルトのプロジェクトの Tutorials フォルダにあります。ロボットは Return Value ステップ アクションを使用して結果を出力します。これにより、API を使用してプログラムで出力を簡単に処理できます。他のロボット (通常は Management Console によるスケジュールで実行) は、データベース データ登録ステップアクションを使用してデータを直接データベースに保存します。この場合、ロボットによって収集されたデータは API クライアントに返されません。

次の例では、NewsMagazine ロボットが実行され、出力がプログラムで処理されます。

入力なしでロボットを実行します:

```
using System;
 using System.Collections.Generic;
 using System.Text;
 using Com.KapowTech.RoboSuite.Api;
 using Com.KapowTech.RoboSuite.Api.Repository.Construct;
 using Com.KapowTech.RoboSuite.Api.Construct;

 namespace Examples
 {
   class Program
   {
       static void Main(string[] args)
       {
           var server = new RoboServer("localhost", 50000);
           var ssl = false;
           var cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);

           Request.RegisterCluster(cluster); // you can only register a cluster
              once per application

           var request = new Request("Library:/Tutorials/NewsMagazine.robot");
           request.RobotLibrary = new DefaultRobotLibrary();
           RqlResult result = request.Execute("MyCluster");

           foreach (RqlObject value in result.GetOutputObjectsByName("Post")) {
               var title = value["title"];
               var preview = value["preview"];
               Console.WriteLine(title + ", " + preview);
           }
           Console.ReadKey();
       }
   }
 }
```

次の表に、関係するクラスとその責任を示します。

`RoboServer`	これは、ロボットを実行できる RoboServer を識別する単純な値オブジェクトです。各 RoboServer を Management Console でアクティブにして、使用する前に KCU を割り当てる必要があります。
`Cluster`	クラスタとは、単一の論理ユニットとして機能する RoboServer のグループです。
`Request`	この class は、ロボット リクエストを作成するために使用されます。リクエストを実行する前に、リクエスト class のあるクラスタを登録する必要があります。
`DefaultRobotLibrary`	ロボット ライブラリがリクエストで識別されたロボットを見つける RoboServer を指示します。後の例では、さまざまなロボット ライブラリの種類と、それらをいつどのように使用するかについて説明します。
`RQLResult`	これには、ロボットの実行結果が含まれます。結果には、値の応答、ログ、およびサーバー メッセージが含まれます。
`RQLObject`	戻り値アクションを使用してロボットから返される各値には、`RQLObject` としてアクセスできます。

最初の行では、API に RoboServer が localhost ポート 50000 で実行されていることが示されています。

```
var server = new RoboServer("localhost", 50000);
```

次の行は、単一の RoboServer のあるクラスタを定義します。クラスタは Request class に登録されているため、このクラスタでリクエストを実行できます。各クラスタは、アプリケーションごとに1回のみ登録できます。これは、アプリケーションの初期化の間に行われます。

クラスタの登録:

```
var ssl = false;
 var cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);
 Request.RegisterCluster(cluster);
```

次のコードは、`NewsMagazine.robot` という名前の口ボットを実行するリクエストを作成し、これは `Library:/Tutorials Library:/` に置かれ、リクエスト用に設定された口ボットライブラリーの参照となります。ここでは、`DefaultRobotLibrary` が使用されます。サーバーのローカル ファイル システムで口ボットを探すのに RoboServer を指示します。口ボット ライブラリの使用方法の詳細は 口ボット ライブラリ を参照してください。

```
var request = new Request("Library:/Tutorials/NewsMagazine.robot");
 request.RobotLibrary = new DefaultRobotLibrary();
```

次の行は、`MyCluster` という名前のクラスタ (以前に登録されたクラスタ) で口ボットを実行し、口ボットが完了すると結果を返します。口ボットの実行中にエラーが発生した場合、ここで例外がスローされます。

```
RqlResult result = request.Execute("MyCluster");
```

最後に、抽出した値を処理します。まず、Post という名前のタイプのすべての抽出値を取得し、それらを反復処理します。各 `RQLObject` の場合では、Post タイプの属性にアクセスし、結果を出力します。属性とマッピングについては、後のセクションで説明します。

```
foreach (RqlObject value in result.GetOutputObjectsByName("Post")) {
     var title = value["title"];
     var preview = value["preview"];
     Console.WriteLine(title + ", " + preview);
```

口ボット入力

API を介して実行されるほとんどの口ボットは、検索キーワードやログイン資格情報などの入力を通じてパラメーター化されます。構成口ボットへの入力は、RoboServer のリクエストの一部で、リクエストの `createInputVariable` メソッドを使って提供されます。

暗黙的な **RQLObjectBuilder** を使用した入力

```
var request = new Request("Library:/Tutorials/Input.robot");
 request.CreateInputVariable("userLogin").SetAttributeEntry
  ("username", "scott").SetAttributeEntry("password", "tiger");
```

上記のコードでは、`Request` を作成し、`CreateInputVariable` を使用し、`userLogin` という名前の入力変数を作成します。次に `setAttribute` を使用して、入力変数のユーザー名とパスワードの属性を設定します。

上記の例は一般的な略記法ですが、`RqlObjectBuilder` を使用してより詳細に表現することもできます:

```
var request = new Request("Library:/NewsMagazine.robot");
 RqlObjectBuilder userLogin = request.CreateInputVariable("userLogin");
 userLogin.SetAttributeEntry("username", "scott");
 userLogin.SetAttributeEntry("password", "tiger");
```

2 つの例は同じです。1 つ目は、匿名の `RqlObjectBuilder` でカスケード メソッド呼び出しを使用するため、より短くなります。

RoboServer がこのリクエストを受信すると、次のことが発生します:

- RoboServer が `Input.robot` (リクエスト用に設定された `RobotLibrary` から) をロードします。
- RoboServer は ロボットに `userLogin` という名前の変数があり、この変数が入力としてマークされていることを確認します。
- ここで RoboServer は `setAttribute` を使って設定した属性を検証し、`userLogin`という名前の入力変数を作成します。その結果、タイプにはユーザー名とパスワードという名前の属性が必要でとなり、両方ともテキストベースの属性である必要があります (次のセクションで API と Design Studio 属性間のマッピングを説明しています) 。
- すべての入力変数に互換性がある場合、RoboServer は ロボットの実行を開始します。

ロボットが複数の入力変数を必要とする場合、それらをすべて作成して ロボットを実行する必要があります。設定する必要があるのは必須属性のみで、API を使用して設定しない必須属性はすべて null 値になります。Facebook と Twitter への ログインを必要とする ロボットがある場合、次のように入力を定義できます。

```
Request request = new Request("Library:/Input.robot");
 request.CreateInputVariable("facebook").SetAttributeEntry
   ("username", "scott").SetAttributeEntry("password", "facebook123");
 request.CreateInputVariable("twitter").SetAttributeEntry
   ("username", "scott").SetAttributeEntry("password", "twitter123");
```

属性タイプ

Design Studio で新しいタイプを定義するとき、各属性の属性タイプを選択します。一部の属性には、ショート テキスト、ロング テキスト、パスワード、HTML、XML などのテキストを含めることができます。また、ロボット内で使用する場合、これらの属性に保存されるテキストに要件がある場合があります。XML 属性にテキストを保存する場合、テキストは有効な XML ドキュメントである必要があります。この検証は、タイプが ロボット内で使用される場合に発生しますが、API はタイプについて何も認識しないため、同じ方法で属性値を検証しません。そのため、API には 8 つの属性タイプがあり、Design Studio には 19 種類のタイプがあります。この表は、API と Design Studio 属性タイプのマッピングを示しています。

API へ Design Studio マッピング

API 属性タイプ	**RoboServer** 属性タイプ
テキスト	ショート テキスト、ロング テキスト、パスワード、HTML、XML、プロパティ、言語、国、通貨、キーの再検索
整数	整数
ブール値	ブール値
数	数
文字	文字
日付	日付
セッション	セッション
バイナリ	バイナリ、画像、PDF

API 属性タイプは、次の方法で .NET にマップされます。

属性の **.Net** タイプ

API 属性タイプ	**Java Class**
テキスト	`System.String`（ストリング）
整数	`System.Int64`
ブール値	`System.Boolean`（ブール）
数	`System.Double`（ダブル）
文字	`System.Char`（char）
日付	`System.DateTime`
セッション	`Com.Kapowtech.Robosuite.Api.Construct.Session`
バイナリ	`Com.Kapowtech.Robosuite.Api.Construct.Binary`

`RqlObjectBuildersetAttribute` メソッドはオーバーロードされるため、適切な .NET class が引数として使用されている場合は、API を介して属性を設定するときに属性タイプを明示的に指定する必要はありません。オブジェクトの属性に可能な Design Studio 属性タイプをすべて設定する方法を次の例に示します。

`setAttribute` の推奨使用法:

```
RqlObjectBuilder inputBuilder = request.CreateInputVariable("AllTypes");
 inputBuilder.SetAttributeEntry("anInt", 42L);
 inputBuilder.SetAttributeEntry("aNumber", 12.34d);
 inputBuilder.SetAttributeEntry("aBoolean", true);
 inputBuilder.SetAttributeEntry("aCharacter", 'c');
 inputBuilder.SetAttributeEntry("aShortText", "some text");
 inputBuilder.SetAttributeEntry("aLongText", "a longer text");
 inputBuilder.SetAttributeEntry("aPassword", "secret");
 inputBuilder.SetAttributeEntry("aHTML", "<html>text</html>");
 inputBuilder.SetAttributeEntry("anXML", "<tag>text</tag>");
 inputBuilder.SetAttributeEntry("aDate", DateTime.Now);
 inputBuilder.SetAttributeEntry("aBinary", (Binary) null);
 inputBuilder.SetAttributeEntry("aPDF", (Binary)null);
 inputBuilder.SetAttributeEntry("anImage", (Binary)null);
 inputBuilder.SetAttributeEntry("aProperties", "name=value\nname2=value2");
 inputBuilder.SetAttributeEntry("aSession", (Session)null);
 inputBuilder.SetAttributeEntry("aCurrency", "USD");
 inputBuilder.SetAttributeEntry("aCountry", "US");
 inputBuilder.SetAttributeEntry("aLanguage", "en");
 inputBuilder.SetAttributeEntry("aRefindKey", "Never use as input");
```

上記の例では、C＃コンパイラーは、どのオーバーロード バージョンの `SetAttributeEntry` メソッドを呼び出すかを決定できないため、null 値をキャストする必要があります。ただし、未構成の属性は自動的に null になるため、null を明示的に設定する必要はありません。

API を使用して入力を作成するときに、 `Attribute` と `AttributeType` を明示的に指定することができます。このアプローチは推奨されませんが、まれに必要になることがあり、次のようになります。

`setAttribute` の推奨されない使用法:

```
RqlObjectBuilder inputBuilder = request.CreateInputVariable("alltypes");
 inputBuilder.SetAttributeEntry(new AttributeEntry("anInt", "42",
   AttributeEntryType.Integer));
```

```
inputBuilder.SetAttributeEntry(new AttributeEntry("aNumber", "12.34",
  AttributeEntryType.Number));
inputBuilder.SetAttributeEntry(new AttributeEntry("aBoolean", "true",
  AttributeEntryType.Boolean));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCharacter", "c",
  AttributeEntryType.Character));
inputBuilder.SetAttributeEntry(new AttributeEntry("aShortText", "some text",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aLongText", "a longer text",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aPassword", "secret",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aHTML", "<html>text</html>",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("anXML", "<tag>text</tag>",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aDate",
  "2012-01-15 23:59:59.123", AttributeEntryType.Date));

inputBuilder.SetAttributeEntry(new AttributeEntry("aBinary", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("aPDF", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("anImage", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("aProperties",
  "name=value\nname2=value2", AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCurrency", "USD",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCountry", "US",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aLanguage", "en",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aRefindKey",
  "Never use this as input", AttributeEntryType.Text));
```

すべての属性値は、文字列の形式で提供する必要があります。文字列値は、指定された `AttributeEntryType` に基づいて適切な .NET オブジェクトに変換されます。Kofax RPA .NET API の上部で、他の汎用 API をビルドする場合にのみ役立ちます。

実行パラメータ

`CreateInputVariable` メソッドに加えて、リクエストには口ボットが RoboServer でどのように実行されるかを制御する多くのプロパティが含まれます。

リクエストに応じた実行制御メソッド

`ApiKey`	RFS、DTS、パスワード ストアなどのリソースへのアクセスを認証します。 省略した場合、接続で提供されたクレデンシャルが使用されます。
`MaxExecutionTime`	口ボットの実行時間を秒単位で制御します。この時間が経過すると、口ボットは RoboServer で停止します。口ボットが実行を開始するまでタイマーは開始されないため、口ボットが RoboServer で待機している場合、これは考慮されません。

`StopOnConnectionLost`	true (デフォルト) の場合、RoboServer がクライアント アプリケーションとの接続が失われたことを検出すると、ロボットは停止します。この値を false に設定し、これを処理するコードが記述されていない場合、アプリケーションは期待どおりに動作しない可能性があります。
`StopRobotOnApiException`	true (デフォルト) の場合、最初の API 例外が発生した後、ロボットは RoboServer で止まります。デフォルトでは、ロボットのほとんどのステップは、ステップの実行に失敗すると API 例外を発生させます。ステップの [エラー処理] タブでこの値を設定します。 false に設定すると、ロボットは API 例外に関係なく実行を続けます。ただし、アプリケーションが `IRobotResponseHandler` を使って結果をストリーミングしない場合、`Execute()` によって例外が引き続きスローされます。false に設定するときは注意してください。
`Username, Password`	RoboServer 資格情報を設定します。RoboServer は認証を要求するように設定できます。このオプションを有効にすると、クライアントは資格情報を提供するか、RoboServer がリクエストを拒否します。
`RobotLibrary`	`RobotLibrary` をリクエストに割り当てます。ロボット ライブラリがリクエストで識別されたロボットを見つける RoboServer を指示します。さまざまなライブラリ タイプとその使用法に関するその他の例については、ロボット ライブラリ を参照してください。
`ExecutionId`	このリクエスト用に `executionId` を設定することができます。指定しない場合は、RoboServer で自動的に生成されます。実行 ID はログ記録に使用され、ロボットをプログラムで停止するためにも必要です。ID は (経時的に) グローバルに一意である必要があります。2 つのロボットが同じ実行 ID を使用している場合、ログの一貫性が失われます。 このプロパティの設定は、ロボットが大規模なワークフローの一部であり、クライアント アプリケーションにすでに一意の識別子がある場合、ロボット ログをシステムの残りの部分と結合できるため便利です。
`setProject(String)`	このメソッドは、ロギングの目的でのみ使用されます。Management Console はこのフィールドを使用してログメッセージをプロジェクトにリンクします。そのため、ログ ビューはプロジェクトでフィルタリングできます。 アプリケーションが `RepositoryRobotLibrary` を使用していない場合、この値を設定してこのロボットが属するプロジェクト (存在する場合) の RoboServer ロギングシステムを知らせます。

ロボット ライブラリ

Design Studio で、ロボットはプロジェクトにグループ化されます。ファイル システムを見ると、これらのプロジェクトは `Library` という名前のフォルダを含む必要があるという唯一の制約を持つフォルダによって表されていることがわかります。

RoboServer の実行リクエストを作成するとき、ロボットの URL でロボットを識別します:

```
Request request = new Request("Library:/Input.robot");
```

ここで、`Library:/` はロボット ライブラリへのシンボリック リファレンスです。RoboServer はロボットを探す必要があります。`RobotLibrary` は、ビルダーで次の方法で指定されます。

```
request.setRobotLibrary(new DefaultRobotLibrary());
```

3 つの異なるロボット ライブラリの実装が利用可能です。選択するものは、展開環境によって異なります。

ロボット ライブラリ

ライブラリ タイプ	説明
`DefaultRobotLibrary`	このライブラリは現在のプロジェクト フォルダでロボットを探すために、RoboServer を設定します。このフォルダは、設定アプリケーションで定義されます。 RoboServer が複数ある場合は、すべての RoboServer でロボットを展開する必要があります。 このロボット ライブラリはキャッシュされないため、ロボットは実行のたびにディスクからリロードされます。このアプローチにより、ロボットは頻繁に変更される開発環境で使用できますが、本番環境には適していません。

ライブラリ タイプ	説明
EmbeddedFileBasedRobotLibrary	このライブラリは、RoboServer に送信される実行リクエストに埋め込まれています。このライブラリを作成するには、ロボットとそのすべての依存関係 (タイプ、スニペット、リソース) を含む .zip ファイルを作成する必要があります。これは、Design Studio の中で **[ツール] > [ロボット ライブラリ ファイルの生成]** メニューで実行できます。 ライブラリはリクエストごとに送信されるため、大規模なライブラリにはオーバーヘッドが追加されますが、ライブラリは RoboServer でキャッシュされ、最高のパフォーマンスを提供します。 1 つの強みは、ロボットとコードを単一のユニットとして展開できることです。これにより検証環境から本番環境へのクリーンな移行が可能になります。ただし、ロボットが頻繁に変更される場合は、頻繁に再デプロイする必要があります。 次のコードを使用して、リクエスト用の組み込みロボット ライブラリを設定できます。 `var request = new Request ("Library:/Tutorials/NewsMagazine. robot"); var stream = new FileStream ("c:\\embeddedLibrary.robotlib", FileMode.Open); request.RobotLibrary = new EmbeddedFileBasedRobotLibrary (stream);`

ライブラリ タイプ	説明
`RepositoryRobotLibrary`	これは最も柔軟な `RobotLibrary`です。 このライブラリは、ロボット ライブラリとして Management Console の組み込みリポジトリを使用します。このライブラリを使用すると、RoboServer はロボットとその依存関係を含むロボット ライブラリを送信する Management Console へ繋がります。 キャッシングは Management Console と RoboServer の内部で、ロボットごとに行われます。Management Console 内部で、生成されたライブラリはロボットとその依存関係に基づいてキャッシュされます。RoboServer で、キャッシュはタイムアウトに基づいているため、Management Console にそれぞれのキャッシュに問い合わせる必要はありません。さらに、RoboServer と Management Console の間で読み込むライブラリは、HTTP パブリック/プライベート キャッシュを使用して、帯域幅をさらに削減します。 `NewsMagazine.robot` が Management Console にアップロードされた場合、ロボットの実行時にリポジトリ ロボット ライブラリを使用できます。 `var request = new Request ("Library:/Tutorials/NewsMagazine. robot"); request.RobotLibrary = new RepositoryRobotLibrary ("http://localhost:50080", "Default Project", 60000);` このコマンドはローカル RoboServer からロボットをロードすることを Management Console に指示し、Management Console で確認する前に 1 分間キャッシュし、ロボットの新しいバージョン (そのタイプとスニペット) が利用可能かどうかを確認します。 さらに、`Library:/protocol` を介してロードされたリソースにより RoboServer は Management Console からリソースを直接要求します。

.NET Advanced

このセクションでは、出力ストリーミング、ロギング、SSL 構成、並列実行などの高度な API 機能について説明します。

負荷分散

`RequestExecutor` の内部では、エグゼキューターに RoboServer の配列が与えられます。Executor が構築されると、それぞれ RoboServer に接続しようとします。接続されると、それぞれ RoboServer に ping リクエストを送信し、サーバーの設定方法を検出します。

負荷分散エグゼキューター

```
RoboServer prod = new RoboServer("prod.kapow.local", 50000);
 RoboServer prod2 = new RoboServer("prod2.kapow.local", 50000);
```

```
Cluster cluster = new Cluster("Prod", new RoboServer[]{ prod, prod2}, false);
Request.RegisterCluster(cluster);
```

負荷は RoboServer 上の未使用の実行スロットの数に基づくクラスタ内にある各オンライン RoboServer に分散されます。次のリクエストは常に、使用可能なスロットが最も多い RoboServer に配信されます。使用可能な実行スロットの数は、最初の ping 応答を通じて取得され、エグゼキューターは、起動した各ロボットが完了するまで追跡します。RoboServer の実行スロットの数は、Management Console > 管理 > RoboServer セクションの [最大同時ロボット数] の設定によって決まります。

RoboServer がオフラインになると、ping リクエストに正常に応答するまでロボット実行リクエストを受信しません。

1 つのクライアント ルール

デフォルトでは、API 接続は 20 件の接続に制限されています。ただし、最高のパフォーマンスを確保するには、RoboServer の特定のクラスタを使用する API クライアントを 1 つだけにすることをお勧めします。同じ RoboServer に対してロボットを実行している JVM が多すぎる場合は、パフォーマンスが低下します。

以下は推奨されませんが、より大きなボリュームを処理する必要がある環境では、common.conf ファイルの `kapow.max.multiplexing.clients` システムプロパティを調整することにより、接続制限を設定できます。

データ ストリーミング

ロボットの実行結果をリアルタイムで表示する必要がある場合は、ロボットが実行を終了して `RqlResult` へアクセスするのを待つ代わりに、API を使用してすぐに抽出した値を返却できます。

API は、API がロボットから返された値を受け取るたびにコールバックを受け取る可能性を提供します。これは、`IRobotResponseHandler` インターフェイスを通じて行います。

AbstractFailFastRobotResponseHandler を使用した応答ストリーミング

```
using System;
 using Com.KapowTech.RoboSuite.Api;
 using Com.KapowTech.RoboSuite.Api.Repository.Construct;
 using Com.KapowTech.RoboSuite.Api.Construct;
 using System.IO;
 using Com.KapowTech.RoboSuite.Api.Engine.Hotstandby;

 namespace Examples
 {

     public class DataStreaming {

         public static void Main(String[] args)  {

             var server = new RoboServer("localhost", 50000);
             var cluster = new Cluster("MyCluster", new RoboServer[] { server },
                false);
             Request.RegisterCluster(cluster);

             var request = new Request("Library:/Tutorials/NewsMagazine.robot");
             IRobotResponseHandler handler = new SampleResponseHandler();
             request.Execute("MyCluster", handler);
         }
```

```
    }


    public class SampleResponseHandler : AbstractFailFastRobotResponseHandler
    {
        override public void HandleReturnedValue(RobotOutputObjectResponse
          response, IStoppable stoppable)
        {
            var title = response.OutputObject["title"];
            var preview = response.OutputObject["preview"];
            Console.WriteLine(title + ", " + preview);
        }
    }
}
```

上記の例では、リクエストの2番目の execute メソッドを使用しています。ロボットを実行するクラスタの名前に加えて `RobotResponseHandler` を想定しています。この例では、デフォルトのエラー処理を提供する `AbstractFailFastRobotResponseHandler` を拡張して `IRobotResponseHandler` を作成し、ロボットが返す値のみを処理します。

`handleReturnedValue` メソッドは、RoboServer からAPI が戻り値を受け取るたびに呼び出されます。この例で使用されている `AbstractFailFastRobotResponseHandler` は、非ストリーミング実行メソッドと同じ方法で例外をスローします。これは、ロボットによって生成された API 例外に応じて例外がスローされることを意味します。

`IRobotResponseHandler` には 3 つのカテゴリにグループ化できるいくつかのメソッドがあります。

ロボットのライフ サイクル イベント
実行の開始時と終了時など RoboServer 上でロボットの実行状態が変化したときに呼び出されるメソッド。

ロボット データ イベント
ロボットが API にデータまたはエラーを返すときに呼び出されるメソッド。

追加のエラー処理
RoboServer または API 内部でのエラーのために呼び出されるメソッド。

RobotResponseHandler - ロボットのライフ サイクル イベント

メソッド名	説明
`void requestSent(RoboServer roboServer, ExecuteRequest request)`	`RequestExecutor` がリクエストを実行するサーバーを見つけたら呼び出されます。
`void requestAccepted(String executionId)`	見つかった RoboServer がリクエストを受け、それをキューに入れるときにときに呼び出されます。
`void RobotStarted(IStoppable stoppable)`	RoboServer が口ボットの実行を開始するときに呼び出されます。これは通常、RoboServer の負荷が大きいか、あるいは複数の API クライアントで使用されていない場合、口ボットがキューに入れられた直後に発生します。

メソッド名	説明
`void robotDone(RobotDoneEvent reason)`	ロボットの実行が RoboServer で完了したときに呼び出されます。`RobotDoneEvent` は、エラーのために実行が正常に終了したか、停止したかを指定するために使用されます。

RobotResponseHandler - ロボット データ イベント

メソッド名	説明
`void HandleReturnedValue(RobotOutputObjectResponse response, IStoppable stoppable)`	ロボットが Return Value アクションを実行し、値がソケットを介して API に返されたときに呼び出されます。
`void HandleRobotError(RobotErrorResponse response, IStoppable stoppable)`	ロボットが API 例外を発生させたときに呼び出されます。通常の状況では、ロボットは最初のAPI例外の後に実行を停止します。この動作は `Request.StopRobotOnApiException = false` を使用してオーバーライドできます。この場合、このメソッドは複数回呼び出されます。これは、生成されたエラーに関係なく実行を継続するデータ ストリーミング ロボットが必要な場合に便利です。
`void HandleWriteLog(RobotMessageResponse response, IStoppable stoppable)`	ロボットがログの書き込みアクションを実行する場合に呼び出されます。これは、ロボットから追加のログ情報を提供するのに役立ちます。

`RobotResponseHandler` **-** 追加のエラー処理

メソッド名	説明
`void HandleServerError(ServerErrorResponse response, IStoppable stoppable)`	RoboServer がエラーを生成すると呼び出されます。たとえば、サーバーがビジー状態でリクエストを処理できない場合、または RoboServer 内部でエラーが発生した場合、ロボットの起動を防ぎます。
`void handleError(RQLException e, IStoppable stoppable)`	API 内でエラーが発生した場合に呼び出されます。クライアントが RoboServer への接続を失った場合、よく起こります。

メソッドの多くには `IStoppable` オブジェクトが含まれ、特定のエラーが発生したか、値が返された後に停止するのに使用できます。

これらのメソッドのいくつかは、`RQLException` をスローする機能を提供します。
ハンドラーを呼び出すスレッドは、`Request.Execute()` を呼び出すスレッドです。
つまり、スローされる例外は、コールスタックをオーバーロードする可能性があります。`handleReturnedValue`、`handleRobotError`、`handleWriteLog` に応じて例外がスローされた場合、`Stoppable.stop()` を呼び出す必要があります。呼び出しを行わない場合、ロボットは `Request.Execute()` への呼び出しが完了しても実行を続ける可能性があります。

データ ストリーミングは、次の使用例のいずれかで最もよく使用されます。

- 結果がリアルタイムでユーザーに表示される Ajax ベースの Web アプリケーション。データがストリーミングされない場合、ロボットの実行が完了するまで結果を表示できません。

- クライアントがロボットの実行中にすべてをメモリに保持することができない程大量のデータを返すロボット。
- 抽出された値がロボットの実行と並行して処理されるように最適化する必要があるプロセス。
- カスタム形式でデータベースにデータを保存するプロセス。
- API 例外のカスタム処理を無視または必要とするロボット (次の例を参照) 。

AbstractFailFastRobotResponseHandler を使用した応答とエラーの収集:

```
using System;
 using System.Collections;
 using System.Collections.Generic;
 using Com.KapowTech.RoboSuite.Api;
 using Com.KapowTech.RoboSuite.Api.Repository.Construct;
 using Com.KapowTech.RoboSuite.Api.Construct;
 using System.IO;
 using Com.KapowTech.RoboSuite.Api.Engine.Hotstandby.Interfaces;

 namespace Examples
 {
     public class DataStreaming
     {

         public static void Main(String[] args)
         {

             var server = new RoboServer("localhost", 50000);
             var cluster = new Cluster("MyCluster", new RoboServer[] { server },
               false);
             Request.RegisterCluster(cluster);

             var request = new Request("Library:/Tutorials/NewsMagazine.robot");
             request.StopRobotOnApiException = false;  // IMPORTANT!!

             ErrorCollectingRobotResponseHandler handler =
               new ErrorCollectingRobotResponseHandler();
             request.Execute("MyCluster", handler); // blocks until robot is
               done, or handler throws an exception

             Console.WriteLine("Extracted values:");
                 foreach (RobotOutputObjectResponse response in handler.
                  GetOutput())
             {
                 var title = response.OutputObject["title"];
                 var preview = response.OutputObject["preview"];
                 Console.WriteLine(title + ", " + preview);
             }

             Console.WriteLine("Errors:");
             foreach (RobotErrorResponse error in handler.GetErrors())
             {
                 Console.WriteLine(error.ErrorLocationCode + ", " + error.
                    ErrorMessage);
             }
         }
     }


     public class ErrorCollectingRobotResponseHandler :
       AbstractFailFastRobotResponseHandler {

         private IList<RobotErrorResponse> _errors =
           new List<RobotErrorResponse>();
```

```
        private IList<RobotOutputObjectResponse> _output =
          new List<RobotOutputObjectResponse>();

        override public void HandleReturnedValue(RobotOutputObjectResponse
          response, IStoppable stoppable)  {
            _output.Add(response);
        }


        override public void HandleRobotError(RobotErrorResponse response,
           IStoppable stoppable) {
            // do not call super as this will stop the robot
            _errors.Add(response);
        }

        public IList<RobotErrorResponse> GetErrors() {
            return _errors;
        }

        public IList<RobotOutputObjectResponse> GetOutput() {
            return _output;
        }
    }
}
```

上記の例は、返された値とエラーを収集する `IRobotResponseHandler` の使い方を示します。このタイプのハンドラーは、エラーが発生した場合でも口ボットが実行を継続する必要がある場合に役立ちます。これは、Web サイトが不安定で、タイムアウトする場合に役立ちます。口ボット エラー (API 例外) のみがハンドラーによって収集されることに注意してください。RoboServer への接続が失われた場合、`Request.Execute()` は引き続き `RQLException` をスローし、RoboServer は口ボットを停止します。

詳細については、`IRobotResponseHandler` ドキュメントを参照してください。

SSL

API は、`RQLService` を通じて RoboServer と通信し、これは特定のネットワーク ポートで API リクエストをリッスンする RoboServer コンポーネントです。RoboServer を開始すると、暗号化された SSL サービス、プレーン ソケット サービス、またはその両方 (2 つの異なるポートを使用) を使用するかどうかを指定します。クラスタ内のすべての RoboServer が同じ `RQLService` を実行している必要があります (ただし、ポートは異なる場合があります)。

RoboServer はポート 50043 で SSL `RQLService` を開始するものと見なします。

```
RoboServer -service ssl:50043
```

次のコードが使用されます。

```
RoboServer server = new RoboServer("localhost", 50043);
 boolean ssl = true;
 Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, ssl);
 Request.RegisterCluster(cluster);
```

クラスタを SSL クラスタとして作成し、それぞれ RoboServer が使用する SSL ポートを指定する必要があります。これで、RoboServer とAPI のすべての通信は暗号化されます。

SSL では、データの暗号化に加えて、リモート パーティの身元を確認することができます。このタイプの確認は、インターネット上で非常に重要です。多くの場合、API クライアントと RoboServer は同じ

ローカル ネットワーク上にあるため、互いの ID を確認する必要はほぼありませんが、必要になる場合もあるため API はこの機能をサポートしています。

サンプル で、コンパイルの仕方と、SSL が含まれた例を実行する方法を参照してください。

リポジトリ統合

Management Console では、このクラスタはスケジュールされたロボット、および REST サービスとして実行されるロボットの実行に使用される RoboServer のクラスタの指定も行います。API を使用すると、`RepositoryClient` を使って Management Console からクラスタ情報が取得できます。詳細については、`RepositoryClient` ドキュメントを参照してください。

リポジトリ統合

```
using System;
 using Com.KapowTech.RoboSuite.Api;
 using Com.KapowTech.RoboSuite.Api.Construct;
 using Com.KapowTech.RoboSuite.Api.Repository.Engine;

 namespace Examples
 {
     public class RepositoryIntegration
     {
         public static void Main(String[] args)
         {
             string userName = "admin";
             string password = "admin";
             RepositoryClient client = new RepositoryClient
               ("http://localhost:50080", userName, password);

             Request.RegisterCluster(client, "Production");
             var request = new Request("Library:/Tutorials/NewsMagazine.robot");
             var result = request.Execute("Production");
             Console.WriteLine(result.ToString());
         }
     }
 }
```

上記の例は、localhost ポート50080 にデプロイされた Management Console に接続する `RepositoryClient` の生成方法を示しています。

`RepositoryClient` の登録時に、Management Console に存在するクラスタの名前を指定します。その後、Management Console に照会を行ってこのクラスタに設定された RoboServer のリストを取得し、2 分ごとに Management Console 上でクラスタ設定が更新されているかどうかを確認します。

この統合により、Management Console ユーザー インターフェイスを使って、動的に変更できるクラスタを Management Console 上で作成することができます。API を使用する Management Console クラスタは 1 つである必要があり、OneClient ルールに違反するため、ロボットのスケジューリングには使用しないでください。

実行ログ

リクエストを実行すると、ロボットがエラーを生成した場合、execute メソッドは例外をスローします。他のタイプのエラーと警告は、`ExecutorLogger` のインターフェイスを通じて報告されます。前の例では、`ExecutionLogger` はロボットの実行時には提供されませんでした。これは `System.out` に書き込むデフォルトの実装です。

以下に、RoboServer のいずれか 1 つがオフラインになった場合の `ExecutorLogger` からの報告の例を示します。この例では、オンラインではない RoboServer のクラスタを設定します。

ExecutorLogger、オフライン サーバーの例:

```
RoboServer rs = new RoboServer("localhost", 50000);
 Cluster cluster = new Cluster("name", new RoboServer[]{rs}, false);
 Request.RegisterCluster(cluster);
```

この例を実行すると、次の内容がコンソールに書き込まれます。

ExecutorLogger、オフライン **RoboServer** コンソール出力:

```
RoboServer[Host=localhost, Port=50000]' went offline.
 Com.KapowTech.RoboSuite.Api.Engine.UnableToConnectException:..........
```

アプリケーションが `System.out` に直接書き込む必要がない場合、クラスタを登録する時に異なる `IExecutorLogger` の実装を提供できます:

DebugExecutorLogger を使用:

```
Request.RegisterCluster(cluster, new DebugExecutorLogger());
```

この例では、`System.out` にも書き込む `DebugExecutorLogger()` を使用します。ただし、API デバッグが有効になっている場合のみです。または、エラー メッセージの処理方法の制御に対する独自の `ExecutorLogger` の実装を提供できます。

内部処理

このセクションでは、クラスタを登録してリクエストを実行する際に内部で何が起こっているのかを説明します。

リクエストでクラスタを登録すると、裏で `RequestExecutor` が作成されます。この `RequestExecutor` は、クラスタ名をキーとしてマップに保存されます。リクエストが実行されると、提供されたクラスタ名を使用して、関連付けられた `RequestExecutor` を探し、リクエストを実行します。

通常の実行:

```
public static void Main(String[] args)
 {
     RoboServer server = new RoboServer("localhost", 50000);
     Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
     Request.RegisterCluster(cluster);

     var request = new Request("Library:/Tutorials/NewsMagazine.robot");
     request.RobotLibrary = new DefaultRobotLibrary();
     var result = request.Execute("MyCluster");
     Console.WriteLine(result);
 }
```

次に、`hiddenRequestExecutor` を使用して同じ例を記述します。

内部処理の実行中:

```
public static void Main(String[] args)
 {
     RoboServer server = new RoboServer("localhost", 50000);
```

```
    Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
    RequestExecutor executor = new RequestExecutor(cluster);

    var request = new Request("Library:/Tutorials/NewsMagazine.robot");
    request.RobotLibrary = new DefaultRobotLibrary();
    var result = executor.Execute(request);
    Console.WriteLine(result);
}
```

`RequestExecutor` はデフォルトで非表示になっているため、管理する必要はありません。クラスタごとに一つの `RequestExecutor` しか作成できないので、直接使用する場合はアプリケーション全体でその参照を保存する必要があります。`Request.RegisterCluster(cluster)` を使用することは `RequestExecutor` およびライフ サイクル ルールを無視できることを意味します。

`RequestExecutor` は、必要な状態とロジックが含まれており、負荷分散とフェイルオーバー機能を提供します。`RequestExecutor` を直接使用すると、いくつかの追加機能も使用できます。

RequestExecutor の機能

`RequestExecutor` がリポジトリに接続されていない場合、`AddRoboServer(..)` と `RemoveRoboServer(..)` を呼び出すことで、RoboServer を動的に追加または削除できます。これらのメソッドは、`RequestExecutor` の内部で使用される分配リストを変更します。

`RequestExecutor.TotalAvailableSlots` プロパティには、内部分配リスト内にあるすべての RoboServer で未使用の実行スロットの数が含まれます。

これらのメソッドを使用すると、使用可能な実行スロットの数が少なくなった場合に、RoboServer を `RequestExecutor` に動的に追加することができます。

`RequestExecutor` を作成するとき、オプションとして `IRqlEngineFactory` を提供できます。`IRqlEngineFactory` では、RoboServer に接続する時に使用される `RQLProtocol` をカスタマイズすることができます。これは、クライアント証明書を使用してセキュリティを強化する場合など、まれな状況でのみ必要です。詳細については、『Kofax RPA 管理者ガイド』の証明書の章を参照してください。

リポジトリ API

リポジトリ API を使用すると、Management Console のリポジトリのクエリを実行でき、プロジェクト、ロボット、および ロボットを呼び出すために必要な入力のリストを取得します。また、ロボット、タイプ、およびリソース ファイルをプログラムで展開することもできます。

リポジトリ クライアント

リポジトリとの通信は、`RepositoryClient` を通じて `Com.KapowTech.RoboSuite.Api.Repository.Engine` のネームスペースで実現されます。

リポジトリからプロジェクトを取得

```
string UserName = "admin";
 string Password = "admin1234";
 RepositoryClient client = new RepositoryClient("http://localhost:50080/", UserName,
 Password);
 Project[] projects = client.GetProjects();
```

```
    foreach(Project p in projects) {
Console.WriteLine(p);
}
```

ここでは、RepositoryClient がユーザー名とパスワードを使用して http://localhost:50080/ で Management Console のリポジトリに接続するように設定されています。

RepositoryClient が作成された後、 GetProjects() メソッドは、プロジェクトのリストのリポジトリを照会するために使用されます。RepositoryClient メソッドのいずれかを呼び出すときにエラーが発生した場合、RepositoryClientException がスローされることに注意してください。

RepositoryClient には次のメソッドがあります。

RepositoryClient のメソッド

メソッド署名	説明
string **ComputeChecksum**(byte[] bytes)	指定されたファイルのチェックサムを返し、データの整合性を検証します。
void **DeleteConnector**(string projectName, string connectorName, bool silent, AdditionalInfo additionalInfo)	プロジェクトから Connector を削除します。connectorName 引数は Connector のフル パスを指定します。
void **DeleteFile**(RepositoryFile file, AdditionalInfo additionalInfo)	指定されたファイルをリポジトリから削除します。
void **DeleteFolder**(string projectName, string folderPath, AdditionalInfo additionalInfo)	指定されたフォルダをリポジトリから削除します。
void **DeleteResource**(string projectName, string resourceName, bool silent, AdditionalInfo additionalInfo)	指定されたリソースをリポジトリから削除します。
void **DeleteRobot**(string projectName, string robotName, bool silent, AdditionalInfo additionalInfo)	指定されたロボットをリポジトリから削除します。
void **DeleteSnippet**(string projectName, string snippetName, boolean silent, AdditionalInfo additionalInfo)	指定されたスニペットをリポジトリから削除します。
void **DeleteType**(string projectName, string typeName, bool silent, AdditionalInfo additionalInfo)	指定されたタイプをリポジトリから削除します。
void **DeployConnector**(string projectName, string connectorName, byte[] connectorBytes, bool failIfExists, AdditionalInfo additionalInfo)	Connector をプロジェクトにデプロイします。指定された名前の Connector がすでに存在する場合、failIfExists を false に設定することで上書きできます。
void **DeployLibrary**(string projectName, EmbeddedFileBasedRobotLibrary library, bool failIfExists, AdditionalInfo additionalInfo)	ライブラリをサーバーにデプロイします。指定された名前のロボット、タイプ、リソースがすでに存在している場合は、failIfExists を false に設定することで上書きできます。
void **DeployResource**(string projectName, string resourceName, byte[] resourceBytes, bool failIfExists, AdditionalInfo additionalInfo)	リソースをプロジェクトにデプロイします。指定された名前のリソースがすでに存在している場合は、failIfExists を false に設定することで上書きできます。

メソッド署名	説明
`void` **`DeployRobot`**`(string projectName, string robotName, byte[] robotBytes, bool failIfExists, AdditionalInfo additionalInfo)`	ロボットをプロジェクトにデプロイします。指定された名前のロボットがすでに存在している場合は、`failIfExists` を `false` に設定することで上書きできます。
`void` **`DeploySnippet`**`(string projectName, string snippetName, byte[] snippetBytes, bool failIfExists, AdditionalInfo additionalInfo)`	スニペットをプロジェクトにデプロイします。指定された名前のスニペットがすでに存在している場合は、`failIfExists` を `false` に設定することで上書きできます。
`void` **`DeployType`**`(string projectName, string typeName, byte[] typeBytes, boolean failIfExists, AdditionalInfo additionalInfo)`	タイプをプロジェクトにデプロイします。指定された名前のタイプがすでに存在する場合、`failIfExists` を false に設定することで上書きできます。
`byte[]` **`GetBytes`**`(RepositoryFile file)`	指定されたファイルの内容を返します。
`DateTime` **`GetCurrentDate`**`()`	Management Console の現在の日付と時間を返します。
`RepositoryFolder` **`GetFileInventory`**`(string projectName, string folderName, string fileName, FileType fileType)`	プロジェクトからファイルと参照ファイルを返します。ファイル インベントリは、参照を取得するために `RepositoryFolder` にラップされていることに注意してください。
`RepositoryFolder` **`GetFolderInventory`**`(string projectName, string folderName)`	プロジェクトのサブフォルダのフォルダとファイルを返します。
`IDictionary<string, string>` **`GetInfo`**`()`	Management Console およびリポジトリ API についての情報を返します。このメソッドは、次の項目を持つマッピングを返します: • 「アプリケーション」: メジャー バージョン、マイナー バージョン、およびドット バージョンを含む Management Console のバージョン。例: 11.5.0 • 「リポジトリ」: リポジトリ API で使用される最新の DTD の ID (`//Kapow Technologies//DTD Repository 1.6//EN` など) • 「rql」を、ロボット クエリ言語 API で使用される最新の DTD の ID (`//Kapow Technologies//DTD RoboSuite Robot Query Language 1.3//EN` など) にします。
`RepositoryFolder` **`GetProjectInventory`**`(string projectName)`	リポジトリからフォルダとファイルのツリー全体を返します。
`Project[]` **`GetProjects`**`()`	このリポジトリに存在するプロジェクトを返します。
`Cluster[]` **`GetRoboServerClusters`**`(bool onlineRoboServers)`	Management Console に登録されているクラスタと RoboServer のリストを返します。`onlineRoboServer` フラグは、オンラインの RoboServer のみを含めるかどうかを示します。
`Robot[]` **`GetRobotsByTag`** `(string projectName, string tagFragment)`	プロジェクトからの `tagFragment` を含むタグを持つロボットのリストを返します。

メソッド署名	説明
`RobotSignature` **`GetRobotSignature`**`(string projectName, string robotName)`	ロボットのフルパス、このロボットの実行に必要な入力変数、およびロボットが返すまたは保存する可能性のあるタイプのリストとともに、ロボットの署名を返します。
`Robot[]` **`GetRobotsInProject`**`(string projectName)`	プロジェクトで使用可能なロボットを返します。
`void` **`MoveFile`**`(RepositoryFile file, string destinationPath)`	リポジトリ内の指定されたファイルを `destinationPath` で指定されたフォルダに移動します。
`string` **`PingRepository`** `()`	リポジトリ サーバーに ping を実行します。サーバーが正常に動作している場合は null を返します。不正な動作があった場合は、エラー メッセージを返します。
`RemoteCertificateValidationCallback`	現在のプロキシ サーバー設定が含まれます。
`void` **`RenameRobot`**`(RepositoryFile robotFile, string newName)`	指定したロボット ファイルの名前を変更します。
`ServerCertificateValidationCallback`	サーバー証明書の検証に使用される現在のコールバックが含まれます。
`void` **`UpdateFile`**`(RepositoryFile file, byte[] bytes, AdditionalInfo additionalInfo)`	リポジトリ内の指定されたファイルを新しいコンテンツで上書きします。

フル パスはプロジェクト フォルダからの相対パスです。

`AdditionalInfo` パラメータには、Management Console の [リソース] ページの [コミット メッセージ] 列に表示されるコメントが含まれています。このパラメータはオプションです。

指定されたクレデンシャルに十分なアクセス権がない場合は、リクエストが拒否されることがあります。

リポジトリには http 経由でアクセスします。.Net バージョンのリポジトリ API を使用する場合、システム用に設定されたプロキシ サーバーはリポジトリ API によって使用されます。

リポジトリ クライアントを使用したデプロイ

次の例は、`RepositoryClient` を使用して、ローカル ファイル システムからロボットとタイプを展開する方法を示しています。

リポジトリへのデプロイ

```
string user = "test";
 string password = "test1234";
 RepositoryClient client = new RepositoryClient("http://localhost:50080", user,
   password);

 byte[] robotBytes = File.ReadAllBytes("c:\\MyRobots\\Library\\Test.robot");
 byte[] typeBytes = File.ReadAllBytes("c:\\MyRobots\\Library\\Test.type");

 // we assume that no one has deleted the Default project
 client.deployRobot("Default project", "Test.robot", robotBytes, true);
```

```
client.deployType("Default project", "Test.type", typeBytes, true);
```

REST としてのリポジトリ API

リポジトリには、 restful サービス サービスを利用してアクセスすることもできます。

ロギング

.NET API は、主にトラブルシューティングのためにログ データを提供します。.NET API によってログ データが書き込まれることはありませんが、代わりに、アプリケーションが .NET API ログをそのアプリケーションのログと統合するためのプラグインを提供します。アプリケーションは、データをログに記録するためにコールバックまたはインターフェイスを登録します。このようにして、アプリケーションは API データ ログを統合します。

ログ機能は、`Com.KapowTech.RoboSuite.Api.Logging` 名前空間にあります。

- 最も単純な形式のロギングにアクセスするには、デリゲート関数 `public delegate void Logger(Level level, string msg);` を `SimpleLogger.SetLogger(Logger logger)` に登録します。

  API がログに記録する内容がある場合、デリゲートが呼び出され、ログ レベルとメッセージが渡されます。最大エラー レベルを超えた場合、デリゲートは呼び出されません。
- エラー レベルを設定するには、`SimpleLogger.LogLevel` プロパティを使用して、値を `Fatal`、`Error`、`Warn`、`Info`、および`Debug` に設定します。

`SimpleLogger` の機能によって必要な内容が提供されない場合は、`ILogFactory` インターフェイスの実装をアプリケーションの `Registered.Logger` に登録します。このインターフェイスは、リクエストに応じて、特定のタイプ (API のコンポーネント) の `ILog` インターフェイスを実装したオブジェクトを返します。詳細については、コンパイルされたヘルプを参照してください。

ログの記録に `log4net` を使用するアプリケーションは、`API/lib/log4netlogging` ディレクトリにある `log4netlogging.dll` アセンブリを使用できます。

`Com.KapowTech.RoboSuite.Api.Log4NetLogging` 名前空間の `Logging` クラスは、`log4net` ログを登録するために次の 2 つのメソッドを提供します。

- `Logging.Install(string config_file)` は、設定ファイルから設定された `log4net` ログ記録を開始します。
- `Install()` には、アプリケーションによる `log4net` 自体の設定が必要です。

ロギング方法が登録されていない場合、API ではすべてのロギング データが暗黙的に無視されます。

Management Console API:

Management Console では、ロボットの実行をキューに入れることができます。ロボットを RoboServer で直接実行するのではなく、ロボットを Management Console のキューに配置できます。実行 ID の設定、データベース接続の定義、最大実行時間の設定、API 例外時のロボットの強制停止などの一部の機能は、ロボットの実行をキューに入れたときは制御できないことに注意してください。キューイングの利点は次のとおりです。

- 特定のリソース (デバイスなど) が利用できない場合でも、ロボットはキューに登録されます。

- 複数のバージョンの RoboServer が利用可能な場合、ロボットは正しい RoboServer にルーティングされます。
- アプリケーションの構築中にクラスタを管理する必要がなくなります。クラスタ設定は、Management Console でプロジェクト レベルで管理されます。

ロボットの実行をキューに入れる

`QueuedRequest` クラスを使用して、Management Console で実行されるロボットをキューに入れます。

ℹ コードは大文字と小文字が区別されます。

NET API の例:

```
QueuedRequest request = new QueuedRequest("myfolder/myrobot.robot");
 request.RobotLibrary = new RepositoryRobotLibrary
    ("http://localhost:50080/", "Default Project",
   60000, "admin", "admin");
 request.Priority = QueuedRequestPriority.HIGH;
 RqlResult result = request.Execute();
```

コンストラクタ

コンストラクタ	説明
`QueuedRequest(String robot)`	指定されたロボットを使用して新しい QueuedRequest を作成します。 プロパティはリクエストを拡張します。プロパティ パラメータについては、次の表を参照してください。

プロパティ

プロパティ	説明
`PollingIntervalMillis`	ロボットの実行ステータスを更新します。ポーリング間のミリ秒単位の間隔で値と API 例外エラーを返します。間隔が短いと、Management Console の負荷が高くなる可能性がありますが、このリクエストの更新レートが高くなります。 デフォルト値は `1000` (1 秒) です。
`RobotURL`	実行するロボットの名前。ロボットがフォルダにある場合は、パスも含まれます。 ℹ このプロパティに含まれる URL は、実際の URL ではありません。この名前は下位互換性のために付けられました。
`RobotLibrary`	実行リクエストで使用するロボット ライブラリ。`execute()` を呼び出す前に、ロボット ライブラリを設定する必要があります。

プロパティ	説明
優先度	キュー内のリクエストの優先度: `MINIMUM`、`LOW`、`MEDIUM`、`HIGH`、または `MAXIMUM`。 デフォルト値は `MEDIUM` です。
`StopRobotOnApiException`	設定すると、最初の API 例外がクライアントに返された後、ロボットが RoboServer 上で停止します。 デフォルト値は `true` です。
[タイムアウト]	ロボットが RoboServer の実行を待機してキューにとどまる最大時間 (秒単位)。 デフォルト値は `600` (10 分) です。

メソッド

メソッド	説明
`CreateInputVariable(String name)`	指定された名前で新しい入力オブジェクトを作成し、オブジェクトの構築に使用できる RQLObjectBuilder を返します。
`createInputVariable(String name, RQLObject rqlObject)`	指定された名前の新しい入力オブジェクトを `rqlObject` から作成します。これは、ある口ボットからの応答を別の口ボットへの入力として使用する必要がある場合に役立ちます。
`CreateRQLObjectBuilder(String name)`	指定された名前の RQLObject を作成する新しい RQLObjectBuilder を作成します。
`CreateOAuthInputVariable(String name, String userName, String applicationName)`	指定されたユーザー名とアプリケーション名に基づいて、Management Console で OAuth の詳細を検索する OAUTH 入力タイプの入力変数を作成します。Management Console で、アプリケーションと OAuth ユーザーが、実行を要求する口ボットと同じプロジェクトにリンクされていることを確認してください。
`Execute()`	口ボットを実行のためにキューに配置し、実行の終了を待機します。
`execute(RobotResponseHandler handler)`	口ボットを実行のためにキューに配置し、リクエストの処理のさまざまなポイントでハンドラを呼び出します。

第 3 章

Management Console REST API

この章では、製品で提供される Management Console REST サービスについて説明します。次の URL の例を使用することで、Swagger UI から REST サービスにアクセスできます。

http://localhost:8080/ManagementConsole/api/swagger-ui.html

Kofax RPA 11.5.0 では、次の REST サービスを使用できます。

REST サービス	目的
タスク	ロボット タスクのキューイング。このサービスを使用すると、ロボットの実行、ロボット タスクのキューイング、およびロボットの実行結果の取得に必要なロボット入力のサンプル構造を取得できます。

タスク

これはロボット タスク キューイングの REST サービスです。

メソッド

POST robotInputExample
ロボットの実行に必要なロボット入力値のサンプル構造を取得するために使用します。これらは、スケジュールの作成時に「入力値を設定」ステップで設定する値です。
[パラメータ] セクションで、リクエスト ボディを編集して `projectName` プロパティと `robotName` プロパティを指定し、[実行] をクリックします。レスポンスには、ロボット タスクをキューに入れるリクエストを作成するために使用可能なロボット入力のサンプル構造が含まれます。

POST queueRobot
ロボット タスクのキューイングに使用します。
[パラメータ] セクションで、次のようにリクエスト ボディを編集します。

1. `priority` プロパティで、次の中から最も適切な優先度レベルを指定します: `MINIMUM`、`LOW`、`MEDIUM`、`HIGH`、または `MAXIMUM`。優先度の高いタスクに必要なリソースへのアクセス権が提供され、優先度の低いタスクよりも早く実行されます。詳細については、『Kofax RPA のヘルプ』の「スケジュール ジョブのキューイング」を参照してください。
2. `projectName` プロパティと `robotName` プロパティを指定します。
3. `robotInputConfig` プロパティに、robotInputExample メソッドで取得した入力のサンプル構造を貼り付け、入力値を適切に編集します。
4. `timeout` プロパティで、タスクがキューイングを停止するときのタイムアウトを指定します。

5. [実行] をクリックします。

GET getRobotOutput/{ticket}
ロボット出力、キューイング ステータス、エラー情報などのロボットの実行結果を取得するために使用します。

POST queueRobot リクエストが実行されると、このリクエストに対して一意の実行チケットが生成されます。queueRobot レスポンスからチケットをコピーし、getRobotOutput リクエストの [パラメータ] セクションに貼り付けます。[実行] をクリックします。

レスポンスには、ロボットの実行のステータスと結果が含まれます。ロボットに出力が含まれる場合は、`values` プロパティに書き込まれます。