1 Invoking LiveCycle ES Using Web Services

Most LiveCycle ES services located in the service container are configured to expose a web service, with full support for web service definition language (WSDL) generation. That is, you can create proxy objects that are able to consume the native soap stack of a given service. As a result, LiveCycle ES services can exchange and process the following SOAP messages:

SOAP request: Sent to a LiveCycle ES service by a client application requesting an action.

SOAP response: Sent to a client application by a LiveCycle ES service after a SOAP request is processed.

Using web services, you can perform the same operations that you can by using the Java API. A benefit of using web services to invoke LiveCycle ES services is that you can create a client application in a development environment that supports SOAP and are not bound to a specific development environment or programming language. For example, you can create a client application using Microsoft Visual Studio .NET and C# as the programming language.

LiveCycle ES services are exposed over the SOAP protocol and are WSI Basic Profile 1.1 compliant. Web Services Interoperability (WSI) is an open standards organization that promotes web service interoperability across platforms. For information, see http://www.ws-i.org/.

LiveCycle ES supports the following web service standards:

Encoding: Supports only document and literal encoding (which is the preferred encoding according to the WSI Basic Profile). (See Invoking LiveCycle ES using Base64 Encoding.)

SOAP with attachments: Supports both MIME and DIME (Direct Internet Message Encapsulation). These protocols are standard ways of sending attachments over SOAP. DIME is used primarily by .NET applications. (See Invoking LiveCycle ES using DIME.)

WS-Security: Supports a user name password token profile, which is a standard way of sending user names and passwords as part of the WS Security SOAP header. LiveCycle ES also supports HTTP basic authentication. (See Passing credentials using WS-Security headers.)

Note: LiveCycle ES does not expose the MIME field as the swaRef type in a service WSDL.

Note: This section uses Apache Axis to generate Java proxy classes that can be used to invoke LiveCycle ES services. However, you can use other tools such as JAX-WS. If you use this tool, then you should use version JAX-WS 2.1 or later. For information about using Apache Axis to generate Java proxy classes, see Creating Java proxy classes using Apache Axis that uses encoding.

Note: LiveCycle ES does not support the MTOM.

To invoke LiveCycle ES services using a web service, typically you create a proxy library that consumes the service WSDL. You can retrieve a service WDSL by specifying the following URL definition (items in square brackets are optional):

```
http://<your_serverhost>:<your_port>/soap/services/<service_name>?wsdl[&version=<version>][&async=true|false][lc_version=<lc_version>]
```

where:

your_serverhost represents the IP address of the J2EE application server hosting LiveCycle ES.

your_port represents the HTTP port that the J2EE application server uses.

service_name represents the service name.

`version` represents the target version of a service (the latest service version is used by default).

`async` specify the value `true` to enable additional operations for asynchronous invocation (`false` by default).

`lc_version` represents the version of LiveCycle ES that you want to invoke. (See Accessing new functionality using web services.)

To retrieve a WSDL that belongs to a short-lived or long-lived process created in Workbench ES, replace `[service name]` with the name of the process. For example, to retrieve the WSDL that belongs to the `EncryptDocument` short-lived process, specify the following WSDL definition:

`http://<your_serverhost>:<your_port>/soap/services/EncryptDocument?wsdl`

Note: For information about the example `EncryptDocument` short-lived process, see Short lived process example.

The following table lists service WSDL definitions (assuming that LiveCycle ES is deployed on the local host and the post is 8080).

Service	WSDL definition
Assembler	`http://localhost:8080/soap/services/AssemblerService?wsdl`
Barcoded Forms	`http://localhost:8080/soap/services/BarcodedFormsService?wsdl`
Convert PDF	`http://localhost:8080/soap/services/ConvertPDFService?wsdl`
Distiller service	`http://localhost:8080/soap/services/DistillerService?wsdl`
DocConverter service	`http://localhost:8080/soap/services/DocConverterService?WSDL`
Encryption	`http://localhost:8080/soap/services/EncryptionService?wsdl`
Forms	`http://localhost:8080/soap/services/FormsService?wsdl`
Form Data Integration	`http://localhost:8080/soap/services/FormDataIntegration?wsdl`
Generate PDF	`http://localhost:8080/soap/services/GeneratePDFService?wsdl`
Generate3d PDF	`http://localhost:8080/soap/services/Generate3dPDFService?WSDL`
Output	`http://localhost:8080/soap/services/OutputService?wsdl`

Service	WSDL definition
PDF Utilities	`http://localhost:8080/soap/services/PDFUtilityService?wsdl`
Reader Extensions	`http://localhost:8080/soap/services/ReaderExtensionsService?wsdl`
Repository	`http://localhost:8080/soap/services/RepositoryService?wsdl`
Rights Management	`http://localhost:8080/soap/services/RightsManagementService?wsdl`
Signature	`http://localhost:8080/soap/services/SignatureService?wsdl`
XMP Utilities	`http://localhost:8080/soap/services/XMPUtilityService?wsdl`

Note: You can also view WSDL files that are located in the LiveCycle ES SDK directory at *[install directory]*\Adobe\LiveCycle8\Livecycle_ES_SDK\wsdl.

Services that cannot be accessed using web services

Although most LiveCycle ES services can be accessed using a WSDL, some LiveCycle ES services cannot be accessed in this manner. The following LiveCycle ES services cannot be invoked using web services:

- Service registry
- Component registry
- Job Manager service operation named `createJob`

Web service data types

One of the most important data types exposed in a web service is a `BLOB` type. This type is a mapping of the `com.adobe.idp.Document` class, which is used to send and retrieve binary data (for example, PDF files, Word documents, XML, and so on) to and from LiveCycle ES services. The `BLOB` type is defined in a service WSDL as follows:

```
<complexType name="BLOB">
 <sequence>
  <element maxOccurs="1" minOccurs="0" name="contentType" type="xsd:string" />
  <element maxOccurs="1" minOccurs="0" name="binaryData"
type="xsd:base64Binary" />
  <element maxOccurs="1" minOccurs="0" name="attachmentID" type="xsd:string"
/>
  <element maxOccurs="1" minOccurs="0" name="remoteURL" type="xsd:string" />
 </sequence>
</complexType>
```

If a LiveCycle ES service operation requires a `BLOB` type as an input value, you must create an instance of the `BLOB` type in your application logic and assign values to fields that belong to the `BLOB` instance as follows:

1. To pass data as text encoded in a Base64 format, set the data in the `BLOB.binaryData` field and set the data type in the MIME format (for example `application/pdf`) in the `BLOB.contentType` field. (See Invoking LiveCycle ES using Base64 Encoding.)

2. To pass data in a MIME or DIME attachment, attach the data to the SOAP request using the SOAP framework's native API and set the attachment ID in the `BLOB.attachmentID` field. (See Invoking LiveCycle ES using DIME.)

3. If data is hosted on a web server and accessible over an HTTP URL, set the HTTP URL in the `BLOB.remoteURL` field. Note that this URL should be accessible from the server. (See Invoking LiveCycle ES using BLOB Data over HTTP.)

If the service returns a `BLOB` type, by default, the result data is hosted on the application server and its HTTP URL is returned in the `BLOB.remoteURL` field. The only exception is when the service takes a `BLOB` instance as an input value and the data is supplied with a MIME or DIME attachment. In this situation, the result data is returned as a MIME or DIME attachment (the output attachment type will match the input attachment type), and the output `BLOB.attachmentID` field contains the result attachment identifier.

To override the default output `BLOB` behavior, extend the SOAP endpoint URL with a suffix as follows:

```
http://<your_serverhost>:<your_port>/soap/services/<service
name>?blob=base64|dime|mime|http
```

1. Set the blob suffix to base64 to return the data in the `BLOB.binaryData` field.

2. Set the blob suffix to dime or mime to return the data as a corresponding attachment type with the attachment identifier returned in the `BLOB.attachmentID` field. Use the SOAP framework's proprietary API to read the data from the attachment.

3. Set the blob suffix to http to keep the data on the application server and return the URL pointing to the data in the `BLOB.remoteURL` field.

Caution: It is strongly recommended that you do not exceed 30 MB when populating a `BLOB` object by invoking its `setBinaryData` method. Otherwise, you may encounter an `OutOfMemory` exception.

The following table lists Java data types and shows the corresponding web service data type.

Java data type	Web service data type
`java.lang.byte[]`	`xsd:base64Binary`
`java.lang.Boolean`	`xsd:boolean`

Java data type	Web service data type
java.util.Date	The DATE type, which is defined in a service WSDL as follows: <complexType name="DATE"> <sequence> <element maxOccurs="1" minOccurs="0" name="date" type="xsd:dateTime" /> <element maxOccurs="1" minOccurs="0" name="calendar" type="xsd:dateTime" /> </sequence> </complexType> If a LiveCycle ES service operation takes a java.util.Date value as input, the SOAP client application must pass the date in the DATE.date field. Setting the DATE.calendar field in this case causes a run-time exception. If the service returns a java.util.Date, then the date is retuned in the DATE.date field.
java.util.Calendar	The DATE type, which is defined in a service WSDL as follows: <complexType name="DATE"> <sequence> <element maxOccurs="1" minOccurs="0" name="date" type="xsd:dateTime" /> <element maxOccurs="1" minOccurs="0" name="calendar" type="xsd:dateTime" /> </sequence> </complexType> If a LiveCycle ES service operation takes a java.util.Calendar value as input, the SOAP client application must pass the date in the DATE.caledendar field. Setting the DATE.date field in this case causes a run-time exception. If the service returns a java.util.Calendar, then the date is returned in the DATE.calendar field.
java.math.BigDecimal	xsd:decimal
com.adobe.idp.Document	BLOB
java.lang.Double	xsd:double
java.lang.Float	xsd:float
java.lang.Integer	xsd:int
java.util.List	MyArrayOf_xsd_anyType
java.lang.Long	xsd:long

Java data type	**Web service data type**

`java.util.Map`	The `apachesoap:Map`, which is defined in a service WSDL as follows:
	`<schema elementFormDefault="qualified" targetNamespace="http://xml.apache.org/xml-soap" xmlns="http://www.w3.org/2001/XMLSchema">`
	`<complexType name="mapItem">`
	`<sequence>`
	`<element name="key" nillable="true" type="xsd:anyType"/>`
	`<element name="value" nillable="true" type="xsd:anyType"/>`
	`</sequence>`
	`</complexType>`
	`<complexType name="Map">`
	`<sequence>`
	`<element maxOccurs="unbounded" minOccurs="0" name="item" type="apachesoap:mapItem"/>`
	`</sequence>`
	`</complexType>`
	`</schema>`
	The Map is represented as a sequence of key/value pairs.

`java.lang.Object`	`xsd:anyType`
`java.lang.Short`	`xsd:short`
`java.lang.String`	`xsd:string`

Java data type	Web service data type

`org.w3c.dom.Document`	The XML type, which is defined in a service WSDL as follows:
	`<complexType name="XML">`
	`  <sequence>`
	`  <element maxOccurs="1" minOccurs="0" name="document" type="xsd:string" />`
	`  <element maxOccurs="1" minOccurs="0" name="element" type="xsd:string" />`
	`  </sequence>`
	`</complexType>`
	If a LiveCycle ES service operation takes an `org.w3c.dom.Document` value as input, the SOAP client application must pass the XML data in the `XML.document` field. Setting the `XML.element` field in this case will cause a run-time exception. If the service returns an `org.w3c.dom.Document`, then the XML data is returned in the `XML.document` field.

`org.w3c.dom.Element`	The XML type, which is defined in a service WSDL as follows:
	`<complexType name="XML">`
	`  <sequence>`
	`  <element maxOccurs="1" minOccurs="0" name="document" type="xsd:string" />`
	`  <element maxOccurs="1" minOccurs="0" name="element" type="xsd:string" />`
	`  </sequence>`
	`</complexType>`
	If a LiveCycle ES service operation takes an `org.w3c.dom.Element` as input, the SOAP client application must pass the XML data in the `XML.element` field. Setting the `XML.document` field in this case will cause a run-time exception. If the service returns an `org.w3c.dom.Element`, then the XML data is retuned in the `XML.element` field.

Accessing multiple services using web services

Due to namespace conflicts, data objects cannot be shared between multiple service WSDLs. Different services may share data types and, therefore the services share the definition of these types in the WSDLs. For example, you cannot add two .NET client assemblies that contain a `BLOB` data type to the same .NET client project. If you attempt to do so, you will generate a compile error.

The following list specifies data types that cannot be shared between multiple service WSDLs:

- `User`
- `Principals`
- `PrincipalReference`
- `Groups`
- `Roles`
- `BLOB`

To avoid this problem, it is recommended that the services WSDL's are compiled together with a special `/sharetypes` command line option. For example:

```
wsdl /sharetypes
http://localhost:8080/soap/services/RightsManagementService?wsdl
http://localhost:8080/soap/services/DirectoryManagerService?wsdl
```

This command produces a single CS file containing no duplicate data types.

Accessing new functionality using web services

Some LiveCycle ES 8.2 services introduced new functionality that can be accessed using web services. For example, the Encryption service introduced the ability to encrypt a PDF document with a certificate. For information, see Encrypting PDF Documents with Certificates.

When generating a client stub using LiveCycle ES 8.2 service WSDL definition, the available WSDL remains compatible with LiveCycle ES 8.0.1.1 so client applications generated for previous LiveCycle ES versions work without making changes. For example, to access the Encryption service WSDL, specify the following WSDL definition:

```
http://localhost:8080/soap/services/EncryptionService?wsdl
```

However, to access new functionality introduced in LiveCycle ES 8.2, you specify the `lc_version` attribute in the WSDL definition. For example, to access new Encryption service functionality, specify the following WSDL definition:

```
http://localhost:8080/soap/services/EncryptionService?wsdl&lc_version=8.2.1
```

Note: When setting the `lc_version` attribute, ensure that you use three digets. For example, 8.2.1 is equal to version 8.2.

1 Invoking LiveCycle ES using Base64 Encoding

You can invoke a LiveCycle ES service using base64 encoding. Base64 encoding refers to encoding attachments that are sent with a web service invocation request. That is, BLOB data is base64 encoded, not the entire SOAP message.

This section discusses invoking the following LiveCycle ES short-lived process, named *EncryptDocument*, using base64 encoding.

When this process is invoked, it performs the following actions:

1. Obtains the unsecure PDF document that is passed to the service as an attachment. This action is based on the `SetValue` operation.
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation.
3. Saves the password-encrypted PDF document as a PDF file to the local file system. This process also returns the encrypted PDF document as an output value. This action is based on the `WriteDocument` operation.

This section discusses creating Java proxy classes that invoke the above service as well as a Microsoft .NET client assembly. Both Java proxy classes and the Microsoft client assembly uses base64 encoding.

Note: This process is not based on an existing LiveCycle ES process. To following along with the code examples that are related to this section, create a process named *EncryptDocument* using Workbench ES. For information, see LiveCycle Workbench ES Help.

Note: Before reading this section, it is recommended that you are familiar with invoking LiveCycle ES using SOAP. (See Invoking LiveCycle ES Using Web Services.)

Note: All web service Quick Starts in *Programming with LiveCycle ES* use base64 encoding.

Creating a .NET client assembly that uses base 64 encoding

You can create a .NET client assembly to invoke a LiveCycle ES service from a Microsoft Visual Studio .NET project. To create a .NET client assembly that uses base64 encoding, perform the following steps:

1. Create a proxy class based on a LiveCycle ES invocation URL.

2. Create a new Microsoft Visual Studio .NET project that will produce the .NET client assembly.

Creating a proxy class

You can create a proxy class that is used to create the .NET client assembly by using a tool that accompanies Microsoft Visual Studio. The name of the tool is wsdl.exe and is located in the Microsoft Visual Studio installation folder. To create a proxy class, open the command prompt and navigate to the folder that contains the wsdl.exe file. Enter the following command at the command prompt:

```
wsdl
http://localhost:8080/soap/services/EncryptDocument?WSDL&lc_version=8.2.1
```

By default, this tool creates a CS file in the same folder that is based on the name of the WSDL. In this situation, it creates a CS file named *EncryptDocumentService.cs*. You use this CS file to create a proxy object that lets you invoke the service that was specified in the invocation URL.

Amend the URL in the proxy class to include `?blob=base64` to ensure that the `BLOB` object returns binary data; that is, in the proxy class, locate the following line of code:

```
"http://localhost:8080/soap/services/EncryptDocument";
```

and change it to:

```
"http://localhost:8080/soap/services/EncryptDocument?blob=base64";
```

Note: For more information about the wsdl.exe tool, see the MSDN Help.

Note: This section uses EncryptDocument as an example. If you are creating a .NET client assembly for another LiveCycle ES service, ensure that you replace EncryptDocument with the name of the LiveCycle ES service for which you are building a client assembly.

Note: Instead of modifying the generated code, the URL can be set using `service.URL` property. For example, you can assign `http://localhost:8080/soap/services/EncryptDocument?blob=base64` to this property. Setting this property is used in the code example that accompanies the DIME section. (See Invoking LiveCycle ES using DIME.)

Developing the .NET client assembly

Create a new Visual Studio Class Library project that produces a .NET client assembly. The CS file that you created using wsdl.exe can be imported into this project. This project produces a DLL file that you can use in other Visual Studio .NET projects to invoke a service.

➤ **To develop the .NET client assembly:**

1. Start Microsoft Visual Studio .NET.
2. Create a new Class Library project and name it DocumentService.
3. Import the CS file that you created using wsdl.exe.
4. In the **Project** menu, select **Add Reference**.
5. In the Add Reference dialog box, select **System.Web.Services.dll**.
6. Click **Select** and then click **OK**.

7. Compile and build the project.

Note: This procedure creates a .NET client assembly named *DocumentService.dll* that you can use to send SOAP requests to the EncryptDocument service.

Caution: Make sure that you added `?blob=base64` to the URL in the proxy class that is used to create the .NET client assembly. Otherwise, you cannot retrieve binary data from the `BLOB` object. (See Web service data types.)

Referencing the .NET client assembly

Place your newly-created .NET client assembly on the computer where you are developing your client application. After you place the .NET client assembly in a directory, you can reference it from a project. You must also reference the `System.Web.Services` library from your project. If you do not reference this library, you cannot use the .NET client assembly to invoke a service.

➤ **To reference the .NET client assembly:**

1. In the **Project** menu, select **Add Reference**.
2. Click the **.NET** tab.
3. Click **Browse** and locate the DocumentService.dll file.
4. Click **Select** and then click **OK**.

Invoking a service using a .NET client assembly that uses base64 encoding

You can invoke the EncryptDocument service (that was built in Workbench ES) using a .NET client assembly that uses base64 encoding. (See Invoking LiveCycle ES using Base64 Encoding.)

To invoke the EncryptDocument service, perform the following steps:

1. Create a Microsoft .NET client assembly that consumes the EncryptDocument service WSDL. For information, see Creating a .NET client assembly that uses base 64 encoding.
2. Reference the Microsoft .NET client assembly. For information, see Referencing the .NET client assembly.
3. Using the Microsoft .NET client assembly, create an `EncryptDocumentService` object by invoking its default constructor.
4. Set the `EncryptDocumentService` object's `Credentials` property with a `System.Net.NetworkCredential` object. Within the `System.Net.NetworkCredential` constructor, specify a LiveCycle ES user name and the corresponding password. You must set authentication values to enable your .NET client application to successfully exchange SOAP messages with LiveCycle ES.
5. Create a `BLOB` object by using its constructor. The `BLOB` object is used to store a PDF document pass to the EncryptDocument process.
6. Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the PDF document and the mode in which to open the file.

7. Create a byte array that stores the content of the `System.IO.FileStream` object. You can determine the size of the byte array by getting the `System.IO.FileStream` object's `Length` property.
8. Populate the byte array with stream data by invoking the `System.IO.FileStream` object's `Read` method and passing the byte array, the starting position, and the stream length to read.
9. Populate the `BLOB` object by assigning its `binaryData` property with the contents of the byte array.
10. Invoke the EncryptDocument process by invoking the `EncryptDocumentService` object's `invoke` method and passing the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.
11. Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents the file location of the password-encrypted document.
12. Create a byte array that stores the data content of the `BLOB` object that was returned by the `EncryptDocumentService` object's `invoke` method. Populate the byte array by getting the value of the `BLOB` object's `binaryData` data member.
13. Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
14. Write the byte array contents to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

View Quick Start

Creating Java proxy classes using Apache Axis that uses encoding

You can use the Apache Axis WSDL2Java tool to convert a service WSDL into Java proxy classes so that you can invoke service operations. Using Apache Ant, you can generate Axis library files from the Repository service WSDL that lets you invoke the Repository service. You can download Apache Axis at the URL http://ws.apache.org/axis/.

The Apache Axis WSDL2Java tool generates JAVA files that contain methods that can be invoked by a client application to send SOAP requests to a service. SOAP requests received by a service are decoded by the same Axis-generated libraries and turned back into the methods and arguments they represent.

Generating Axis library files

Apache Ant generates Axis library files by referencing a service WSDL. You can generate Axis Java library files by performing the following steps:

1. Install Apache Ant on the client computer. It is available at http://ant.apache.org/bindownload.cgi.
   - Add the bin directory to your class path.
   - Set the `ANT_HOME` environment variable to the directory where you installed Ant.
2. Install Apache Axis 1.4 on the client computer. It is available at http://ws.apache.org/axis/.
3. Set up the class path to use the Axis JAR files in your web service client, as described in the Axis installation instructions at http://ws.apache.org/axis/java/install.html.

4. Use the Apache WSDL2Java tool in Axis to generate Java proxy classes. You must create an Ant build script to accomplish this task. The following script is a sample Ant build script named *build.xml*:

```
<?xml version="1.0"?>
<project name="axis-wsdl2java">

<path id="axis.classpath">
<fileset dir="C:\axis-1_4\lib" >
   <include name="**/*.jar" />
</fileset>
</path>

<taskdef resource="axis-tasks.properties" classpathref="axis.classpath" />

<target name="repository-wsdl2java-client" description="task">
<axis-wsdl2java
   output="C:\JavaFiles"
   testcase="false"
   serverside="false"
   verbose="true"
   username="administrator"
   password="password"

url="http://localhost:8080/soap/services/EncryptDocument?WSDL&lc_version=8.2
.1" >
</axis-wsdl2java>
</target>

</project>
```

   Within this Ant build script, notice that the `url` property is set to reference the EncryptDocument WSDL running on localhost. The `username` and `password` properties must be set to a valid LiveCycle ES user name and password.

5. Create a BAT file to execute the Ant build script. The following command can be located within a BAT file that is responsible for executing the Ant build script:

```
ant -buildfile "build.xml" repository-wsdl2java-client
```

This Ant build script generates JAVA files that can invoke the EncryptDocument service. The JAVA files are written to the C:\JavaFiles folder as specified by the `output` property. To successfully invoke the EncryptDocument service, you must import all of these JAVA files into your class path. By default, these files belong to a Java package named `com.adobe.idp.services`. It is recommended that you place all of these CLASS files into a JAR file and then import the JAR file into your client application's class path.

Note: There are different ways to put .CLASS files into a JAR. One way is using a Java IDE like Eclipse. Create a new Java project and create a `com.adobe.idp.services` package (all .CLASS files belong to this package). Next import all the .CLASS files into the package. Finally, export the project as a JAR file.

Amend the URL in the `EncryptDocumentServiceLocator` class to include `?blob=base64` to ensure that the `BLOB` object returns binary data. That is, in the `EncryptDocumentServiceLocator` class, locate the following line of code:

```
http://localhost:8080/soap/services/EncryptDocument;
```

and change it to:

```
http://localhost:8080/soap/services/EncryptDocument?blob=base64;
```

Instead of modifying the generated code, the URL can be set using `service.URL` property. For example, you can assign `http://localhost:8080/soap/services/EncryptDocument?blob=base64` to this property. Setting this property is used in the code example that accompanies the DIME section. (See Invoking LiveCycle ES using DIME.)

You must also add the following Axis JAR files to your Java project's class path:

- activation.jar
- axis.jar
- commons-codec-1.3.jar
- commons-collections-3.1.jar
- commons-discovery.jar
- commons-logging.jar
- dom3-xml-apis-2.5.0.jar
- jai_imageio.jar
- jaxen-1.1-beta-9.jar
- jaxrpc.jar
- log4j.jar
- mail.jar
- saaj.jar
- wsdl4j.jar
- xalan.jar
- xbean.jar
- xercesImpl.jar

These JAR files are in the [install directory]/Adobe/LiveCycle8/sdk/lib/thirdparty directory.

Caution: Make sure that you added `?blob=base64` to the URL in the `EncryptDocumentServiceLocator` class. Otherwise, you cannot retrieve binary data from the `BLOB` object. For information, see Web service data types.

Invoking a service using Axis-generated library files that use base64 encoding

You can invoke the EncryptDocument service (that was built in Workbench ES) using Axis-generated library files that uses base64 encoding. (See Invoking LiveCycle ES using Base64 Encoding.)

To invoke the EncryptDocument service using Axis-generated library files, perform the following steps:

1. Create Java proxy classes that consume the EncryptDocument service WSDL. (See Creating Java proxy classes using Apache Axis that uses encoding.)
2. Include the Java proxy and Axis classes into your class path.
3. Create an `EncryptDocumentServiceLocator` object by using its constructor.

4. Create an `EncryptDocument` object by invoking the `EncryptDocumentServiceLocator` object's `getEncryptDocument` method.

5. Set authentication values by setting the `javax.xml.rpc.Stub.USERNAME_PROPERTY` and `javax.xml.rpc.Stub.PASSWORD_PROPERTY` values with valid LiveCycle ES user name and password values.

   ```
   ((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.
   USERNAME_PROPERTY, "administrator");
   ((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.
   PASSWORD_PROPERTY, "password");
   ```

6. Retrieve the PDF document to send to theEncryptDocument process by creating a `java.io.FileInputStream` object by using its constructor and passing a string value that specifies the location of the PDF document.

7. Create a byte array and populate it with the contents of the `java.io.FileInputStream` object.

8. Create a `BLOB` object by using its constructor.

9. Populate the `BLOB` object by invoking its `setBinaryData` method and passing the byte array.

10. Invoke the EncryptDocument process by invoking the `EncryptDocument` object's `invoke` method and passing the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.

11. Create a byte array to store the data stream that represents the encrypted PDF document by invoking the `BLOB` object's (ensure you use the `BLOB` object returned by the invoke method) `getBinaryData` method.

12. Create a `java.io.File` object by using its constructor. This object will represent the encrypted PDF document.

13. Create a java.io.FileOutputStream object by usings it constructor and passing the `java.io.File` object.

14. Invoke the `java.io.FileOutputStream` object's `write` method and pass the byte array that contains the data stream that represents the encrypted PDF document.

View Quick Start

1 Invoking LiveCycle ES using DIME

You can invoke LiveCycle ES services using SOAP with attachments. LiveCycle ES supports both MIME and DIME (Direct Internet Message Encapsulation) web service standards. DIME enables binary attachments, such as PDF documents, to be sent along with invocation requests rather than encoding the attachment which is the case when using base64 encoding, which may increase the attachment size. (See Invoking LiveCycle ES using Base64 Encoding.)

This section discusses invoking the following LiveCycle ES short-lived process, named *EncryptDocument*, using DIME.

When this process is invoked, it performs the following actions:

1. Obtains the unsecure PDF document that is passed to the service as an attachment. This action is based on the `SetValue` operation.
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation.
3. Saves the password-encrypted PDF document as a PDF file to the local file system. This process also returns the encrypted PDF document as an output value. This action is based on the `WriteDocument` operation.

Note: This process is not based on an existing LiveCycle ES process. To following along with the code examples that are related to this section, create a process named *EncryptDocument* using Workbench ES. For information, see LiveCycle Workbench ES Help.

Note: Before reading this section, it is recommended that you are familiar with invoking LiveCycle ES using SOAP. (See Invoking LiveCycle ES Using Web Services.)

Creating a .NET project that uses DIME

To create a .NET project that is able to invoke a LiveCycle ES service using DIME, perform the following tasks:

- Install Web Services Enhancements 2.0 on your development computer.
- From within your .NET project, create a web reference to the LiveCycle ES service.

Installing Web Services Enhancements 2.0

Install Web Services Enhancements 2.0 on your development computer and integrate it with Microsoft Visual Studio .NET. You can download Web Services Enhancements 2.0 from the following URL:

www.microsoft.com/downloads/search.aspx

From this web page, search for Web Services Enhancements 2.0 and download it onto your development computer. This places a file named Microsoft WSE 2.0 SPI.msi on your computer. Run the install program and follow the online directions.

Note: Web Services Enhancements 2.0 supports DIME. The supported version of Microsoft Visual Studio is 2003 when working with Web Services Enhancements 2.0. Web Services Enhancements 3.0 does not support DIME, but supports MTOM. However, LiveCycle ES 8.2 does not supprt MTOM.

Creating a web reference to a LiveCycle ES service

After you install Web Services Enhancements 2.0 on your development computer and create a Microsoft .NET project, create a web reference to the LiveCycle ES service that you want to invoke using DIME. For example, to create a web reference to the EncryptDocument process and assuming that LiveCycle ES is installed on the local computer, specify the following URL:

```
http://localhost:8080/soap/services/EncryptDocument?WSDL
```

After you create a web reference, the following two proxy data types are available for you to use within your .NET project `EncryptDocumentService` and `EncryptDocumentServiceWse`. To invoke EncryptDocument using DIME, use the `EncryptDocumentServiceWse` type.

Note: Before creating a web reference to the LiveCycle ES service, ensure that you reference Web Services Enhancements 2.0 in your project. (See Installing Web Services Enhancements 2.0.)

➤ **To reference the WSE library:**

1. In the Project menu, select Add Reference.
2. In the Add Reference dialog box, select Microsoft.Web.Services2.dll.
3. Select System.Web.Services.dll.
4. Click Select. and then click OK.

➤ **To create a web reference to a LiveCycle ES service:**

1. In the Project menu, select Add Web Reference.
2. In the URL dialog box, specify the URL to the LiveCycle ES service.
3. Click Go.
4. Click Add Reference.

Note: Ensure that you enable your .NET project to use the WSE library. From within the Project Explorer, right-click the project name and select enable WSE 2.0. Ensure that the check box on the dialog box that appears is selected.

Invoking a service using DIME in a .NET project

You can invoke a LiveCycle ES service using DIME. To fully explain how to do this task, this section describes how to invoke the EncryptDocument process. This process accepts an unsecure PDF document to encrypt and returns a password-encrypted PDF document.

To invoke the EncryptDocument process using DIME, perform the following steps:

1. Create a Microsoft .NET project that enables you to invoke a LiveCycle ES service using DIME. Ensure that you include Web Services Enhancements 2.0 and create a web reference to the LiveCycle ES service. (See Creating a .NET project that uses DIME.)

2. After setting a web reference to the EncryptDocument process, create an `EncryptDocumentServiceWse` object by using its default constructor.

3. Set the `EncryptDocumentServiceWse` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies theLiveCycle ES user name and password value.

4. Create a `Microsoft.Web.Services2.Dime.DimeAttachment` object by using its constructor and passing the following values:
   - A string value that specifies GUID value. You can obtain a GUID value by invoking the `System.Guid.NewGuid.ToString` method.
   - A string value that specifies the content type. Because this process requires a PDF document, specify `application/pdf`.
   - A `TypeFormat` enumeration value. Specify `TypeFormat.MediaType`.
   - A string value that specifies the location of the PDF document to pass to the LiveCycle ES process.

5. Create a `BLOB` object by using its constructor.

6. Add the DIME attachment to the `BLOB` object by assigning the `Microsoft.Web.Services2.Dime.DimeAttachment` object's `Id` data member value to the `BLOB` object's `attachmentID` data member.

7. Invoke the `EncryptDocumentServiceWse.RequestSoapContext.Attachments.Add` method and pass the `Microsoft.Web.Services2.Dime.DimeAttachment` object.

8. Invoke the EncryptDocument process by invoking the `EncryptDocumentServiceWse` object's `invoke` method and passing the `BLOB` object that contains the DIME attachment. This process returns an encrypted PDF document within a `BLOB` object.

9. Obtain the attachment identifier value by getting the value of the returned `BLOB` object's `attachmentID` data member.

10. Iterate through the attachments located in `EncryptDocumentServiceWse.ResponseSoapContext.Attachments` and use the attachment identifier value to obtain the encrypted PDF document.

11. Obtain a `System.IO.Stream` object by getting the value of the `Attachment` object's `Stream` data member.

12. Create a byte array and pass that byte array to the `ystem.IO.Stream` object's `Read` method. This method populates the byte array with a data stream that represents the encrypted PDF document.

13. Create a `System.IO.FileStream` object by invoking its constructor and passing a string value that represents an PDF file location. This object represents the encrypted PDF document.

14. Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.

15. Write the contents of the byte array to the PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

View Quick Start

Creating Java proxy classes using Apache Axis that uses DIME

You can use the Apache Axis WSDL2Java tool to convert a service WSDL into Java proxy classes so that you can invoke service operations. Using Apache Ant, you can generate Axis library files from a LiveCycle ES service WSDL that lets you invoke the service. You can download Apache Axis at the URL http://ws.apache.org/axis/.

The Apache Axis WSDL2Java tool generates JAVA files that contain methods that can be invoked by a client application to send SOAP requests to a service. SOAP requests received by a service are decoded by the same Axis-generated libraries and turned back into the methods and arguments they represent.

Note: To create Java proxy classes that uses DIME to invoke a LiveCycle ES service, follow the same process that is described in the Invoking LiveCycle ES using Base64 Encoding section. The only difference is that you do not have to amend the URL to include `?blob=base64` to ensure that the `BLOB` object returns binary data.

To invoke the *EncryptDocument* service using Axis-generated library files and using DIME, perform the following steps:

1. Create Java proxy classes that consume the EncryptDocument service WSDL. For information, see Creating Java proxy classes using Apache Axis that uses DIME.

2. Include the Java proxy classes into your class path.

3. Create an `EncryptDocumentServiceLocator` object by using its constructor.

4. Create an `URL` object by using its constructor and passing a string value that specifies the LiveCycle ES service WSDL definition. Ensure that you specify `?blob=dime` at the end of the SOAP endpoint URL. For example, `http://localhost:8080/soap/services/EncryptDocument?blob=dime`.

5. Create an `EncryptDocumentSoapBindingStub` object by invoking its constructor and passing the `EncryptDocumentServiceLocator` object and the `URL` object.

6. Set the LiveCycle ES user name and password value by invoking the `EncryptDocumentSoapBindingStub` object's `setUsername` and `setPassword` methods.

```
encryptionClientStub.setUsername("administrator");
encryptionClientStub.setPassword("password");
```

7. Retrieve the PDF document to send to the EncryptDocument service by creating a `java.io.File` object and passing a string value that specifies the PDF document location.

8. Create a `javax.activation.DataHandler` object by using its constructor and passing a `javax.activation.FileDataSource` object. The `javax.activation.FileDataSource`

object can be created by using its constructor and passing the `java.io.File` object that represents the PDF document.

9. Create an `org.apache.axis.attachments.AttachmentPart` object by using its constructor and passing the `javax.activation.DataHandler` object.
10. Attach the attachment by invoking the `EncryptDocumentSoapBindingStub` object's `addAttachment` method and passing the `org.apache.axis.attachments.AttachmentPart` object.
11. Create a `BLOB` object by usings its constructor. Populate the `BLOB` object with the attachment identifier value by invoking the `BLOB` object's `setAttachmentID` method and passing the attachment identifier value. This value can be obtained by invoking the `org.apache.axis.attachments.AttachmentPart` object's `getContentId` method.
12. Invoke the EncryptDocument process by invoking the `EncryptDocumentSoapBindingStub` object's `invoke` method and passing the `BLOB` object that contains the DIME attachment. This process returns an encrypted PDF document within a `BLOB` object.
13. Obtain the attachment identifier value by invoking the returned `BLOB` object's `getAttachmentID` method. This method returns a string value that represents the identifier value of the returned attachment.
14. Retrieve the attachments by invoking the `EncryptDocumentSoapBindingStub` object's `getAttachments` method. This method returns an array of `Objects` that represent the attachments.
15. Iterate through the attachments (the `Object` array) and use the attachment identifier value to obtain the encrypted PDF document. Each element is an `org.apache.axis.attachments.AttachmentPart` object.
16. Obtain the `javax.activation.DataHandler` object associated with the attachment by invoking the `org.apache.axis.attachments.AttachmentPart` object's `getDataHandler` method.
17. Obtain a `java.io.FileStream` object by invoking the `javax.activation.DataHandler` object's `getInputStream` method.
18. Create a byte array and pass that byte array to the `java.io.FileStream` object object's `read` method. This method populates the byte array with a data stream that represents the encrypted PDF document.
19. Create a `java.io.File` object by using its constructor. This object will represent the encrypted PDF document.
20. Create a `java.io.FileOutputStream` object by usings it constructor and passing the `java.io.File` object.
21. Invoke the `java.io.FileOutputStream` object's `write` method and pass the byte array that contains the data stream that represents the encrypted PDF document.

View Quick Start

1 Invoking LiveCycle ES using BLOB Data over HTTP

You can invoke LiveCycle ES services using web services and passing BLOB data over HTTP. Passing BLOB data over HTTP is an alternative web service technique that you can use when you do not want to use base64 encoding, DIME, or MIME. For example, you can pass data over HTTP in a Microsoft .NET project that uses Web Service Enhancement 3.0, which does not support DIME or MIME. When using BLOB data over HTTP, input data is uploaded before the LiveCycle ES service is invoked.

This section discusses invoking the following LiveCycle ES short-lived process, named *EncryptDocument*, by passing BLOB data over HTTP.

When this process is invoked, it performs the following actions:

1. Obtains the unsecure PDF document that is passed to the service as an attachment. This action is based on the `SetValue` operation.
2. Encrypts the PDF document with a password. This action is based on the `PasswordEncryptPDF` operation.
3. Saves the password-encrypted PDF document as a PDF file to the local file system. This process also returns the encrypted PDF document as an output value. This action is based on the `WriteDocument` operation.

Note: This process is not based on an existing LiveCycle ES process. To following along with the code examples that are related to this section, create a process named *EncryptDocument* using Workbench ES. (See LiveCycle Workbench ES Help.)

Note: Before reading this section, it is recommended that you are familiar with invoking LiveCycle ES using SOAP. (See Invoking LiveCycle ES Using Web Services.)

Creating a .NET client assembly that uses data over HTTP

To create a client assembly that uses data over HTTP, follow the same process as specified in the Creating a .NET client assembly that uses base 64 encoding section. However, amend the URL in the proxy class to include `?blob=http` instead of `?blob=base64` to ensure that data is passed over HTTP. That is, in the proxy class, locate the following line of code:

```
"http://localhost:8080/soap/services/EncryptDocument";
```

and change it to:

```
"http://localhost:8080/soap/services/EncryptDocument?blob=http";
```

Referencing the .NET client assembly

Place your newly-created .NET client assembly on the computer where you are developing your client application. After you place the .NET client assembly in a directory, you can reference it from a project. You must also reference the `System.Web.Services` library from your project. If you do not reference this library, you cannot use the .NET client assembly to invoke a service.

➤ **To reference the .NET client assembly:**

1. In the **Project** menu, select **Add Reference**.
2. Click the **.NET** tab.
3. Click **Browse** and locate the DocumentService.dll file.
4. Click **Select** and then click **OK**.

Invoking a service using a .NET client assembly that uses BLOB data over HTTP

You can invoke the EncryptDocument service (that was built in Workbench ES) using a .NET client assembly that uses data over HTTP. (See Invoking LiveCycle ES using BLOB Data over HTTP.)

To invoke the EncryptDocument service, perform the following steps:

5. Reference the Microsoft .NET client assembly. For information, see Referencing the .NET client assembly.
6. Using the Microsoft .NET client assembly, create an `EncryptDocumentService` object by invoking its default constructor.
7. Set the `EncryptDocumentService` object's `Credentials` property with a `System.Net.NetworkCredential` object. Within the `System.Net.NetworkCredential` constructor, specify a LiveCycle ES user name and the corresponding password. You must set authentication values to enable your .NET client application to successfully exchange SOAP messages with LiveCycle ES.
8. Create a `BLOB` object by using its constructor. The `BLOB` object is used pass data to the EncryptDocument process.
9. Assign a string value to the `BLOB` object's `remoteURL` data member that specifies the URI location of a PDF document to pass to the EncryptDocument service.
10. Invoke the EncryptDocument process by invoking the `EncryptDocumentService` object's `invoke` method and passing the `BLOB` object. This process returns an encrypted PDF document within a `BLOB` object.
11. Create a `System.UriBuilder` object by using its constructor and passing the value of the returned `BLOB` object's `remoteURL` data member.

12. Convert the `System.UriBuilder` object to a `System.IO.Stream` object. (The C# quick start that follows this list illustrates how to perform this task.)
13. Create a byte array and populate it with the data located in the `System.IO.Stream` object.
14. Create a `System.IO.BinaryWriter` object by invoking its constructor and passing the `System.IO.FileStream` object.
15. Write the byte array contents to a PDF file by invoking the `System.IO.BinaryWriter` object's `Write` method and passing the byte array.

View Quick Start

Creating Java proxy classes using Apache Axis that uses BLOB data over HTTP

You can use the Apache Axis WSDL2Java tool to convert a service WSDL into Java proxy classes so that you can invoke service operations. Using Apache Ant, you can generate Axis library files from a LiveCycle ES service WSDL that lets you invoke the service. You can download Apache Axis at the URL http://ws.apache.org/axis/.

Note: To use Java proxy classes that uses data over HTTP to invoke a LiveCycle ES service, follow the same process that is described in the Invoking LiveCycle ES using Base64 Encoding section. The only difference is that you have to amend the URL to include `?blob=http`.

To invoke the *EncryptDocument* service using Axis-generated library files and SOAP over HTTP, perform the following steps:

1. Create Java proxy classes that consume the EncryptDocument service WSDL.
2. Include the Java proxy and Axis classes into your class path.
3. Create an `EncryptDocumentServiceLocator` object by using its constructor.
4. Create an `EncryptDocument` object by invoking the `EncryptDocumentServiceLocator` object's `getEncryptDocument` method.
5. Set authentication values by setting the `javax.xml.rpc.Stub.USERNAME_PROPERTY` and `javax.xml.rpc.Stub.PASSWORD_PROPERTY` values with valid LiveCycle ES user name and password values.

```
((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.
USERNAME_PROPERTY, "administrator");
((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.
PASSWORD_PROPERTY, "password");
```

6. Create a `BLOB` object by using its constructor.
7. Populate the `BLOB` object by invoking its `setRemoteURL` method and passing a string value that specifies the URI location of a PDF document to pass to the EncryptDocument service.
8. Invoke the EncryptDocument process by invoking the `EncryptDocument` object's `invoke` method and passing the `BLOB` object that contains the PDF document. This process returns an encrypted PDF document within a `BLOB` object.

9. Create a byte array to store the data stream that represents the encrypted PDF document by invoking the `BLOB` object's (ensure you use the `BLOB` object returned by the invoke method) `getBinaryData` method.

10. Create a `java.io.File` object by using its constructor. This object will represent the encrypted PDF document.

11. Create a java.io.FileOutputStream object by usings it constructor and passing the `java.io.File` object.

12. Invoke the `java.io.FileOutputStream` object's `write` method and pass the byte array that contains the data stream that represents the encrypted PDF document.

View Quick Start

1 Invoking Long-Lived Processes

You can programmatically invoke long-lived processes that were created in Workbench ES. A long-lived process is invoked asynchronously and cannot be invoked synchronously due to the following factors:

- A process can span a significant amount of time.
- A process can span organizational boundaries.
- A process needs external input in order for it to finish. For example, consider a situation where a form is sent to a manager, who may be out of the office. In this situation, the process will not finish until the manager returns and fills out the form.

When a long-lived process is invoked, LiveCycle ES creates an invocation identifier value as part of creating a record that tracks the status of the long-lived process and is stored in the LiveCycle ES database. Using the invocation identifier value, you can track the status of the long-lived process. (This section discusses how to get the invocation identifier value and how to use it to track the status of the long-lived operation.) You can use the process invocation identifier value to perform Process Manager operations such as terminating a running process instance. (See Terminating Process Instances.)

Note: LiveCycle ES does not create an invocation identifier value nor create a record when a short-lived process is invoked or when a service operation is invoked such the Encryption service's `EncryptwithPassword` operation. As a result, you cannot obtain an invocation identifier value when invoking a service operation that is not part of a long-lived process.

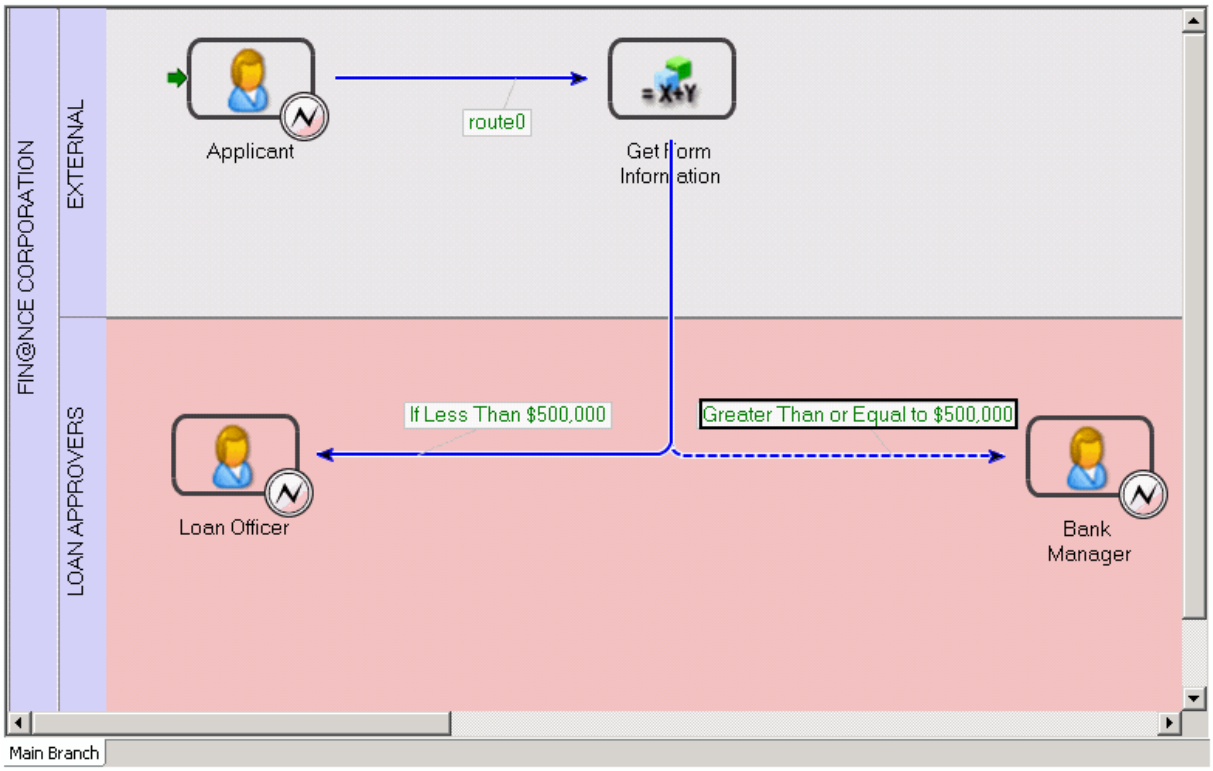
When you invoke a service, you can specify the version that you want to invoke. The following choices are available:

No version: If you do not specify version information, the invocation request is routed to the latest version of the service.

Explicit version: If you specify the explicit version, the invocation request is routed to that specific version. Setting an explicit version enables a client application to request that it is always executed against the version that works (the lowest risk). Most often, the preferable setting is the `originalVersion`, whereby client applications can give the service container enough information to ensure that the request is routed to a compatible version.

Original version: If you specify the service's original version, the invocation request is routed to the service with the latest version that is compatible with the specified version. A service version is considered compatible with an inbound request if it has the same major version number as specified in the original version.

For the purpose of this discussion, consider the following long-lived process named *MortgageLoan - Prebuilt*.



This process is invoked when an applicant submits a loan form. If the loan request is greater than $500,000, the loan request is sent to a bank manager; otherwise, the loan request is sent to a loan officer. However, the process is not complete until either the loan officer or the bank manager approves or rejects the loan request.

This process requires XML data as input. The name of the input XML parameter is *inXML*. The XML data is used to populate the form that is sent to either the loan officer or bank manager, depending on the loan amount. If the loan amount is greater, then $500,000.00, then the process is routed to the bank manager. Otherwise, it is routed to the loan officer.

For the purposes of this discussion, assume that the following XML data is used as input to this process.

```
<MortgageApp>
<MortgageFields>
     <PropertyPrice>700000</PropertyPrice>
     <DownPayment>100000</DownPayment>
     <Mortgage>600000</Mortgage>
     <Term>20</Term>
     <InterestRate>6.0</InterestRate>
</MortgageFields>
<ApplicantFields>
     <LastName>White</LastName>
     <FirstName>Sam</FirstName>
     <PhoneNumber>555-5555</PhoneNumber>
     <SSN>123-456-678</SSN>
</ApplicantFields>
</MortgageApp>
```

In the *GetForm* information step (the second operation in the process shown in the previous illustration), an XFAForm variable named `LoanForm` is created. This variable specifies the location of the form design that is used. The XML data that is required as input to this process is merged with the specified form design. The following XPATH expression, which is set in Workbench ES, is responsible for merging the XML data that is passed to this process with the form design.

```
/process_data/LoanForm/object/data/xdp/datasets/data = /process_data/inXML
```

After this step, the form (now populated with the XML data) is sent to either the loan officer or the bank manager, who accesses the form in Workspace ES. This section discusses how to programmatically invoke this process as well as creating the XML data that must be sent (the quick starts associated with this section dynamically creates the XML data that is shown in this section).

Note: The *MortgageLoan - Prebuilt* process is an example process that is available with LiveCycle ES. However, it was modified to be programmatically invoked. This process accepts an input value named *inXML* whose data type is XML. The XML data is assigned to the *LoanForm* process variable whose data type is `XFAForm` using the expression specified in this section.

View summary of steps

Summary of steps

To invoke a long-lived process, perform the following steps:

1. Include project files.
2. Create a service client.
3. Prepare input values.
4. Invoke the service operation.
5. Retrieve the results.

View Java walkthrough View web service walkthrough

Include project files

Include necessary files into your development project. If you are creating a client application using Java, then include the necessary JAR files. If you are using web services, then make sure that you include the proxy files.

The following JAR files must be added to your project's classpath:

- adobe-livecycle-client.jar
- adobe-usermanager-client.jar
- adobe-jobmanager-client-sdk.jar
- adobe-utilities.jar (Required if LiveCycle ES is deployed on JBoss)
- jbossall-client.jar (Required if LiveCycle ES is deployed on JBoss)

For information about the location of these JAR files, see Including LiveCycle ES library files.

Create a service client

Before you can programmatically invoke a long-lived process, you must create a `com.adobe.idp.dsc.clientsdk.ServiceClient` object.

If you are using web services to invoke a long-lived process, the name of the service client is based on the long-lived process name. For example, if the name of the long-lived process is `MortgageLoan - Prebuilt`, then the name of the service client is `MortgageLoanPrebuiltService`.

Prepare input values

You must prepare input values that are required by the long-lived process. This step is specific to the process. That is, if the process requires four input values, then you must prepare four input values to pass to the process.

If you are using the Java Invocation API to invoke a long-lived process, you must pass required input values by using a `java.util.HashMap` object. For each parameter to pass to a service, invoke the `java.util.HashMap` object's `put` method and specify the name-value pair that is required by the long-lived process. You must specify the exact name of the parameter that belongs to the long-lived process and a corresponding value. For example, if the name of the input parameter is `inXML` and requires XML data, you can create an `org.w3c.dom.Document` object that represents the required XML data.

Note: This topic uses the `MSXML2.DOMDocument50Class` class to create the XML data in the web service example. To uses this data type, ensure that you reference the Microsoft XML V5 library in your Visual Studio project.

Invoke the service operation

To invoke a long-lived process, specify the name of the long-lived process operation. Typically the name of a long-lived process is named `invoke`. When invoking the service operation, you must pass required input values. For example, to invoke the long-lived process introduced in this section, you must pass XML data. This XML data is merged with the loan form that is sent to either the bank manager or the loan officer.

Note: If you are using web services, the name of the operation is `invoke_Async`.

Retrieve the results

Because a long-lived process is invoked asynchronously, an identifier value that is used to obtain the status of the long-lived process can be retrieved. To obtain the status of a long-lived process, you use classes that belong to the `com.adobe.idp.jobmanager` Java package (this is why you must include the adobe-jobmanager-client-sdk.jar file in your class path). For information about these classes, see LiveCycle ES API References.

Invoking a long-lived process using the Java invocation API

To invoke a long-lived process using the Java invocation API, perform the following tasks:

1. Include project files

   Include client JAR files, such as the adobe-livecycle-client.jar, in your Java project's class path. (See Including LiveCycle ES library files.)

2. Create a service client

   - Create a `ServiceClientFactory` object that contains connection properties. (See Setting connection properties.)
   - Create a `ServiceClient` object by using its constructor and passing the `ServiceClientFactory` object. A `ServiceClient` object lets you invoke a service operation. It handles tasks such as locating, dispatching, and routing invocation requests

3. Prepare input values

   - Create a `java.util.HashMap` object by using its constructor.
   - Invoke the `java.util.HashMap` object's `put` method for each input parameter to pass to the long-lived process.

   Note: In the Java quick start that accompanies this section, an `org.w3c.dom.Document` object that represents the XML data to pass to the process is created.

4. Invoke the service operation

   - Create an `InvocationRequest` object by invoking the `ServiceClientFactory` object's `createInvocationRequest` method and passing the following values:
     - A string value that specifies the name of the long-lived process to invoke. To invoke the long-lived process introduced in this section, specify `MortgageLoan - Prebuilt`.
     - A string value that represents the name of the process operation. Typically the name of a long-lived process operation is `invoke`.
     - The `java.util.HashMap` object that contains the parameter values that the service operation requires.
     - A Boolean value that specifies `false`, which creates an asynchronous request (this is required to invoke a long-lived operation).
   - Send the invocation request to the service by invoking the `ServiceClient` object's `invoke` method and passing the `InvocationRequest` object. The `invoke` method returns an `InvocationReponse` object.

5. Retrieve the results

   - Get the invocation identifier of the long-lived operation by invoking the `InvocationReponse` object's `getInvocationId` method.
   - Create a `JobManager` object by using its constructor and passing the `ServiceClientFactory` object.
   - Create a `JobId` object that represents the status of the long-lived process by using its constructor and passing the invocation identifier value that was returned by the `getInvocationId` method.
   - Check the status of the long-lived operation by invoking the `JobManager` object's `getStatus` method and passing the `JobId` object. This method returns a `JobStatus` object.

- Determine the status of the long-lived process by invoking the `JobStatus` object's `getStatusCode` method. If the long-lived operation completed successfully, this method returns `JobStatus.JOB_STATUS_COMPLETED`. If the long-lived operation did not complete successfully, this method returns `JobStatus.JOB_STATUS_FAILED`. For information about other status values, see the LiveCycle ES API References.
- Dispose of the job by invoking the `JobManager` object's `disposeJob` method and passing the `JobId` object that corresponds to the job you want to dispose.

View Quick Start

Invoking a long-lived process using the web service API

To invoke a long-lived process using the web service API, perform the following tasks:

1. Include project files
   - Create a Microsoft .NET client assembly that consumes the MortgageLoan - Prebuilt service WSDL. (See Creating a .NET client assembly that uses base 64 encoding.)
   - Reference the Microsoft .NET client assembly. (See Referencing the .NET client assembly.)

   Note: This section assumes that you create a process named *MortgageLoan - Prebuilt* that accepts XML data as input.

2. Create a service client
   - Using the Microsoft .NET client assembly, create a `MortgageLoanPrebuiltService` object by invoking its default constructor.
   - Set the `MortgageLoanPrebuiltService` object's `Credentials` data member with a `System.Net.NetworkCredential` value that specifies the user name and password value.

3. Prepare input values

   Prepare input values that are required to invoke the long-lived process. For example, to invoke the *MortgageLoan - Prebuilt* long-lived process, create XML data.

   Note: In the C# quick start that accompanies this section, a `MSXML2.DOMDocument50Class` object is used to create the XML data that is passed to the process. To uses this data type, ensure that you reference the Microsoft XML V5 library in your Visual Studio project.

4. Invoke the service operation
   - Create an `XML` object by using its constructor. This data type is created from the MortgageLoan - Prebuilt service WSDL.
   - Populate the `XML` object with XML data by assigning its `document` data member with the value of the `MSXML2.DOMDocument50Class` object's `xml` data member.
   - Invoke the process by invoking the `MortgageLoanPrebuiltService` object's `invoke_Async` method and passing the `XML` object.

5. Retrieve the results
   - Create a Microsoft .NET client assembly that consumes the JobManager service WSDL. (See Creating a .NET client assembly that uses base 64 encoding.)
   - Reference the Microsoft .NET client assembly. (See Referencing the .NET client assembly.)
   - Create a `JobManagerService` object by using its constructor.
   - Create a `JobId` object by using its constructor.
   - Set the `JobId` object's `id` data memeber with the return value of the `MortgageLoanPrebuiltService` object's `invoke_Async` method.
   - Assign the value `true` to the `JobId` object's `persistent` data memeber.
   - Create a `JobStatus` object by invoking the `JobManagerService` object 's `getStatus` method and passing the `JobId` object.
   - Get the status value by retrieving the value of the `JobStatus` object's `statusCode` data member.

View Quick Start

Invoking a long-lived process using LiveCycle Remoting

To start a LiveCycle ES process from a Flex application, you synchronously invoke the `invoke` operation of the process and provide the input parameter(s) that the operation expects. For information about a long-lived process, see Understanding processes.

Note: These instructions assume that you have already created a process.

You can invoke a long-lived process by performing the following tasks:

1. Use the LiveCycle Administration Console to create a remoting endpoint for the process.
2. Create an `mx:RemoteObject` instance through either ActionScript or MXML. Associate it with the remoting endpoint for the process.
3. Set up a ChannelSet, add a channel to communicate with the LiveCycle ES server, and associate it with the `mx:RemoteObject` instance.
4. Call the service's setCredentials method to specify the user ID and password.
5. Collect the data required by the process and store it as an XML object, in a format that matches the XML used by the process.
6. Send the invocation request to the service by using the RemoteObject instance to call the `invoke` method and passing the variable that contains the XML data.

View Quick Start

1 Invocation API Quick Starts

The following Quick Starts are available for programmatically invoking services:

Description	Remoting API	Java API	Web service API
Invoking a long-lived process using the Java invocation API	N/A	View Quick Start	N/A
Invoking a long-lived process using the web service API	N/A	N/A	View Quick Start
Invoking a long-lived process using LiveCycle Remoting	View Quick Start	N/A	N/A
Invoking a service using a Java client library	N/A	View Quick Start	N/A
Invoking LiveCycle ES using DIME	N/A	View Quick Start	View Quick Start
Invoking LiveCycle ES using Base64 Encoding	N/A	View Quick Start	View Quick Start
Invoking LiveCycle ES using BLOB Data over HTTP	N/A	View Quick Start	View Quick Start
Invoking a service using LiveCycle Remoting	View Quick Start	N/A	N/A

Note: Quick Starts located in *Programming with LiveCycle ES* are based on LiveCycle ES being deployed on JBoss Application Server and the Microsoft Windows operating system. However, if you are using another operating system, such as UNIX, replace Windows-specific paths with paths that are supported by the applicable operating system. Likewise, if you are using another J2EE application server, ensure that you specify valid connection properties. (See Setting connection properties.)

Quick Start: Invoking a long-lived process using the Invocation API

The following Java code example invokes a long-lived process named *MortgageLoan - Prebuilt*. Notice that this process is invoked asynchronously. This quick start contains a user-defined method named `GetDataSource` that creates XML data that is passed to the process. The XML data is created using a `org.w3c.dom.Document` instance. (See Invoking Long-Lived Processes.)

```
/*
 * This Java Quick Start uses the following JAR files
 * 1. adobe-jobmanager-client-sdk.jar
 * 2. adobe-livecycle-client.jar
 * 3. adobe-usermanager-client.jar
 * 4. adobe-utilities.jar
 * 5. jbossall-client.jar (use a different JAR file if LiveCycle ES is not
deployed on JBoss)
 *
```

```
 *  These JAR files are located in the following path:
 * <install directory>/Adobe/LiveCycle8/LiveCycle_ES_SDK/client-libs
 *
 * For complete details about the location of these JAR files,
 * see "Including LiveCycle ES library files" in
 * Programming with LiveCycle ES
 */
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.idp.dsc.clientsdk.ServiceClient;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactoryProperties;
import com.adobe.idp.dsc.FaultResponse;
import com.adobe.idp.dsc.InvocationRequest;
import com.adobe.idp.dsc.InvocationResponse;
import java.util.Properties;
import java.util.Map;
import java.util.HashMap;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Element;
import com.adobe.idp.jobmanager.client.JobManager;
import com.adobe.idp.jobmanager.common.JobId;
import com.adobe.idp.jobmanager.common.JobStatus;

public class InvokeLongLived {
   public static void main(String[] args) {
      try{
         //Set connection properties required to invoke LiveCycle ES
         Properties connectionProps = new Properties();

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_DEFAULT_EJB_E
NDPOINT, "jnp://localhost:1099");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_TRANSPORT_PRO
TOCOL,ServiceClientFactoryProperties.DSC_EJB_PROTOCOL);

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_SERVER_TYPE,
"JBoss");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_US
ERNAME, "administrator");

connectionProps.setProperty(ServiceClientFactoryProperties.DSC_CREDENTIAL_PA
SSWORD, "password");

         //Create a ServiceClientFactory object
         ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

         //Create a ServiceClient object
         ServiceClient myServiceClient = myFactory.getServiceClient();

         //Create a Map object to store the parameter value
         Map params = new HashMap();
```

```
        //Populate the Map object with a parameter value
        //required to invoke the MortgageLoan long-lived process
        //This process requires XML data
        org.w3c.dom.Document inXML = GetDataSource();
        params.put("inXML", inXML);

        //Create an InvocationRequest object
        InvocationRequest request =  myFactory.createInvocationRequest(
          "MortgageLoan - Prebuilt",    //Specify the long-lived process name
            "invoke",      //Specify the operation name
            params,   //Specify input values
            false);   //Create an asynchronous request


        //Send the invocation request to the long-lived process and
        //get back an invocation response object
        InvocationResponse response = myServiceClient.invoke(request);
        String invocationId = response.getInvocationId();

        //Create a Job Manager object to check the
        //results of an asynchronous request
        JobManager jobManager = new JobManager(myFactory);
        JobStatus jobStatus = null;

        //Create a JobID object that represents the status of the
        //long-lived operation
        JobId myJobId = new JobId(invocationId);

        //Wait and check the results of the long-lived operation
        for (int i=0;i<5;i++) {
          Thread.sleep(60000);
          jobStatus = jobManager.getStatus(myJobId);
          System.out.println("Job Status: " + jobStatus.getStatusCode());
          if (jobStatus.getStatusCode()== JobStatus.JOB_STATUS_COMPLETED ||
jobStatus.getStatusCode()==JobStatus.JOB_STATUS_FAILED) {
            break;
            }
          }//end of for loop

        if (jobStatus.getStatusCode()==JobStatus.JOB_STATUS_COMPLETED) {
          System.out.println("INVOCATION COMPLETED SUCCESSFULLY!");
          InvocationResponse jobResponse = jobManager.getResponse(myJobId);
          jobManager.disposeJob(myJobId);

         } else if (jobStatus.getStatusCode()==JobStatus.JOB_STATUS_FAILED) {
          System.out.println("INVOCATION COMPLETED FAILED!");
          FaultResponse _fr = jobManager.getFaultResponse(new
JobId(invocationId));
          System.out.println(_fr.getStackTrace());
          jobManager.disposeJob(myJobId);

         } else {
          System.out.println("INVOCATION STATUS " + jobStatus.getStatusCode()
+ " NOT COMPLETE.");
         }
```

```
    }catch (Exception e) {

        e.printStackTrace();
      }
   }
    //Create XML data to pass to the long-lived process
    private static org.w3c.dom.Document GetDataSource()
    {
        org.w3c.dom.Document document = null;

        try
        {
           //Create DocumentBuilderFactory and DocumentBuilder objects
           DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
           DocumentBuilder builder = factory.newDocumentBuilder();

           //Create a new Document object
           document = builder.newDocument();

           //Create MortgageApp - the root element in the XML
           Element root = (Element)document.createElement("MortgageApp");
           document.appendChild(root);

           //Create Mortgage fields and append it to MortgageApp
           Element MortgageFields =
(Element)document.createElement("MortgageFields");
           root.appendChild(MortgageFields);

           //Create ApplicantFields and append it to MortgageApp
           Element ApplicantFields =
(Element)document.createElement("ApplicantFields");
           root.appendChild(ApplicantFields);

           //Create the PropertyPrice element - a child to Mortgage fields
           Element PropertyPrice =
(Element)document.createElement("PropertyPrice");
           PropertyPrice.appendChild(document.createTextNode("700000"));
           MortgageFields.appendChild(PropertyPrice);

           //Create the DownPayment element - a child to Mortgage fields
           Element DownPayment
=(Element)document.createElement("DownPayment");
           DownPayment.appendChild(document.createTextNode("100000"));
           MortgageFields.appendChild(DownPayment);

           //Create the Term element - a child to Mortgage fields
           Element Term = (Element)document.createElement("Term");
           Term.appendChild(document.createTextNode("20"));
           MortgageFields.appendChild(Term);

           //Create the InterestRate element - a child to Mortgage fields
           Element InterestRate =
(Element)document.createElement("InterestRate");
```

```
            InterestRate.appendChild(document.createTextNode("6.0"));
            MortgageFields.appendChild(InterestRate);

            //Create the LastName element - a child to ApplicantFields fields
            Element LastName = (Element)document.createElement("LastName");
            LastName.appendChild(document.createTextNode("White"));
            ApplicantFields.appendChild(LastName);

            //Create the FirstName element - a child to ApplicantFields fields
            Element FirstName = (Element)document.createElement("FirstName");
            FirstName.appendChild(document.createTextNode("Sam"));
            ApplicantFields.appendChild(FirstName);

            //Create the PhoneNumber element - a child to ApplicantFields fields
            Element PhoneNumber =
(Element)document.createElement("PhoneNumber");
            PhoneNumber.appendChild(document.createTextNode("555-5555"));
            ApplicantFields.appendChild(PhoneNumber);

            //Create the SNN element - a child to ApplicantFields fields
            Element SSN = (Element)document.createElement("SSN");
            SSN.appendChild(document.createTextNode("123-456-678"));
            ApplicantFields.appendChild(SSN);

           }
        catch (Exception e) {
              System.out.println("The following exception occurred:
"+e.getMessage());
            }
       return document;
        }
    }
```

Quick Start: Invoking a long-lived process using the web service API

The following C# code example invokes a long-lived process named *MortgageLoan - Prebuilt*. Notice that this process is invoked asynchronously. This quick start contains a user-defined method named `GetDataSource` that creates XML data that is passed to the process. The XML data is created using a `MSXML2.DOMDocument50Class` instance. (See Invoking Long-Lived Processes.)

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

            try
         {
```

```
            //Create a ReaderExtensionsServiceService client object
                MortgageLoanPrebuiltService mortgageClient = new
MortgageLoanPrebuiltService();
                mortgageClient.Credentials = new
System.Net.NetworkCredential("administrator", "password");

                //Create XML data to send to the long-lived process
                MSXML2.DOMDocument50Class xmlData = getXMLSource();
                XML inXML = new XML();
                inXML.document= xmlData.xml;

                //Invoke the long-lived process
                String invocationID = mortgageClient.invoke_Async(inXML);

                //Create a Job Manager object to check the
                //results of an asynchronous request
                JobManagerService jobManager = new JobManagerService();
                jobManager.Credentials = new System.Net.NetworkCredential(
                    "administrator",
                    "password"
                    );

                //Create a JobID object that represents the status of the
                //long-lived operation
                JobId jobId = new JobId();
                jobId.id = invocationID;
                jobId.persistent = true;
                JobStatus jobStatus = jobManager.getStatus(jobId);
                System.Int16 val2 = jobStatus.statusCode;
                Console.WriteLine("The value of the long-lived operation is
"+val2);
            }

            catch (System.Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
        }

        //Create XML data to pass to the long-lived process
        static private MSXML2.DOMDocument50Class getXMLSource()
        {

            MSXML2.DOMDocument50Class myXMLDoc = new
MSXML2.DOMDocument50Class();
            MSXML2.IXMLDOMElement MortgageApp = null;
            MSXML2.IXMLDOMElement MortgageFields = null;
            MSXML2.IXMLDOMElement ApplicantFields = null;
            MSXML2.IXMLDOMElement PropertyPrice = null;
            MSXML2.IXMLDOMElement DownPayment = null;
            MSXML2.IXMLDOMElement Term = null;
            MSXML2.IXMLDOMElement Mortgage = null;
            MSXML2.IXMLDOMElement InterestRate = null;
            MSXML2.IXMLDOMElement LastName = null;
            MSXML2.IXMLDOMElement FirstName = null;
```

```
MSXML2.IXMLDOMElement PhoneNumber = null;
MSXML2.IXMLDOMElement SSN = null;

//Create MortgageApp - the root element in the XML
MortgageApp = myXMLDoc.createElement("MortgageApp");
myXMLDoc.appendChild(MortgageApp);

//Create Mortgage fields and append it to MortgageApp
MortgageFields = myXMLDoc.createElement("MortgageFields");
MortgageApp.appendChild(MortgageFields);

//Create ApplicantFields element and append it to MortgageApp
ApplicantFields = myXMLDoc.createElement("ApplicantFields");
MortgageApp.appendChild(ApplicantFields);

//Create the PropertyPrice element - a child to Mortgage fields
PropertyPrice = myXMLDoc.createElement("PropertyPrice");
PropertyPrice.appendChild(myXMLDoc.createTextNode("900000"));
MortgageFields.appendChild(PropertyPrice);

//Create the DownPayment element - a child to Mortgage fields
DownPayment = myXMLDoc.createElement("DownPayment");
DownPayment.appendChild(myXMLDoc.createTextNode("100000"));
MortgageFields.appendChild(DownPayment);

//Create the Mortgage element - a child to Mortgage fields
Mortgage = myXMLDoc.createElement("Mortgage");
Mortgage.appendChild(myXMLDoc.createTextNode("800000"));
MortgageFields.appendChild(Mortgage);

//Create the Term element - a child to Mortgage fields
Term = myXMLDoc.createElement("Term");
Term.appendChild(myXMLDoc.createTextNode("20"));
MortgageFields.appendChild(Term);

//Create the InterestRate element - a child to Mortgage fields
InterestRate = myXMLDoc.createElement("InterestRate");
InterestRate.appendChild(myXMLDoc.createTextNode("6.0"));
MortgageFields.appendChild(InterestRate);

//Create the LastName element - a child to ApplicantFields fields
LastName = myXMLDoc.createElement("LastName");
LastName.appendChild(myXMLDoc.createTextNode("MCCue"));
ApplicantFields.appendChild(LastName);

//Create the FirstName element - a child to ApplicantFields fields
FirstName = myXMLDoc.createElement("FirstName");
FirstName.appendChild(myXMLDoc.createTextNode("Kevin"));
ApplicantFields.appendChild(FirstName);

//Create the PhoneNumber element - a child to ApplicantFields fields
PhoneNumber = myXMLDoc.createElement("PhoneNumber");
PhoneNumber.appendChild(myXMLDoc.createTextNode("555-5555"));
ApplicantFields.appendChild(PhoneNumber);
```

```
            //Create the SSN element - a child to ApplicantFields fields
            SSN = myXMLDoc.createElement("SSN");
            SSN.appendChild(myXMLDoc.createTextNode("123-456-678"));
            ApplicantFields.appendChild(SSN);

            return myXMLDoc;
        }

    }
}
```

Quick Start: Invoking a long-lived process using LiveCycle Remoting

The following MXML code example shows how a Flex™ client application can pass XML data that represents a user's mortgage application to a LiveCycle ES mortgage application process and start the process.

When the Flex user submits the mortgage application, the LiveCycle ES process is invoked and populated with the XML data from the Flex application. The XML data passed in the `lc.invoke` operation matches the structure of the schema defined for the mortgage application process in LiveCycle ES. The `createXML` method in the script block of the MXML provides the XML data to the process; it is stored in the variable named `xml`, which is passed a parameter to the process in the `lc.invoke({xmlData: xml})` method. The XML elements in the `createXML` method are bound to the form fields in which the user enters data, shown after this example.

Note: The following example requires additional mxml files to be included in your project. You can obtain these mxml files at the following URL: http://www.adobe.com/devnet/livecycle/quickstart/longlivedprocess/.

The complete application and source code are included in the LiveCycle ES installation.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
creationPolicy="all"
   backgroundGradientColors="[#FFFFFF, #FFFFFF]"
creationComplete="initApp()">

   <mx:Script>
      <![CDATA[

         import mx.controls.Alert;
         import mx.rpc.events.FaultEvent;
         import mx.rpc.events.ResultEvent;
         import flash.net.navigateToURL;
         import mx.messaging.ChannelSet;
         import mx.messaging.channels.AMFChannel;
         import mx.collections.ArrayCollection;
         import mx.rpc.livecycle.JobId;
         import mx.rpc.livecycle.JobStatus;
         import mx.rpc.livecycle.DocumentReference;

         // Holds the job ID returned by LC.JobManager
         private var ji:JobId;
```

```
            private function initApp():void
            {
                /* var amfChannel:AMFChannel = new AMFChannel(null,
                   "http://10.60.147.127:9081/remoting/messagebroker/amf");
               */
                 var amfChannel:AMFChannel = new AMFChannel(null,
                  "http://10.60.84.33:8080/remoting/messagebroker/amf");
                  //10.60.84.33:8080
               var channelSet:ChannelSet = new ChannelSet();
               channelSet.addChannel(amfChannel);
               lc.channelSet = channelSet;
               jmService.channelSet = channelSet;
            }

            private function submitApplication():void
            {
               var xml:XML = createXML();
               lc.invoke_Async({xmlData: xml});
            }

            // Handles async call that invokes the long-lived process
            private function resultHandler(event:ResultEvent):void
            {
               ji = event.result as JobId;
               bGetStatus.enabled = true;
               jobStatusDisplay.text = "Job Status ID: " + ji.jobId as String;
            }

            private function getStatus():void
            {
               jmService.getStatus(ji);
            }

            private function getStatusHandler(event:ResultEvent):void
            {
               var res:JobStatus = event.result as JobStatus;
               var resInt:int = res.statusCode;
               jobStatusDisplay.text = new String(res.toString());
               if(res.statusCode == JobStatus.JOB_STATUS_COMPLETED)
               {
                 bGetResponse.enabled = true;
               }
            }

            // Get results (should only be called once you know that the job has
    completed)
            private function getResp():void{
               jmService.getResponse(ji);
            }

            private function getResponseHandler(event:ResultEvent):void
            {
               var res:Object = event.result;
               var docRef:DocumentReference = res["pdfDoc"] as DocumentReference;
               //Alert.show(docRef.url);
```

```
            navigateToURL(new URLRequest(docRef.url as String), "_blank");
         }

         private function createXML():XML
         {
            var model:XML =

               <mortgageApplication>

                  <applicant>
                     <firstName>{applicant.firstName.text}</firstName>
                     <lastName>{applicant.lastName.text}</lastName>
                     <daytimePhone>{applicant.daytimePhone.text}</daytimePhone>
                     <mobilePhone>{applicant.mobilePhone.text}</mobilePhone>
                     <notifyMobile>{applicant.notifyMobile.selected}
                     </notifyMobile>
                     <email>{applicant.email.text}</email>
                     <usCitizen>{applicant.citizenYes.selected}</usCitizen>
                  </applicant>

                  <property>
                     <address>{property.address.text}</address>
                     <city>{property.city.text}</city>
                     <state>{property.stateCB.selectedLabel}</state>
                     <zip>{property.zip.text}</zip>
                     <type>
                        {property.singleFamily.selected?"single
family":"condominium"}
                     </type>
                  </property>

                  <mortgage>
                     <price>{mortgage.price.value}</price>
                     <downPayment>{mortgage.downPayment.value}</downPayment>
                     <loanAmount>
                        {mortgage.price.value - mortgage.downPayment.value}
                     </loanAmount>
                     <closingDate>
                        {mortgage.closingDate.selectedDate}
                     </closingDate>
                  </mortgage>

                  <employment/>
                  <assets/>
               </mortgageApplication>

            var jobList:ArrayCollection = employment.jobList;
            var length:int = jobList.length;
            for (var i:int=0; i<length; i++) {
               var job:XML =
                  <job>
                     <company>{jobList[i].company}</company>
                     <startDate>{jobList[i].startDate}</startDate>
                     <endDate>{jobList[i].endDate}</endDate>
                     <salary>{jobList[i].salary}</salary>
```

```
                </job>;
            model.employment[0].appendChild(job);
         }

         var accountList:ArrayCollection = assets.accountList;
         length = accountList.length;
         for (var j:int=0; j<length; j++) {
            var account:XML =
               <account>
                  <bank>{accountList[j].bank}</bank>
                  <accountId>{accountList[j].accountId}</accountId>
                  <balance>{accountList[j].balance}</balance>
               </account>;
            model.assets.appendChild(account);
         }

         return model;
      }

      private function faultHandler(event:FaultEvent):void
      {
         Alert.show(
            event.fault.faultString + "\n" +
            event.fault.faultCode + "\n" +
            event.fault.faultDetail,
            "Error");
      }

   ]]>
</mx:Script>

<!-- <mx:Style source="main.css"/> -->
<!-- Declare the RemoteObject and set its destination to the mortgage-app
remoting endpoint defined in LiveCycle. -->
<mx:RemoteObject id="lc" destination="mortgage-app"
   result="resultHandler(event)" fault="faultHandler(event)"/>

<mx:RemoteObject id="jmService" destination="LC.JobManager"
showBusyCursor="true" fault="faultHandler(event)">
    <mx:method name="getStatus" result="getStatusHandler(event)"/>
    <mx:method name="getResponse" result="getResponseHandler(event)"/>
</mx:RemoteObject>


<mx:Panel title="My Mortgage Application" backgroundAlpha="0.8"
backgroundImage="img/background.jpg">

   <mx:Accordion width="700" height="550" backgroundAlpha=".8">
      <Applicant id="applicant" label="Applicant Information"/>
      <Property id="property" label="Property Information"/>
      <MortgageInfo id="mortgage" label="Mortgage Information"/>
      <Employment id="employment" label="Employment History" />
      <Assets id="assets" label="Financial Assets"/>
   </mx:Accordion>
```

```
      <mx:ControlBar bottom="20">
         <mx:Button label="Submit Application"
icon="@Embed('img/icon_save.png')" click="submitApplication()"/>
         <mx:Button label="Get Status" id="bGetStatus" enabled="false"
icon="@Embed('img/icon_save.png')" click="getStatus()"/>
         <mx:Button label="Get Response" id="bGetResponse" enabled="false"
icon="@Embed('img/icon_save.png')" click="getResp()"/>
      </mx:ControlBar>
      <mx:Text id="jobStatusDisplay" width="300" />
   </mx:Panel>
</mx:Application>
```

The XML elements in the `createXML` method are bound to the form fields in which the user enters data, such as the following form (contained in another MXML file in the application):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Form xmlns:mx="http://www.adobe.com/2006/mxml">

   <mx:StringValidator source="{firstName}" property="text"/>
   <mx:SocialSecurityValidator source="{ssn}" property="text"/>
   <mx:PhoneNumberValidator source="{daytimePhone}" property="text"/>
   <mx:PhoneNumberValidator source="{mobilePhone}" property="text"/>
   <mx:EmailValidator source="{email}" property="text"/>

   <mx:FormItem label="First Name" required="true">
      <mx:TextInput id="firstName" width="200"/>
   </mx:FormItem>

   <mx:FormItem label="Last Name" required="true">
      <mx:TextInput id="lastName" width="200"/>
   </mx:FormItem>

   <mx:Spacer height="12"/>

   <mx:FormItem label="Social Security Number" required="true">
      <mx:TextInput id="ssn" width="200"/>
   </mx:FormItem>

   <mx:FormItem label="Daytime Phone Number" paddingTop="12" required="true">
      <mx:TextInput id="daytimePhone" width="200"/>
   </mx:FormItem>

   <mx:FormItem label="Mobile Phone Number">
      <mx:TextInput id="mobilePhone" width="200"/>
      <mx:HBox horizontalGap="0">
         <mx:CheckBox id="notifyMobile"/>
         <mx:Text text="Notify me on this numberwhen the status of my mortgage
changes" width="200"/>
      </mx:HBox>
   </mx:FormItem>

   <mx:Spacer height="12"/>

   <mx:FormItem label="Email Address" required="true">
      <mx:TextInput id="email" width="200"/>
   </mx:FormItem>
```

```
   <mx:Spacer height="12"/>

   <mx:FormItem label="Are you a US citizen?">
      <mx:RadioButton id="citizenYes" label="Yes" selected="true"
groupName="citizen"/>
      <mx:RadioButton id="citizenNo" label="No" groupName="citizen"/>
   </mx:FormItem>

</mx:Form>
```

For a complete set of form fields, see the Mortgage example.

Quick Start: Invoking the Repository service using a Java client library

The following Java code example adds a form design (an XDP file) to the repository by using the Repository service's Java client library.

```
import java.io.FileInputStream;
import java.util.Properties;
import com.adobe.idp.Document;
import com.adobe.idp.dsc.clientsdk.ServiceClientFactory;
import com.adobe.repository.bindings.dsc.client.ResourceRepositoryClient;
import com.adobe.repository.infomodel.Id;
import com.adobe.repository.infomodel.Lid;

import com.adobe.repository.infomodel.bean.RepositoryInfomodelFactoryBean;
import com.adobe.repository.infomodel.bean.Resource;
import com.adobe.repository.infomodel.bean.ResourceContent;


public class UploadForm {

   public static void main(String[] args) {

      try
      {
      //This example will upload an XDP file to the LiveCycle Repository
      //Set LiveCycle ES service connection properties
      Properties connectionProps = new Properties();
      connectionProps.setProperty("DSC_DEFAULT_EJB_ENDPOINT",
"jnp://localhost:1099");
      connectionProps.setProperty("DSC_TRANSPORT_PROTOCOL","EJB");
      connectionProps.setProperty("DSC_SERVER_TYPE", "JBoss");
      connectionProps.setProperty("DSC_CREDENTIAL_USERNAME", "administrator");
      connectionProps.setProperty("DSC_CREDENTIAL_PASSWORD", "password");

      ServiceClientFactory myFactory =
ServiceClientFactory.createInstance(connectionProps);

      //Create a ResourceRepositoryClient object
      ResourceRepositoryClient repositoryClient = new
ResourceRepositoryClient(myFactory);

      //Specify the parent path
```

```
      String parentResourcePath = "/";

     //Create a RepositoryInfomodelFactoryBean object
      RepositoryInfomodelFactoryBean infomodelFactory = new
RepositoryInfomodelFactoryBean(null);

     //Create a Resource object to add to the Repository
      Resource newResource = (Resource) infomodelFactory.newImage(
                  new Id(),
                  new Lid(),
                  "Loan.xdp");

     //Create a ResourceContent object that contains the content (file bytes)
     ResourceContent content = (ResourceContent)
infomodelFactory.newResourceContent();

     //Create a Document that references an XDP file
     //to add to the Repository
     FileInputStream myForm = new FileInputStream("C:\\Adobe\\Loan.xdp");
     Document form = new Document(myForm);

     //Set the description and the MIME type
     content.setDataDocument(form);
     content.setMimeType("application/vnd.adobe.xdp+xml");

     //Assign content to the Resource object
     newResource.setContent(content) ;

     //Set a description of the resource
     newResource.setDescription("An XDP file");

     //Commit to repository, and update resource
     //in memory (by assignment)
     Resource addResource =
repositoryClient.writeResource(parentResourcePath, newResource);

     //Get the description of the returned Resource object
     System.out.println("The description of the new resource is
"+addResource.getDescription());

     //Close the FileStream object
     myForm.close();

     } catch (Exception e) {

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

Quick Start: Invoking a service using base64 in a Microsoft .NET project

The following C# code example invokes a process named EncryptDocument from a Microsoft .NET project using Base64 encoding. (See Invoking LiveCycle ES using Base64 Encoding.)

An unsecured PDF document based on a PDF file named map.pdf is passed to the LiveCycle ES process. The process returns a password-encrypted PDF document that is saved as a PDF file named mapEncrypt.pdf.

```
/*
 * Ensure that you create a .NET client assembly that uses
 * base64 encoding. This is required to populate a BLOB
 * object with data or retrieve data from a BLOB object.
 *
 * For information, see "Invoking LiveCycle ES using Base64 Encoding" in
 * Programming with LiveCycle ES
 */
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;

namespace ConsoleApplication1
{

    class InvokeEncryptDocumentUsingBase64
    {

        const int BUFFER_SIZE = 4096;
        [STAThread]
        static void Main(string[] args)
        {

            try
            {
                String pdfFile = "C:\\Adobe\\map.pdf";
                String encryptedPDF = "C:\\Adobe\\mapEncrypt.pdf";

                //Create an EncryptDocumentServiceWse object and set
authentication values
                EncryptDocumentService encryptClient = new
EncryptDocumentService();
                encryptClient.Credentials = new
System.Net.NetworkCredential("administrator", "password");

                //Reference the PDF file to send to the EncryptDocument process
                FileStream fs = new FileStream(pdfFile, FileMode.Open);

                //Create a BLOB object
                BLOB inDoc = new BLOB();

                //Get the length of the file stream
                int len = (int)fs.Length;
                byte[] ByteArray = new byte[len];
```

```
                //Populate the byte array with the contents of the FileStream
object
                fs.Read(ByteArray, 0, len);
                inDoc.binaryData = ByteArray;

                //Invoke the EncryptDocument process
                BLOB outDoc = encryptClient.invoke(inDoc);

                //Populate a byte array with BLOB data
                byte[] outByteArray = outDoc.binaryData;

                //Create a new file named UsageRightsLoan.pdf
                FileStream fs2 = new FileStream(encryptedPDF,
FileMode.OpenOrCreate);

                //Create a BinaryWriter object
                BinaryWriter w = new BinaryWriter(fs2);
                w.Write(outByteArray);
                w.Close();
                fs2.Close();
             }
            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
        }
    }
}
```

Quick Start: Invoking a service using Axis-generated files that use Base64 encoding

The following Java code example invokes a process named EncryptDocument using Axis-generated files that use Base64 encoding. (See Invoking LiveCycle ES using Base64 Encoding.)

An unsecured PDF document based on a PDF file named map.pdf is passed to the LiveCycle ES process. The process returns a password-encrypted PDF document that is saved as a PDF file named mapEncrypt.pdf.

```
/*
 * This Java Quick Start uses axis generated Java files and
 * base64 encoding.)
 * For complete details,
 * see "Invoking LiveCycle ES using Base64 Encoding" in
 * Programming with LiveCycle ES
 */
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import com.adobe.idp.services.BLOB;
import com.adobe.idp.services.EncryptDocument;
import com.adobe.idp.services.EncryptDocumentServiceLocator;

public class InvokeDocumentEncryptBase64 {
```

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

     try{

        String pdfFile = "C:\\Adobe\\map.pdf";
           String encryptedPDF = "C:\\Adobe\\mapEncrypt.pdf";

        //create a service locator
        EncryptDocumentServiceLocator locate = new
EncryptDocumentServiceLocator();

        //specify the service target URL and object type
        EncryptDocument encryptionClient = locate.getEncryptDocument();

        //Use the binding stub with the locator

((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.USERN
AME_PROPERTY, "administrator");

((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.PASSW
ORD_PROPERTY, "password");

        //Reference the PDF document to pass to the EncrptDocuemnt process
        FileInputStream file = new FileInputStream(pdfFile);

        //Create a byte array to store the PDF document
        int len = file.available();
        byte [] myByteArray = new byte[len];
        int i = 0;
        while (i < len) {
            i += file.read(myByteArray, i, len);
        }

        //Create a BLOB object and populate it with the byte array
        BLOB inDoc = new BLOB();
        inDoc.setBinaryData(myByteArray);

        //Invoke the EncryptDocument process
        BLOB outDoc = encryptionClient.invoke(inDoc);

        //Populate a byte array with the encrypted PDF document
        byte[] myFile = outDoc.getBinaryData();

        //Create a File object
        File outFile = new File(encryptedPDF);

        //Create a FileOutputStream object.
        FileOutputStream myFileW = new FileOutputStream(outFile);

        //Call the FileOutputStream object's write method and pass the pdf data
        myFileW.write(myFile);

        //Close the FileOutputStream object
        myFileW.close();
```

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

}
```

Quick Start: Invoking the Repository service using LiveCycle Remoting

The following Flex code example adds a form design (an XDP file) to the repository by using LiveCycle Remoting.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
creationComplete="initializeChannelSet();">
  <mx:Script>
    <![CDATA[

  import mx.rpc.livecycle.DocumentReference;
  import flash.net.FileReference;
  import flash.events.Event;
  import flash.events.DataEvent;
  import mx.messaging.ChannelSet;
  import mx.messaging.channels.AMFChannel;
  import mx.rpc.events.ResultEvent;

  private var fileRef:FileReference = new FileReference();
  private var docRef:DocumentReference = new DocumentReference();
  private var parentResourcePath:String = "/";
  private var serverPort:String = "servername:8080";

  // Set up channel set to talk to LiveCycle.
  // This must be done before calling any service or process, but only once for
the entire application.
  // Note that this uses runtime configuration to configure the destination
correctly, so
  // no other setup is needed in remoting-config.xml.
  private function initializeChannelSet():void {
    var cs:ChannelSet= new ChannelSet();
    cs.addChannel(new AMFChannel("remoting-amf", "http://" + serverPort +
"/remoting/messagebroker/amf"));
    repositoryService.setCredentials("administrator", "password");
    repositoryService.channelSet = cs;
  }

  // Call this method to upload the file to be added to the repository.
  // This creates a file picker and lets the user select the file to upload.
  private function uploadFile():void {
    fileRef.addEventListener(Event.SELECT, selectHandler);
    fileRef.addEventListener(DataEvent.UPLOAD_COMPLETE_DATA,completeHandler);
    fileRef.browse();
  }
```

```
  // Gets called for selected file. Does the actual upload via our file upload
servlet.
  private function selectHandler(event:Event):void {
    var request:URLRequest = new URLRequest("http://" + serverPort +
"/remoting/lcfileupload");
    fileRef.upload(request);
  }

  // Called once the file is completely uploaded. Now it is safe to access that
object for other things.
  private function completeHandler(event:DataEvent):void {
    var params:Object = new Object();
    docRef.url = event.data as String;
    docRef.referenceType=DocumentReference.REF_TYPE_URL;
    // At this point we can do whatever we want with the file that has been
uploaded.
    // Our docRef variable has the object. Refer to the asdoc for
DocumentReference for methods and properties.
    // Note that the url is publicly accessible at this point. Useful for
testing purposes.
    writeResource();
  }

  // Uses RepositoryService API to write resource to repository. Sets the name
to "resource.xdp" right now.
  // Note that the destination name of the RemoteObject below is the Endpoint
name in the adminui.
  private function writeResource():void {
    var resource:Object = new Object();
    var content:Object = new Object();

    // define resource
    resource.name = "resource.xdp";
    content.dataDocument = docRef;
    content.mimeType = "application/vnd.adobe.xdp+xml";
    resource.content = content;
    resource.description = "An XDP File";

    // call to remote object

repositoryService.writeResource({"parentResourcePath":parentResourcePath,"re
source":resource});
  }

  private function resultHandler(event:ResultEvent):void {
    // Do anything else here.
  }

    ]]>
  </mx:Script>

  <mx:RemoteObject id="repositoryService" destination="RepositoryService"
result="resultHandler(event);"/>
```

```
  <mx:Panel id="lcPanel" title="Repository Service LiveCycle Remoting Example"
      height="25%" width="25%" paddingTop="10" paddingLeft="10"
paddingRight="10" paddingBottom="10">
     <mx:Label width="100%" color="blue"
          text="Select a PDF file. This example automatically uploads it to the
repository."/>
    <mx:Button label="Select and Upload" click="uploadFile()" />
  </mx:Panel>
</mx:Application>
```

Quick Start: Invoking a service using DIME in a .NET project

The following C# code example invokes a process named EncryptDocument from a Microsoft .NET project using Dime. (See Invoking LiveCycle ES using DIME.)

An unsecured PDF document based on a PDF file named map.pdf is passed to the LiveCycle ES process using DIME. The process returns a password-encrypted PDF document that is saved as a PDF file named mapEncrypt.pdf.

```
/**
 *
 * Ensure that you create a .NET project that uses
 * Web Services Enhancements 2.0. This is required to send a
 * LiveCycle ES process an attachment using DIME.
 *
 * For information, see "Invoking LiveCycle ES using DIME" in Programming with
LiveCycle ES.
 */

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;
using Microsoft.Web.Services2.Dime;
using Microsoft.Web.Services2.Attachments;
using Microsoft.Web.Services2.Configuration;
using Microsoft.Web.Services2;

//The following statement represents a web reference to
//the LiveCycle ES server that contains the process that
//is invoked
using ConsoleApplication1.LC_Host;

namespace ConsoleApplication1
{

  class InvokeEncryptDocumentUsingDime
    {

        const int BUFFER_SIZE = 4096;
        [STAThread]
        static void Main(string[] args)
        {
```

```
        try
           {
               String pdfFile = "C:\\Adobe\\map.pdf";
               String encryptedPDF = "C:\\Adobe\\mapEncrypt.pdf";

               //Create an EncryptDocumentServiceWse object and set
authentication values
               EncryptDocumentServiceWse encryptClient = new
EncryptDocumentServiceWse();
               encryptClient.Credentials = new
System.Net.NetworkCredential("administrator", "password");

               // Create the DIME attachment representing a PDF document
               DimeAttachment inputDocAttachment = new DimeAttachment(
                   System.Guid.NewGuid().ToString(),
                   "application/pdf",
                   TypeFormat.MediaType,
                   pdfFile);

                //Create a BLOB object
                BLOB inDoc = new BLOB();

                //Set the DIME attachment ID
                inDoc.attachmentID = inputDocAttachment.Id;

encryptClient.RequestSoapContext.Attachments.Add(inputDocAttachment);

                //Invoke the EncryptDocument process
                BLOB outDoc = encryptClient.invoke(inDoc);

                //Get the returned attachment identifier value
                String encryptedDocId = outDoc.attachmentID;
                FileStream myStream = new FileStream(encryptedPDF,
FileMode.Create, FileAccess.Write);

                //Iterate through the attachments
                foreach (Attachment attachment in
encryptClient.ResponseSoapContext.Attachments)
                  {
                    if (attachment.Id.Equals(encryptedDocId))
                      {
                        //Create a byte array that contains the encrypted PDF
document
                        System.IO.Stream mySteam2 = attachment.Stream;
                        byte[] myBytes = new byte[mySteam2.Length];
                        int size = (int)mySteam2.Length;
                        mySteam2.Read(myBytes, 0, size);

                        //Save the encrypted PDF document as a PDF file
                        FileStream fs2 = new FileStream(encryptedPDF,
FileMode.OpenOrCreate);

                        //Create a BinaryWriter object
                        BinaryWriter w = new BinaryWriter(fs2);
                        w.Write(myBytes);
```

```
                    w.Close();
                    fs2.Close();
                    Console.Out.WriteLine("Saved converted document at:" +
encryptedPDF);
                }
            }
          }
          catch (Exception ee)
          {
             Console.WriteLine(ee.Message);
          }
       }
    }
}
```

Quick Start: Invoking a service using DIME in a Java project

The following Java code example invokes a process named EncryptDocument using DIME. (See Invoking LiveCycle ES using DIME.)

An unsecured PDF document based on a PDF file named Loan.pdf is passed to the LiveCycle ES process using DIME. The process returns a password-encrypted PDF document that is saved as a PDF file named EncryptLoan.pdf.

```
/**
 * Ensure that you create Java Axis files that
 * are required to send a LiveCycle ES process
 * an attachment using DIME.
 *
 * For information, see "Invoking LiveCycle ES using DIME" in Programming with
LiveCycle ES.
 */
import com.adobe.idp.services.*;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import javax.activation.DataHandler;
import javax.activation.FileDataSource;

import org.apache.axis.attachments.AttachmentPart;

public class InvokeDocumentEncryptDime {
   public static void main(String[] args) {

   try{

      //create a service locator
      EncryptDocumentServiceLocator locate = new
EncryptDocumentServiceLocator();

      //specify the service target URL and object type
      URL serviceURL = new
URL("http://localhost:8080/soap/services/EncryptDocument?blob=dime");
```

```
     //Use the binding stub with the locator
     EncryptDocumentSoapBindingStub encryptionClientStub = new
EncryptDocumentSoapBindingStub(serviceURL,locate);
     encryptionClientStub.setUsername("administrator");
     encryptionClientStub.setPassword("password");

     //Get the DIME Attachments - which is the PDF document to encrypt
     java.io.File file = new java.io.File("C:\\Adobe\\Loan.pdf");

     //Create a DataHandler object
     DataHandler buildFile = new DataHandler(new FileDataSource(file));

     //Use the DataHandler object to create an AttachmentPart object
     AttachmentPart part = new AttachmentPart(buildFile);
     //get the attachment ID
     String attachmentID = part.getContentId();

     //Add the attachment to the encryption service stub
     encryptionClientStub.addAttachment(part);

     //Inform ES where the attachment is stored by providing the attachment id
     BLOB inDoc = new BLOB();
     inDoc.setAttachmentID(attachmentID);

     BLOB outDoc = encryptionClientStub.invoke(inDoc);

     //Go through the returned attachments and get the encrypted PDF document
     byte[] resultByte = null;
     attachmentID = outDoc.getAttachmentID();

     //Find the proper attachment
     Object[] parts = encryptionClientStub.getAttachments();
     for (int i=0;i<parts.length;i++){
       AttachmentPart attPart = (AttachmentPart)  parts[i];
       if (attPart.getContentId().equals(attachmentID)) {
         //DataHandler
         buildFile = attPart.getDataHandler();
         InputStream stream = buildFile.getInputStream();

         byte[] pdfStream = new byte[stream.available()];
         stream.read(pdfStream);

         //Create a File object
         File outFile = new File("C:\\Adobe\\EncryptLoan.pdf");

         //Create a FileOutputStream object.
         FileOutputStream myFileW = new FileOutputStream(outFile);

         //Call the FileOutputStream object's write method and pass the pdf
data
         myFileW.write(pdfStream);

         //Close the FileOutputStream object
         myFileW.close();
```

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

}
```

Quick Start: Invoking a service using BLOB data over HTTP in a Java project

The following Java code example invokes a process named EncryptDocument using data over HTTP. (See Invoking LiveCycle ES using BLOB Data over HTTP.)

An unsecured PDF document based on a PDF file named Loan.pdf is passed to the LiveCycle ES process using SOAP over HTTP. The process returns a password-encrypted PDF document that is saved as a PDF file named EncryptLoan.pdf.

```
/*
 * This Java Quick Start uses axis generated Java files and
 * SOAP over HTTP.
 * For complete details,
 * see "Invoking LiveCycle ES using SOAP over HTTP" in
 * Programming with LiveCycle ES
 */
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import com.adobe.idp.services.BLOB;
import com.adobe.idp.services.EncryptDocument;
import com.adobe.idp.services.EncryptDocumentServiceLocator;

public class InvokeDocumentEncryptHTTP {

   public static void main(String[] args) {

   try{

      //Create a service locator
      EncryptDocumentServiceLocator locate = new
EncryptDocumentServiceLocator();

      //Create an EncryptDocument object
      EncryptDocument encryptionClient = locate.getEncryptDocument();

      //Use the binding stub with the locator

((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.USERN
AME_PROPERTY, "administrator");
```

```
((javax.xml.rpc.Stub)encryptionClient)._setProperty(javax.xml.rpc.Stub.PASSW
ORD_PROPERTY, "password");

     //Create a BLOB object and populate it by invoking the setRemoteURL method
     BLOB inDoc = new BLOB();
     inDoc.setRemoteURL("http://localhost:8080/WebApplication/Loan.pdf");

     //Invoke the EncryptDocument process
     BLOB outDoc = encryptionClient.invoke(inDoc);

     //Retrieve an InputStream from the returned BLOB instance
     URL myURL = new URL(outDoc.getRemoteURL());
     InputStream stream = myURL.openStream();

     //Create a byte array and populate it with stream data
     byte[] pdfStream = new byte[stream.available()];
     int offset = 0;
       int remaining = stream.available();
       while (remaining > 0)
        {
            int read = stream.read(pdfStream , offset, remaining);
            remaining -= read;
            offset += read;
        }

     //Create a File object
     File outFile = new File("C:\\Adobe\\EncryptLoan.pdf");

     //Create a FileOutputStream object.
     FileOutputStream myFileW = new FileOutputStream(outFile);

     //Call the FileOutputStream object's write method and pass the pdf data
     myFileW.write(pdfStream);

     //Close the FileOutputStream object
     myFileW.close();
   }
   catch(Exception e)
   {
     e.printStackTrace();
   }
  }
}
```

Quick Start: Invoking a service using BLOB data over HTTP in a .NET project

The following C# code example invokes a process named EncryptDocument from a Microsoft .NET project using data over HTTP. (See Invoking LiveCycle ES using BLOB Data over HTTP.)

An unsecured PDF document based on a PDF file named Loan.pdf is passed to the LiveCycle ES process using SOAP over HTT. The process returns a password-encrypted PDF document that is saved as a PDF file named EncryptedPDF.pdf.

```
/*
 * Ensure that you create a .NET client assembly that uses
 * SOAP over HTTP. This is required to populate a BLOB
 * object's remote URL data memeber.
 *
 * For information, see "Invoking LiveCycle ES using SOAP over HTTP" in
 * Programming with LiveCycle ES
 */
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.IO;
using System.Security.Policy;

namespace ConsoleApplication1
{

    class InvokeEncryptDocumentUsingHTTP
    {

        const int BUFFER_SIZE = 4096;
        [STAThread]
        static void Main(string[] args)
        {

            try
            {
                String urlData =
"http://localhost:8080/WebApplication/Loan.pdf";

                //Create an EncryptDocumentServiceWse object and set
authentication values
                EncryptDocumentService encryptClient = new
EncryptDocumentService();
                encryptClient.Credentials = new
System.Net.NetworkCredential("administrator", "password");

                //Create a BLOB object
                BLOB inDoc = new BLOB();

                //Populate the BLOB object's remoteURL data member
                inDoc.remoteURL = urlData;

                //Invoke the EncryptDocument process
                BLOB outDoc = encryptClient.invoke(inDoc);

                //Create a UriBuilder object using the
                //BLOB object's remoteURL data member field
                UriBuilder uri = new UriBuilder(outDoc.remoteURL);

               //Convert the UriBuilder to a Stream object
                System.Net.WebRequest wr =
System.Net.WebRequest.Create(uri.Uri);
                System.Net.WebResponse response = wr.GetResponse();
```

```
                System.IO.StreamReader sr = new
System.IO.StreamReader(response.GetResponseStream());
                Stream mySteam = sr.BaseStream;

                //Create a byte array
                byte[] myData = new byte[BUFFER_SIZE];

               //Populate the byte array
               PopulateArray(mySteam, myData);

               //Create a new file named UsageRightsLoan.pdf
               FileStream fs2 = new FileStream("C:\\Adobe\\EncryptedPDF.pdf",
FileMode.OpenOrCreate);

               //Create a BinaryWriter object
               BinaryWriter w = new BinaryWriter(fs2);
               w.Write(myData);
               w.Close();
               fs2.Close();
            }
            catch (Exception ee)
            {
                Console.WriteLine(ee.Message);
            }
        }

        public static void PopulateArray(Stream stream, byte[] data)
        {
            int offset = 0;
            int remaining = data.Length;
            while (remaining > 0)
            {
                int read = stream.Read(data, offset, remaining);
                if (read <= 0)
                    throw new EndOfStreamException();
                remaining -= read;
                offset += read;
            }
        }

    }
}
```