Developing Custom Applications

Adobe® LiveCycle™ Policy Server

Version 7.0

Adobe® LiveCycle™ Policy Server 7.0 Developing Custom Applications for Microsoft® Windows® and UNIX®
December 2004

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

3 Working with Policies (Continued)

List of Examples

Preface

This guide provides information about how to use the Adobe® LiveCycle™ Policy Server SDK to develop custom Policy Server client applications.

What's in this guide?

This guide contains the following information:

- Requirements for setting up the development environment.
- How to use the API to programmatically interact with Policy Server to create and manage Policy Server collateral, such as policies, watermarks, and principals.
- Code examples that show how to achieve specific tasks.

Note: The SDK also includes Java libraries for developing custom service providers that integrate with Policy Server. Although this guide does not describe how to use these libraries, the API Reference includes information about them. For information about developing custom service providers, contact Adobe Customer Support.

Who should read this guide?

Java developers who want to create custom Policy Server client applications should read this guide. To use this guide, you should be familiar with the Java programming language.

Related documentation

You can use other product documentation to learn more about Policy Server:

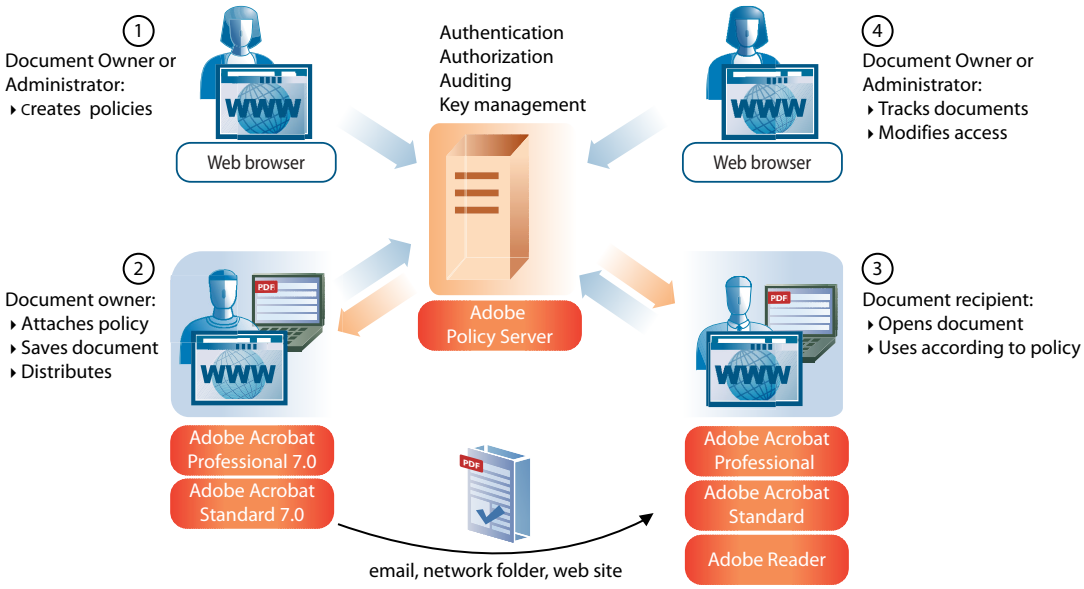
For information about	See
Changes to the product that occurred late in the development cycle	Policy Server Readme
The Policy Server SDK API	API Reference
How to install Policy Server	*Installing and Configuring* guide
Policy Server features and security information	*Overview* guide
How to use the Policy Server administrator and user features	Policy Server Help
Other Adobe LiveCycle products	http://www.adobe.com

1 Introduction

This chapter provides a description of the type of software that you can develop using the Policy Server SDK API. Also included are descriptions of the Java libraries that you use.

Policy Server is a web-based security system that enables users to dynamically apply confidentiality settings to their PDF documents and maintain control over the documents no matter how widely users distribute them.

Policy Server consists of several components, including a server, client applications, and an SDK. The SDK enables Java developers to create applications that access the server component. The public API included with Policy Server provide the tools required to perform most of the tasks that the Policy Server web applications perform.



Note: See the *Overview* guide for more information about the Policy Server features and architecture.

Java libraries

The Policy Server SDK includes two Java libraries that enable you to programmatically interact with Policy Server:

- The com.adobe.edc.sdk package provides classes and interfaces for connecting to Policy Server and instantiating managers for manipulating Policy Server collateral, such as licenses, policies, and watermarks.
- The com.adobe.edc.sdk.infomodel package provides classes and interfaces for creating and manipulating objects that represent Policy Server collateral, such as licenses, policies, and watermarks.

Note: The SDK also includes Java libraries for developing custom service providers that integrate with Policy Server. Although this guide does not describe how to use these libraries, the API Reference includes information about them. For information about developing custom service providers, contact Adobe Customer Support.

2 Invoking Policy Server

You use the Policy Server API to create custom applications capable of interacting with Policy Server. For example, using the Policy Server API, you can create an application that dynamically creates new policies and uses them to secure PDF documents. This chapter explains how to invoke Policy Server using the Policy Server API.

The Policy Server API is implemented in Java and has public methods that enable you to invoke Policy Server. Using a Java development environment, you can use the static `EDCFactory` object to connect to Policy Server. For information about this object, see the API Reference.

In addition to using the Policy Server API, you must also use standard Java classes. The examples in this chapter describe how to use the Policy Server API and standard Java classes to invoke Policy Server.

This chapter contains the following information:

Including the API library files

The Policy Server API consists of many JAR files that you must set in your application's class path. If you do not reference these JAR files, you cannot use the Policy Server API in your Java project. These JAR files are installed along with the Policy Server SDK.

Two sets of JAR files are available that you can use. One set is used to create custom applications that interact with Policy Server deployed on IBM® WebSphere®. The other set is used to create custom applications that interact with Policy Server deployed on JBoss. You must use the set of JAR files that correspond to the J2EE application server on which Policy Server is deployed.

- If you are developing for the JBoss version of Policy Server, the JAR files are in this location:

  <*install directory*>/PolicyServer_SDK/sdk/lib/JBoss

- If you are developing for the WebSphere version of Policy Server, the JAR files are in this location:

  <*install directory*>/PolicyServer_SDK/sdk/lib/WebSphere

JBoss API library files

The following table lists the JAR files that you must include in your application's class path if Policy Server is deployed on JBoss:

asn1.jar	axis.jar	commons-discovery.jar	edc-sdk.jar	jaxb-api.jar
jaxb-impl.jar	jaxb-libs.jar	commons-logging.jar	jax-qname.jar	jaxrpc.jar
jsafe.jar	jsafeJCE.jar	log4j.jar	wss4j.jar	opensaml.jar
saaj.jar	sdk-ejb-client.jar	relaxngDatatype.jar	wsdl4j.jar	
xmlsec-1.0.5.jar	xsdlib.jar	JBossall-client.jar	jndi.properties	

You must also ensure that your application can access the jar files that are installed in the <*install directory*>/PolicyServer_SDK/sdk/lib/JBoss/lib/Endorsed directory. To do this, you can perform either of the following tasks:

- Copy the contents of the Endorsed directory to the %JAVA_HOME%/jre/lib/endorsed directory.
- Set the system property in your application to point to the location of the files in Endorsed (-Djava.endorsed.dirs=<*location of jar files*>).

The Endorsed directory includes these files:

- namespace.jar
- dom3xerceslmpl-2.4.0.jar
- dom3-xml-apis-2.4.0.jar
- xalan.jar

Note: If you are connecting to Policy Server in EJB mode, you also need to add the file *appserver root*/client/jbossall-client.jar to your application's class path, where appserver root is the application server directory. For more information about EJB mode, see "Connecting to Policy Server" on page 12.

WebSphere API library files

The following table lists the JAR files that you must include in your application's class path if Policy Server is deployed on WebSphere:

asn1.jar	axis.jar	commons-discovery.jar	edc-sdk.jar	jaxb-api.jar
jaxb-impl.jar	jaxb-libs.jar	commons-logging.jar	jax-qname.jar	jaxrpc.jar
jsafe.jar	jsafeJCE.jar	log4j.jar	opensaml.jar	
saaj.jar	sdk-ejb-client.jar	relaxngDatatype.jar	wsdl4j.jar	
xmlsec-1.0.5.jar	xsdlib.jar	jndi.properties	wss4j.jar	

You must also ensure that your application can access the jar files that are located in the <*install directory*>/PolicyServer_SDK/sdk/lib/WebSphere/lib/Endorsed directory. To do this, you can perform either of the following tasks:

- Copy the contents of the Endorsed directory to the %JAVA_HOME%/jre/lib/endorsed directory.
- Set the system property in your application to point to the location of the files in Endorsed (-Djava.endorsed.dirs=<*location of jar files*>).

Note: If you are using WebSphere, the value of the JAVA_HOME environment variable should be the Java directory that is installed with WebSphere.

This Endorsed directory includes these files:

- namespace.jar
- dom3xerceslmpl-2.4.0.jar
- dom3-xml-apis-2.4.0.jar
- xalan.jar

Adding import statements

The Policy Sever API consists of different packages. You must add the following import statements to your Java project to successfully use the Policy Server API:

```
import com.adobe.edc.sdk.*;
import com.adobe.edc.sdk.infomodel.* ;
```

Note: For information about these Java packages, see the API Reference.

Connecting to Policy Server

A custom application must connect to Policy Server before the application can interact with it. You connect to Policy Server by using the `EDCFactory` object's `connect` method. This method must be called from within a try statement.

The `connect` method requires a Java `Properties` object as an argument that specifies a value for some or all of the following properties:

- EDCFactory.USERNAME_PROPERTY_NAME - The user name used to connect
- EDCFactory.PASSWORD_PROPERTY_NAME - The corresponding password
- EDCFactory.MODE_PROPERTY_NAME - The mode used to connect to Policy Server
- EDCFactory.URL_PROPERTY_NAME - The Policy Server URL, required only when using SOAP

Note: EDCFactory defines other properties that require values when using document managers. For more information, see "Creating a DocumentManager object" on page 15.

The properties belong to the static `EDCFactory` object. Because this object is static, you do not have to instantiate it to call the `connect` method. This method returns a non-static `EDCFactory` object that you use to perform other tasks, such as creating Policy Server manager objects. The non-static `EDCFactory` object represents a session with Policy Server.

You create a Java `Properties` object by using its constructor. Call the `Properties` object's `setProperty` method to assign a value to each property.

When setting the `MODE_PROPERTY_NAME` property, you can specify the SOAP mode or the EJB mode. The performance of the EJB mode is better than the performance of the SOAP mode. As a result, it is recommended that you use the EJB mode if the custom application and Policy Server are located within the same firewall. The EJB mode uses the RMI/IIOP protocol. When using the EJB mode on JBoss, you must include the JBossall-client.jar file in your application's class path.

However, if a firewall is located between Policy Server and the custom application, it is recommended that you use the SOAP mode. This mode uses http(s) as the underlying transport and is able to communicate across firewall boundaries.

Connecting to Policy Server using SOAP

If you use the SOAP mode, set the `EDCFactory.URL_PROPERTY_NAME` property to

```
http://<ServerName>:<Port>/edcws/services/EDCPolicyService?wsdl
```

where `ServerName` is the name of the J2EE application server on which Policy Server is deployed and `Port` is the port that the J2EE application server uses. It is not necessary to set this property if you use the EJB mode.

The following code example connects to Policy Server using the SOAP mode.

Example 2.1 ***Connecting to Policy Server using SOAP***

```
// Create a Java Properties object
Properties apsProperty = new Properties();

//Specify property values
apsProperty.setProperty(EDCFactory.USERNAME_PROPERTY_NAME,"<user name>");
apsProperty.setProperty(EDCFactory.PASSWORD_PROPERTY_NAME,"<password>");
apsProperty.setProperty(EDCFactory.URL_PROPERTY_NAME,http://<ServerName>:<Po
rt>/edcws/services/EDCPolicyService?wsdl);
apsProperty.setProperty(EDCFactory.MODE_PROPERTY_NAME,"soap");

try
{
   //Establish a connection to Policy Server
   EDCFactory apsSession = EDCFactory.connect(apsProperty);
   //Perform additional tasks using apsSession
}
catch (Exception ex)
   System.out.println("The exception is " +ex.getMessage());
```

Note: To successfully instantiate a Java `Properties` object, add the following import statement to your Java project: `import java.util.*`.

Connecting to Policy Server using EJB

If you use the EJB mode to connect to Policy Server, it is unnecessary to set the `EDCFactory.URL_PROPERTY_NAME` property. However, you must configure the jndi.properties file by specifying the URL of Policy Server. You must also ensure that this file is referenced in your run-time environment.

The jndi.properties file is installed along with the JAR files that you must include in your project's class path. For information about the location of these files, see the *Installing and Configuring* guide.

The following code example connects to Policy Server using the EJB mode.

Example 2.2 ***Connecting to Policy Server using EJB***

```
// Create a Java Properties object
Properties apsProperty = new Properties();

//Specify property values
apsProperty.setProperty(EDCFactory.USERNAME_PROPERTY_NAME,"<user name>");
apsProperty.setProperty(EDCFactory.PASSWORD_PROPERTY_NAME,"<password>");
apsProperty.setProperty(EDCFactory.MODE_PROPERTY_NAME,"ejb");

try
{
   //Establish a connection to Policy Server
   EDCFactory apsSession = EDCFactory.connect(apsProperty);
   //Perform additional tasks using apsSession
}
catch (Exception ex)
   System.out.println("The exception is " +ex.getMessage());
```

Creating Policy Server manager objects

The Policy Server API contains different interfaces that enable you to manage Policy Server resources. For example, you can create a `PolicyManager` object, which is an instance of the `PolicyManager` interface, to create and mange policies.

Using a non-static `EDCFactory` object, you can create the following objects:

- PolicyManager
- DocumentManager
- EventManager
- LicenseManager
- UserManager
- WatermarkManager

Note: You cannot create these objects using a static `EDCFactory` object. You must use an `EDCFactory` object that is returned by the `connect` method. For information, see "Connecting to Policy Server" on page 12.

Creating a PolicyManager object

A `PolicyManager` object can be created by calling the `EDCFactory` object's `getPolicyManager` method. This method returns an instance of a `PolicyManager` interface. You use a `PolicyManager` object to manage policies. For information about policies, see "Working with Policies" on page 18.

The following code example creates a `PolicyManager` object.

Example 2.3 *Creating a PolicyManager object*

```
//Establish a connection to Policy Server
EDCFactory apsSession = EDCFactory.connect(apsProperty);

//Create a PolicyManager object
PolicyManager apsPolicyManager = apsSession.getPolicyManager();
```

Creating a DocumentManager object

A `DocumentManager` object can be created by calling the `EDCFactory` object's `getDocumentManager` method. You use a `DocumentManager` object to manage documents. For information, see "Managing Documents" on page 36.

To successfully create a `DocumentManager` object, you must set the static `EDCFactory` object's `PACKAGER_EXECUTABLE_PATH` property prior to calling the `connect` method. Set this property with the location of an additional security component.

The following code example creates a `DocumentManager` object.

Example 2.4 *Creating a DocumentManager object*

```
//Establish a connection to Policy Server
apsProperty.setProperty(EDCFactory.PACKAGER_EXECUTABLE_PATH,"path");
EDCFactory apsSession = EDCFactory.connect(apsProperty);

//Create a DocumentManager object
DocumentManager apsDocManager = apsSession.getDocumentManager();
```

Note: The security component is not part of Policy Server. For information about the security component, contact Adobe Customer Support.

Creating an EventManager object

An `EventManager` object can be created by calling the `EDCFactory` object's `getEventManager` method. You use an `EventManager` object to work with events that are supported by Policy Server. For information, see "Registering Event Handlers" on page 52.

The following code example creates an `EventManager` object.

Example 2.5 *Creating an EventManager object*

```
//Establish a connection to Policy Server
EDCFactory apsSession = EDCFactory.connect(apsProperty);

//Create an EventManager object
EventManager apsEventManager = apsSession.getEventManager();
```

Creating a LicenseManager object

A `LicenseManager` object can be created by calling the `EDCFactory` object's `getLicenseManager` method. You use a `LicenseManager` object to manage Policy Server licenses. For example, using a `LicenseManager` object, you can revoke a license from a policy-protected document, resulting in the document being inaccessible. For information, see "Revoking documents" on page 39.

The following code example creates a `LicenseManager` object.

Example 2.6 ***Creating a LicenseManager object***

```
//Establish a connection to Policy Server
EDCFactory apsSession = EDCFactory.connect(apsProperty);

//Create a LicenseManager object
LicenseManager apsLicenseManager = apsSession.getLicenseManager();
```

Creating a UserManager object

A `UserManager` object can be created by calling the `EDCFactory` object's `getUserManager` method. You use a `UserManager` object to manage Policy Server external users. For information, see "Working with Policy Server Principals" on page 33.

The following code example creates a `UserManager` object.

Example 2.7 ***Creating a UserManager object***

```
//Establish a connection to Policy Server
EDCFactory apsSession = EDCFactory.connect(apsProperty);

//Create a UserManager object
UserManager apsUserManager = apsSession.getUserManager();
```

Creating a WatermarkManager object

A `WatermarkManager` object can be created by calling the `EDCFactory` object's `getWatermarkManager` method. You use a `WatermarkManager` object to work with watermarks. For example, you can insert a watermark into a document and set the text. For information, see "Working with Watermarks" on page 44.

The following code example creates a `WatermarkManager` object.

Example 2.8 ***Creating a WatermarkManager object***

```
//Establish a connection to Policy Server
EDCFactory apsSession = EDCFactory.connect(apsProperty);

// Create a WatermarkManager object
WatermarkManager apsWatermarkManager = apsSession.getWatermarkManager();
```

Working with InfomodelObjectFactory objects

You use an `InfomodelObjectFactory` object to create Policy Server objects. Using this object, you can perform the following tasks:

- Create a Policy object
- Create a License object.
- Create a Permission object
- Create a PolicyEntry object
- Create a ValidityPeriod object
- Create a Watermark object
- Create a special Principal object

An `InfomodelObjectFactory` object is a static object and as a result, you do not have to instantiate it. Other sections of this guide discuss how to use an `InfoModelObjectFactory` object to perform tasks such as creating a `Policy` object. For information, see "Creating a Policy object" on page 19.

Disconnecting from Policy Server

You can close a connection to Policy Server by calling the `EDCFactory` object's `closeConnection` method. Once you call this method, your application cannot interact with Policy Server. To interact with Policy Server, you must connect to it. For information, see "Connecting to Policy Server" on page 12.

If an application has more than one `EDCFacotry` object that represents a connection to Policy Server, calling `closeConnection` from one `EDCFactory` object will not affect another connection. You must call `closeConnection` for each `EDCFactory` object that is connected to Policy Server.

3 Working with Policies

This chapter explains how you can use the Policy Server API to create and maintain security policies that belong to Policy Server. A *policy* is a collection of information that includes document security settings, authorized users, and usage rights. You can create and save any number of policies, using security settings appropriate for different situations and users. Policies enable you to perform these tasks:

- Specify who can open the document. Recipients can either belong or be external to your organization.
- Specify how recipients can use the document. You can restrict access to different Acrobat and Adobe Reader features, including the ability to print and copy text, make changes, and add signatures and comments to a document.
- Change the access and security settings at any time, even after you distribute the policy-protected document.
- Monitor the use of the document after you distribute it. You can see how the document is being used and who is using it. For example, you can find out when somebody has opened the document.

There are three Policy Server APIs you use to work with policies. These are `Policy`, `PolicyEntry`, and `PolicyManager`. This chapter discusses how to create `PolicyEntry` and `Policy` objects. For information about creating a `PolicyManager` object, see "Creating a PolicyManager object" on page 14.

This chapter contains the following information:

Topic	Description	See
Creating policies	Describes how to create new policies.	page 18
Setting policy attributes	Describes how to use a `Policy` object to set policy attributes.	page 20
Working with policy entries	Describes how use a `PolicyEntry` object to set security and user information.	page 24
Managing policies	Describes how to use a `PolicyManager` object to manage policies.	page 28
Querying policy information	Describes how to retrieve information about a policy, such as its name.	page 32

Creating policies

To create a policy, perform the following tasks:

- Create a `Policy` object by calling the `InfomodelObjectFactory` object's `createPolicy` method.
- Set the policy's attributes. For information, see "Setting policy attributes" on page 20.
- Create a policy entry. For information, see "Working with policy entries" on page 24.
- Register the policy. For information, see "Registering policies" on page 29.

Creating a Policy object

You create a new policy by calling the `InfomodelObjectFactory` object's `createPolicy` method. This method returns a `Policy` object. For information about an `InfomodelObjectFactory` object, see "Working with InfomodelObjectFactory objects" on page 17.

The `createPolicy` method has two versions:

- createPolicy()
- createPolicy(byte[] xmlAsBytes)

The first version creates a `Policy` object that does not have policy attributes set. Using methods that belong to the `Policy` object, such as the `setName` method, you can set policy attributes. For information, see "Setting policy attributes" on page 20.

The second version creates a `Policy` object that is based on a Portable Document Rights Language (PDRL) XML file. If you use this version of `createPolicy`, you must convert an existing PDRL file into a byte array and pass it to `createPolicy`. For information about PDRL, contact Adobe Customer Support.

You can view a policy as a PDRL XML file by calling the `Policy` object's `toXML` method. This method returns a byte array representing the policy. You can then perform stream operations on the byte array, such as saving the byte array as a PDRL XML file. For information about this method, see the API Reference.

You can create a `Policy` object by calling the `InfomodelObjectFactory` object's `createPolicy` method. The following code example creates a `Policy` object.

Example 3.1 ***Creating a Policy object***

```
Policy myPolicy = InfomodelObjectFactory.createPolicy();
```

Creating a Policy object based on a PDRL XML file

You can create a `Policy` object by passing a byte array representing a PDRL file to the `InfomodelObjectFactory` object's `createPolicy` method. The policy's attributes are defined by the values in the PDRL file. Using the Policy object's methods, you can modify the attributes. For example, you can change the policy's owner. For information, see "Changing the owner of a policy" on page 30.

The following code example creates a `Policy` object that is based on a PDRL XML file named SamplePolicy.xml.

Example 3.2 ***Creating a Policy object based on a PDRL XML file***

```
//Create an InputStream object using a FileInputStream constructor
InputStream xmlPolicyFile = new FileInputStream("C:\\SamplePolicy.xml");

//Get the size of the InputStream object
int bufSize = xmlPolicyFile.available();

//Create a byte array and allocate bufSize bytes
byte[] policyArray = new byte[bufSize];

//Populate the byte array
xmlPolicyFile.read(policyArray);

//Call createPolicy and pass the byte array
Policy mySamplePolicy = InfomodelObjectFactory.createPolicy(policyArray);
```

Caution: If the schema of the PDRL XML file is invalid, an exception is thrown.

Setting policy attributes

You can use the `Policy` object to set policy attributes. For example, you can define the name of a policy by calling the `Policy` object's `setName` method. Using this object, you can set the following policy attributes:

- Offline lease period
- Plain text metadata
- Description
- Name
- Event tracking
- Validity period
- Alternative Id
- Watermark Id
- EncryptAttachmentsOnly

Note: The `name` attribute must be set. Even though the other attributes are optional, it is recommended that you set them to meet your business requirements.

Setting a document's offline lease period

The `offline lease period` defines the number of days a recipient can take the document offline (use it without an active Internet or network connection). To continue using the document, the recipient must synchronize the document with Policy Server by opening it online.

Set this attribute by calling the `Policy` object's `setOfflineLeasePeriod` method and specifying the number of days. The following code example sets the `offline lease period` to five days.

Example 3.3 ***Setting a document's offline lease period***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

//Set the offline lease period to 5 days
myPolicy.setOfflineLeasePeriod(5);
```

Setting a document's metadata

Metadata is information about the document and can be viewed through the Properties dialog box or the Acrobat Advanced menu. You can determine whether a policy enables a recipient to view metadata by calling the `Policy` object's `isPlaintextMetadata` method. This method requires a boolean value that indicates whether a recipient can view metadata.

If you set the `setEncryptAttachmentsOnly` attribute to `true`, you must also set this attribute to `true`. For information, see "Setting the EncryptAttachmentsOnly attribute" on page 23.

The following code example enables a recipient to view metadata.

Example 3.4 ***Setting a document's metadata***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

//Enable a user to view metadata
myPolicy.setPlaintextMetadata(true);
```

Note: Once a policy is registered with Policy Server, this attribute cannot be changed. For information, see "Registering policies" on page 29.

Setting a policy's name and description

A policy's name and description can be defined by using the `Policy` object's `setName` and `setDescription` methods. The `setName` method requires a string value that uniquely identifies the policy, and the `setDescription` method requires a string value that describes the policy. Policy names must be unique among each user. There can be two policy names with the same name provided that the policy name belongs to two separate users. However, a single user cannot have the same policy name.

Note: Policies must have a name defined.

The following code example sets the policy's name and description.

Example 3.5 ***Setting a policy's name and description***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

//Set the policy name and description
myPolicy.setName("Policy2004");
myPolicy.setDescription("This policy belongs to Adobe Policy Server");
```

Setting a document's event tracking

Using the Policy Server API, you can enable or disable tracking of events associated with a policy-protected document. For example, events such as viewing or copying of a document can be tracked. Tracked events appear in the list on the Events page. For information about the Events page, see the *Installing and Configuring* guide.

You can enable event tracking by calling the `Policy` object's `setTracked` method and specifying `true`. The following code example enables event tracking.

Example 3.6 ***Setting a document's event tracking***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

//Enable event tracking
myPolicy.setTracked(true);
```

Setting a policy's validity period

A validity period is the time period during which a policy-protected document is accessible to authorized recipients. A validity period can be set to one of these options:

- A set number of days that the document is accessible from the time which the document is published
- An end date after which the document is not accessible
- A specific date range for which the document is accessible
- Always valid

Before you can set a policy's validity date, you must create a `ValidityPeriod` object by calling the `InfomodelObjectFactory` object's `createValidityPeriod` method. This method returns an object instance based on the `ValidityPeriod` interface.

You define the validity period by calling one of two methods that belong to the `ValidityPeriod` object. The `setRelativeExpirationDays` method sets the validity period to be a relative number of days. You can specify an integer value that defines the number of days.

The `ValidityPeriod` object's `setAbsoluteValidityPeriod` method creates a date range for which the policy is valid. This method requires a Java `Calendar` object that represents a start date and another Java `Calendar` object that represents the end date. The policy is valid within the date range.

You can specify just a start date, which results in the policy being valid after the start date. If you specify just a end date, the policy is valid until the end date. However, an exception is thrown if both a start date and an end date are not defined.

After you create a `ValidityPeriod` object, you can set a policy's validity period by calling the `Policy` object's `setValidityPeriod` method and passing the `ValidityPeriod` object. The following code example creates an absolute validity period with a date range of a week.

Example 3.7 ***Setting a policy's validity period***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

// Create a Calendar object...
Calendar myCalendar = Calendar.getInstance();
myCalendar.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
myCalendar.set( Calendar.DAY_OF_WEEK, Calendar.SUNDAY );
myCalendar.set( Calendar.HOUR_OF_DAY, 21 );
myCalendar.clear( Calendar.MINUTE );
myCalendar.clear( Calendar.SECOND );
myCalendar.clear( Calendar.MILLISECOND );

//Set a validity period from now to 9pm Pacific Time next Sunday
ValidityPeriod vp = InfomodelObjectFactory.createValidityPeriod();
myCalendar.add( Calendar.WEEK_OF_YEAR, 1 );
vp.setAbsoluteValidityPeriod(Calendar.getInstance(), myCalendar );
myPolicy.setValidityPeriod(vp);
```

Setting a policy's alternative identifier

You use the `Policy` object's `setAlternateId` method to set an alternative identifier for a policy. This method requires a string value that represents the alternative identifier. The default format of a policy identifier is similar to the format of a Universal Unique Identifier (UUID). For example, the following represents a policy identifier:

```
1EBE43F3-382E-F6DA-0F3B-7BB001209966
```

It is easier to use a policy's alternative identifier value instead of using its default identifier value. You can set a policy's identifier with a value that describes the policy. For example, assume that a policy is valid until the end of a month. You can set the alternative identifier as Sept2004. The only restrictions are that an identifier cannot exceed 255 characters or have a duplicate value. An alternative identifier value must be unique.

The following code example sets a policy's alternative identifier to Sept2004.

Example 3.8 ***Setting a policy's alternative Id***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

//Set an alternative Id
myPolicy.setAlternateId("Sept2004");
```

Setting a policy's watermark

You can specify a watermark to add to the pages of a document. Watermarks help ensure the security of a document by uniquely identifying the document and controlling copyright infringement. To specify a watermark, call the `Policy` object's `setWatermarkId` method and pass a string value that represents the watermark's identifier.

To get the watermark's identifier, create a `Watermark` object and call its `getId` method. For information about creating a `Watermark` object, see "Creating a Watermark object" on page 44.

The following code example sets a policy's watermark (assume that a `Watermark` object is named myWatermark).

Example 3.9 ***Setting a policy's watermark***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

// Set the policy's watermark
myPolicy.setWatermarkId(myWatermark.getId());
```

Setting the EncryptAttachmentsOnly attribute

You use the `Policy` object's `setEncryptAttachmentsOnly` method to specify whether only document attachments are encrypted or both the document content and the attachment are encrypted. If only the attachment is encrypted, the policy protects only document attachments, not the document. By default, both the document and the attachments are protected by the policy.

The following code example sets this attribute to `true`, which results in only the attachment being encrypted (the default is `false`).

Example 3.10 ***Setting the EncryptAttachmentsOnly attribute***

```
//Create a policy
Policy myPolicy = InfomodelObjectFactory.createPolicy();

// Set the EncryptAttachmentsOnly attribute
myPolicy.setEncryptAttachmentsOnly(true);
```

Note: Once a policy is registered with Policy Server, this attribute cannot be changed. For information, see "Registering policies" on page 29.

Working with policy entries

A policy entry attaches principals, which are groups and users, and permissions to a policy. If you do not create a policy entry, a policy will not have permissions or users associated with it. For example, assume that you perform these tasks:

- Create a policy entry that enables a group to only view a document while online and prohibits recipients from copying it.
- Attach the policy entry to the policy.
- Secure a document with the policy. For information, see "Creating a policy-protected document" on page 37.

These actions result in recipients only being able to view the document online and not being able to copy it. The document remains secure until security is removed from it. For information, see "Removing policy security from a document" on page 38.

➤ **To create a policy entry**

1. Create a `PolicyEntry` object. For information, see "Creating a PolicyEntry object" on page 24.
2. Set the policy entry's permissions. Policy entries must include at least one permission. For information, see "Adding permissions to a policy entry" on page 25.
3. Set the policy entry's principal. Policy entries must include only one principal. For information, see "Adding a principal to a policy entry" on page 26.
4. Attach the policy entry to a policy. For information, see "Attaching a policy entry to a policy" on page 27.

Creating a PolicyEntry object

A policy entry is created by calling the `InfomodelObjectFactory` object's `createPolicyEntry`. This method returns an object instance of the `PolicyEntry` interface. The following code example creates a `PolicyEntry` object.

Example 3.11 ***Creating a PolicyEntry object***

```
//Create a PolicyEntry object
PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();
```

Note: You can specify the validity period of a policy entry by calling the `PolicyEntry` object's `setValidityPeriod` and passing a `ValidityPeriod` object. For information about creating a `ValidityPeriod` object, see "Setting a policy's validity period" on page 22.

Working with permissions

You can use a `PolicyEntry` object to perform the following permission tasks:

- Add permissions to a policy entry
- Retrieve permissions from a policy entry
- Remove a specific permission
- Remove all permissions

Adding permissions to a policy entry

You can add permissions to a policy entry by using the `PolicyEntry` object's `addPermission` method. However, before you call this method, create a `Permission` object by calling the `InfomodelObjectFactory` object's `createPermission` method. This method returns an object instance of the `Permission` interface.

A `Permission` object consists of field values that define permissions. For example, the `OPEN_ONLINE` field enables a recipient to open a document while online. For a complete list of all fields that belong to the `Permission` interface, see the API Reference.

When you call the `createPermission` method, pass a static `Permission` object and specify a field that corresponds to a constant value specifying the permission. For example, to enable a recipient to open a document while online, pass `Permission.OPEN_ONLINE`. Call `createPermission` for each permission you want to add to a policy entry.

To add a permission to a policy entry, call the `PolicyEntry` object's `addPermission` method and pass a `Permission` object. Call `addPermission` for each permission you want to add to a policy entry.

The following code example adds the open online and copy permissions to a policy entry.

Example 3.12 ***Adding permissions to a policy entry***

```
//Create a PolicyEntry object
PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();

//Create Permission objects
Permission onlinePermission =
InfomodelObjectFactory.createPermission(Permission.OPEN_ONLINE) ;
Permission copyPermission =
InfomodelObjectFactory.createPermission(Permission.COPY);

//Add permissions to the policy entry
myPolicyEntry.addPermission(onlinePermission);
myPolicyEntry.addPermission(copyPermission);
```

Note: A policy entry must have at least one permission added. The `OPEN_ONLINE` permission should always be added to ensure that a document can be opened online.

Retrieving permissions from a policy entry

You can retrieve all permissions that are associated with a policy entry by calling the `PolicyEntry` object's `getPermissions` method. This method returns a Java `List` object, where each element is a `Permission` object. You can iterate through the `List` object to retrieve permissions.

The following code example retrieves all permissions from a `PolicyEntry` object.

Example 3.13 ***Retrieving permissions from a policy entry***

```
//Get all permissions associated with this policy entry object
List allPermissions = myPolicyEntry.getPermissions();

//Iterate through the list and display the name of each permission
Iterator it = allPermissions.iterator();
while (it.hasNext())
{
   Permission myPermission = (Permission) it.next();
   System.out.println("The name of the permission is
"+myPermission.getName());
}
```

Removing a specific permission from a policy entry

You can remove a specific permission from a policy entry by calling the `PolicyEntry` object's `removePermission` method. You must pass a `Permission` object that represents the permission to remove. The following code example removes a permission associated with the `copyPermission` object.

Example 3.14 ***Removing a specific permission from a policy entry***

```
myPolicyEntry.removePermission(copyPermission);
```

Note: The `copyPermission` object was added to a `PolicyEntry` object in a previous code example. For information, see "Adding permissions to a policy entry" on page 25.

Removing all permissions from a policy entry

You can remove all permissions from a policy entry by calling the `PolicyEntry` object's `clearPermissions` method. After you call this method, the policy entry does not contain any permissions. The following code example removes all permission from a policy entry.

Example 3.15 ***Removing all permissions from a policy entry***

```
myPolicyEntry.clearPermissions();
```

Working with policy principals

You can use a `PolicyEntry` object to perform the following principal tasks:

- Add a principal to a policy entry
- Retrieve a principal from a policy entry
- Remove principals

Adding a principal to a policy entry

You can add a principal to a policy entry by using the `PolicyEntry` object's `setPrincipal` method. However, before you call this method, create a `Principal` object by calling the `InfomodelObjectFactory` object's `createSpecialPrincipal` method. This method returns an object instance of the `Principal` interface. For information about creating a `Principal` object, see "Creating a special principal object" on page 33.

To add a principal to a policy entry, call the `PolicyEntry` object's `setPrincipal` method and pass a `Principal` object. The following code example adds the publisher of the document (publisher principal) to a policy entry.

Example 3.16 ***Adding a principal to a policy entry***

```
//Create a PolicyEntry object
PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();

//Create principal object
Principal publisherPrincipal =
InfomodelObjectFactory.createSpecialPrincipal(InfomodelObjectFactory.PUBLISH
ER_PRINCIPAL );

//Add a principal object to the policy entry
myPolicyEntry.setPrincipal(publisherPrincipal);
```

Note: There must be only one principal added to a policy entry. It is strongly recommend that a policy entry with the publisher principal be added to a policy so that the publisher has permission to view documents secured with the policy.

Retrieving a principal associated with a policy entry

You can retrieve a principal that is associated with a policy entry by calling the `PolicyEntry` object's `getPrincipal` method. This method returns a `Principal` object or null if a principal does not exist. The following code example retrieves a principal.

Example 3.17 ***Retrieving a principal from a policy entry***

```
//Get a principal object from a policy entry
Principal myPrincipal = myPolicyEntry.getPrincipal();
If (myPrincipal == null)
   System.out.println("There is no principal associated with this policy
entry");
```

Removing a principal

You can remove a principal from a policy entry by calling the `PolicyEntry` object's `clearPrincipal` method. After you call this method, the policy entry does not contain a principal. The following code example removes a principal from a policy entry.

Example 3.18 ***Removing a principal from a policy entry***

```
myPolicyEntry.clearPrincipal();
```

Attaching a policy entry to a policy

A policy entry can be attached to a policy after permission and principal values are defined. To attach a policy entry to a policy, call the `Policy` object's `attachPolicyEntry` method and pass a `PolicyEntry` object. Before calling this method, ensure that a `PolicyEntry` object exists. For information, see "Creating a PolicyEntry object" on page 24.

The following code example shows how to attach a policy entry to a policy.

Example 3.19 ***Attaching a policy entry to a policy***

```
//Create a PolicyEntry object
PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();

//Create Permission objects
Permission onlinePermission =
InfomodelObjectFactory.createPermission(Permission.OPEN_ONLINE) ;
Permission copyPermission =
InfomodelObjectFactory.createPermission(Permission.COPY ) ;

//Add permissions to the policy entry
myPolicyEntry.addPermission(onlinePermission);
myPolicyEntry.addPermission(copyPermission);

//Create principal object
Principal publisherPrincipal =
InfomodelObjectFactory.createSpecialPrincipal(InfomodelObjectFactory.PUBLISH
ER_PRINCIPAL);

//Add a principal object to the policy entry
myPolicyEntry.setPrincipal(publisherPrincipal);

//Attach the policy editor to the policy
myPolicy.addPolicyEntry(myPolicyEntry);
```

Note: You can remove all policy entries from a policy by calling the `Policy` object's `clearPolicyEntries` method.

Managing policies

You can manage a policy by using a `PolicyManager` object. Using this object, you can perform the following tasks:

- Register a policy
- Retrieve a policy
- Search for one or more policies
- Update a policy
- Change the owner of a policy
- Delete a policy

Note: In the following code examples, the name of the `PolicyManager` object is `apsPolicyManager`. For information about creating this object, see "Creating a PolicyManager object" on page 14.

Registering policies

A policy must be registered with Policy Server before it can be used. Register a policy after setting its attributes, permissions, and principal values. For information about the tasks to complete before registering a policy, see "Creating policies" on page 18.

You register a policy by calling the `PolicyManager` object's `registerPolicy` method and passing a `Policy` object that represents the policy to register. The following code example shows how to register a policy with Policy Server.

Example 3.20 ***Registering a policy***

```
//Create a PolicyEntry object
PolicyEntry myPolicyEntry = InfomodelObjectFactory.createPolicyEntry();

//Create Permission objects
Permission onlinePermission =
InfomodelObjectFactory.createPermission(Permission.OPEN_ONLINE) ;
Permission copyPermission =
InfomodelObjectFactory.createPermission(Permission.COPY);

//Add permissions to the policy entry
myPolicyEntry.addPermission(onlinePermission);
myPolicyEntry.addPermission(copyPermission);

//Create principal object
Principal publisherPrincipal =
InfomodelObjectFactory.createSpecialPrincipal(
InfomodelObjectFactory.PUBLISHER_PRINCIPAL);

//Add a principal object to the policy entry
myPolicyEntry.setPrincipal(publisherPrincipal);

//Attach the policy editor to the policy
myPolicy.addPolicyEntry(myPolicyEntry);

//Register the policy with Policy Server
String policyId = apsPolicyManager.registerPolicy(myPolicy);
```

Note: If you attempt to register the same policy twice, an exception is thrown.

Retrieving existing policies

You retrieve existing policies from Policy Server by calling the `PolicyManager` object's `getPolicies` method. This method returns an array of `Policy` objects. Before calling this method, create a `PolicySearchFilter` object by using its public constructor. This object acts as a policy filter that enables you to define search criteria.

You define the search criteria by calling methods that belong to the `PolicySearchFilter` object. For example, you can call the `PolicySearchFilter` object's `setOwner` method to return all policies that have a specific owner. You can create a `PolicySearchFilter` object and pass it to `getPolicies` without defining search criteria. In this situation, all policies are returned.

The following code example retrieves the first ten policies from Policy Server.

Example 3.21 ***Retrieving multiple policies***

```
//Create a PolicySearchFilter object
PolicySearchFilter sf = new PolicySearchFilter() ;

//Get the first ten policies
Policy [] allPolicies = apsPolicyManager.getPolicies(sf,10);

//Iterate through the Policy array and get policy names
for (int zz = 0; zz< allPolicies.length; zz++)
{
   Policy myPolicy = (Policy)allPolicies[zz] ;
   System.out.println("The policy name is "+myPolicy.getName());
}
```

Note: An empty search filter returns all policies (up to the max) that the user has access to.

Retrieving a specific policy

You can retrieve a specific policy from Policy Server by calling the `PolicyManager` object's `getPolicy` method and passing the Id of the policy to retrieve. This method returns a `Policy` object that corresponds to the Id value. If no policies have the specified Id value, this method throws an SDKException.

You can call the `Policy` object's `getId` method to get an Id of a policy. For information about this method, see the API Reference.

The following code example retrieves a policy that corresponds to the policy that has an Id value of `1EBE43F3-382E-F6DA-0F3B-7BB001209966`.

Example 3.22 ***Retrieving a specific policy***

```
//Create a PolicyManager object
PolicyManager apsPolicyManager = apsSession.getPolicyManager();

//Get a policy that corresponds to 1EBE43F3-382E-F6DA-0F3B-7BB001209966
Policy myPolicy =
apsPolicyManager.getPolicy("1EBE43F3-382E-F6DA-0F3B-7BB001209966");
```

Note: If you want to retrieve a policy by its alternative identifier, use the `PolicyManager` object's `getPolicyByAlternateId` method. For information about an alternative identifier, see "Setting a policy's alternative identifier" on page 23.

Changing the owner of a policy

You can change the owner of a policy by calling the `PolicyManager` object's `changePolicyOwner` method. This method requires the policy Id and a `Principal` object that represents the new owner as arguments. If Policy Server does not recognize either the new policy Id or the principal, an exception is thrown.

You cannot change the owner to a special principal that you created. The `Principal` object that you specify must be based on a user and must already exist. For information about retrieving an existing user, see Retrieving users. For information about special principals, see Creating a special principal object.

Note: If a policy includes permissions for the special principal of type PUBLISHER_PRINCIPAL, the policy owner inherits the permissions set for that special principal.

The following code example changes a policy's owner.

Example 3.23 ***Changing the owner of a policy***

```
//Create a PolicyManager object
PolicyManager apsPolicyManager = apsSession.getPolicyManager();

//Change the policy owner to newPrincipal
apsPolicyManager.changePolicyOwner("1EBE43F3-382E-F6DA-0F3B-7BB001209966",
newPrincipal);
```

Updating policies

You can update an existing policy at any time. To make changes to an existing policy, you retrieve it, modify it, and then update the policy on the server.

For example, assume that you retrieve an existing policy by calling the `getPolicy` method and modifying its validity period. Before the change takes effect, you must update the policy by calling the `PolicyManager` object's `updatePolicy` method and passing the `Policy` object that represents the modified policy. For information about a validity period, see "Setting a policy's validity period" on page 22.

The following code example retrieves a policy, modifies its validity period, and then updates it.

Example 3.24 ***Updating a policy***

```
//Get a policy
Policy myPolicy =
apsPolicyManager.getPolicy("1EBE43F3-382E-F6DA-0F3B-7BB001209966");

// Create a Calendar object...
Calendar myCalendar = Calendar.getInstance();
myCalendar.setTimeZone(TimeZone.getTimeZone( "America/Los_Angeles"));
myCalendar.set( Calendar.DAY_OF_WEEK, Calendar.SUNDAY );
myCalendar.set( Calendar.HOUR_OF_DAY, 21 );
myCalendar.clear( Calendar.MINUTE );
myCalendar.clear( Calendar.SECOND );
myCalendar.clear( Calendar.MILLISECOND );

//Change the validity period from now to 9pm Pacific Time next Sunday
ValidityPeriod vp = InfomodelObjectFactory.createValidityPeriod();
myCalendar.add( Calendar.WEEK_OF_YEAR, 1 );
vp.setAbsoluteValidityPeriod(Calendar.getInstance(), myCalendar);
myPolicy.setValidityPeriod(vp);

//Update the policy
apsPolicyManager.updatePolicy(myPolicy);
```

Deleting Policies

You can delete a policy by calling the `PolicyManager` object's `deletePolicy` method and passing a Policy Id that identifies the policy to delete.

After the policy is deleted, it can no longer be applied to new documents. However, the policy is still applicable to existing documents that are using it.

If Policy Server does not recognize the specified Id or if the user connected to Policy Server is not authorized to delete a policy, an exception is thrown. The following code example deletes a policy.

Example 3.25 ***Deleting a policy***

```
//Create a PolicyManager object
PolicyManager apsPolicyManager = apsSession.getPolicyManager();

//Delete the policy that corresponds to "1EBE43F3-382E-F6DA-0F3B-7BB001209966
apsPolicyManager.deletetPolicy("1EBE43F3-382E-F6DA-0F3B-7BB001209966");
```

Tip: If you only know the alternate identification of the policy you want to delete, you can retrieve the associated policy object using the alternate identification, and then use the object to retrieve the policy identification. You cannot delete a policy by using its alternate identification.

Querying policy information

You can use the methods that belong to the `Policy` object to retrieve policy information. For example, you can call the `Policy` object's `getName` method to determine the policy name. For a complete list of all the methods you can use to retrieve policy information, see the API Reference.

4 Working with Policy Server Principals

This chapter explains how to use the Policy Server API to create and manage special principals. A principal can either be a user or a group, and a special principal is a system-defined principal that cannot be modified or updated.

The two Policy Server APIs that you use to work with principals are `Principal` and `UserManager`. The chapter explains how to create a `Principal` object. For information about creating a `UserManager` object, see "Creating a UserManager object" on page 16.

This chapter contains the following information:

Topic	Description	See
Creating a principal object	Describes how to create a `Principal` object.	page 33
Retrieving existing principals	Describes how to retrieve existing principals. You can retrieve existing groups and users.	page 34
Querying principal information	Describes how to retrieve principal information. For example, you can determine whether a principal is a group or user.	page 35

Creating a special principal object

You create a `Principal` object by calling the `InfomodelObjectFactory` object's `createSpecialPrincipal` method. This method returns an object instance of the `Principal` interface that represents a special principal. A special principal has limited functionality compared to a regular principal. For example, a special principal cannot own a policy. For information, see "Changing the owner of a policy" on page 30.

When you call `createSpecialPrincipal`, you must specify a principal field value that belongs to the static `InfomodelObjectFactory` object. For example, this object has a field named `PUBLISHER_PRINCIPAL` that sets the principal to a publisher principal. For information about `InfomodelObjectFactory` fields, see the API Reference.

You create a `Principal` object when you want to attach a principal to a policy entry. For information, see "Adding a principal to a policy entry" on page 26.

A regular principal cannot be created using a `InfomodelObjectFactory` object. That is, you cannot create a user or a group. However, you can retrieve existing users or groups. For information, see "Retrieving existing principals" on page 34.

The following code example creates a `Principal` object that represents a special principal.

Example 4.1 ***Creating a principal object***

```
//Create principal object
Principal publisherPrincipal =
InfomodelObjectFactory.createSpecialPrincipal(
InfomodelObjectFactory.PUBLISHER_PRINCIPAL);
```

Retrieving existing principals

You can retrieve the following existing principal types from Policy Server: groups and users. To retrieve either type, you use a `PrincipalSearchFilter` object. You create this object by using its public constructor. This object acts as a principal filter that enables you to define search criteria by calling its methods. For example, you call the `PrincipalSearchFilter` object's `setEmail` method to search for principals that have the full or the partial specified email value. For information about the `PrincipalSearchFilter` object's methods, see the API Reference.

Retrieving groups

You retrieve existing Policy Server groups by calling the `UserManager` object's `getGroups` method. This method requires a `PrincipalSearchFilter` object that defines the search criteria and an integer value that specifies how many groups to return. It returns an array of `Principal` objects that conform to the search criteria.

The following code returns up to 20 Policy Server groups that have a domain value of `myDomain.com` (this domain is a sample that is used for this code example). The domain value is specified by calling the `PrincipalSearchFilter` object's `setDomainName` method.

Example 4.2 ***Retrieving Policy Server groups***

```
//Create a PrincipalSearchFilter object
PrincipalSearchFilter principalSearch = new PrincipalSearchFilter();

//Define the search criteria
principalSearch.setDomainName("myDomain.com");

//Get Policy Server groups
Principal [] allPrincipals = apsUserManager.getGroups(principalSearch,20);

//Iterate through the Principal array
for (int zz=0; zz<allPrincipals.length;zz++)
{
   Principal myPrincipal = (Principal)allPrincipals[zz];
   System.out.println("The name of the group is " +myPrincipal.getFullName());
}
```

Retrieving users

You retrieve existing Policy Server users by calling the `UserManager` object's `getUsers` method. This method requires a `PrincipalSearchFilter` object that defines the search criteria and an integer value that specifies how many users to return. It returns an array of `Principal` objects that conform to the search criteria.

The following code example returns a user that has the user name Tony Blue and displays the user's email address. This search criteria is defined by calling the `PrincipalSearchFilter` object's `setFullName` method.

Example 4.3 ***Retrieving Policy Server groups***

```
//Create a PrincipalSearchFilter object
PrincipalSearchFilter principalSearch = new PrincipalSearchFilter();

//Define the search criteria
principalSearch.setFullName("Tony Blue");

//Get an Policy Server user with the user name Tony Blue
Principal [] allPrincipals = apsUserManager.getUsers(principalSearch,10);

//Iterate through the Principal array
for (int zz=0; zz<allPrincipals.length;zz++)
{
   Principal myPrincipal = (Principal)allPrincipals[zz];
   System.out.println("The user's email is " +myPrincipal.getEmailAddress());
}
```

Querying principal information

You can use the methods that belong to the `Principal` object to retrieve principal information. For example, you can call the `Principal` object's `getType` method to determine if a principal type is a group or user. For a complete list of all the methods that you can use to retrieve principal information, see the API Reference.

5 Managing Documents

This chapter explains how to use the Policy Server API to control access to and manage policy-protected documents. Using the Policy Server API, you can programmatically perform these tasks:

- Secure a PDF document with an existing policy.
- Switch the policy that is attached to a PDF document (you can only attach one policy at a time to a document). Users who apply policies to documents can switch the policies, provided they created the policy or the policy is an organizational one that enables this capability for the user who applies it.
- Revoke and reinstate the ability to access a policy-protected document. Administrators can revoke and reinstate access to any PDF document. Users can revoke access to their policy-protected documents if the documents are protected by policies they created or by organizational policies that permit this capability for the user who applies the policy.

Note: The first two tasks listed above require that your application can access an additional security component. Policy Server does not include this component. For information about obtaining the security component, contact Adobe Customer Support.

The two main Policy Server APIs that you use to manage documents are the `DocumentManager` and `LicenseManager` interfaces. You also use the `License` interface to perform other tasks related to managing documents. For example, using a `License` object, you retrieve an Id of a license, which is then used to revoke a document.

This chapter contains the following information:

Topic	Description	See
Securing documents	Describes how to secure PDF documents with existing policies.	page 36
Revoking and reinstating documents	Describes how to revoke and reinstate PDF document access capabilities.	page 39
Managing licenses	Describes how to manage licenses applied to policies.	page 40
Querying license information	Describes how to retrieve license information, such as a license Id.	page 43

Securing documents with policies

The `DocumentManager` interface enables you to secure a PDF document with an existing policy, remove security from a policy-protected document, and obtain a license from a policy-protected document. Before you can perform any of these tasks, you must create a `DocumentManager` object. For information, see "Creating a DocumentManager object" on page 15.

Creating a policy-protected document

You use the `DocumentManager` object's `installDocumentSecurity` method to secure a document with a policy. This method requires the following arguments:

- A Java `File` object that represents the document to secure.
- A Java `File` object that represents the policy-protected document. An existing file is overwritten.
- The identifier of the policy that is used to secure the document.
- A string value that is used to identify the policy-protected document. This value cannot exceed 50 characters.
- A string value that specifies an alternate identifier to associate with the license. This optional parameter can be null.
- A `PackageReporter` object that is used to receive progress information. This optional parameter can be null.

Before you can successfully secure a document, you must ensure that a `Policy` object exists. The return value of the `Policy` object's `getId` method is used as the `installDocumentSecurity` method's third argument. For information about creating this object, see "Creating policies" on page 18.

The `installDocumentSecurity` method returns a `License` object that represents the license that is used to secure the document. This object is an instance of the `License` interface. For information about this interface, see the API Reference.

The following code example creates a policy-protected document by using a `Policy` object named `myPolicy`.

Example 5.1 ***Securing a document***

```
//Reference the PDF document to secure
File pdfDocument = new File("C:\\PurchaseOrder.pdf");

//Create a File object that represents the policy-protected document
File secureDocument = new File("C:\\securePurchaseOrder.pdf");

//Specify the name used to identify the policy-protected document
String secureDocName = "securePurchaseOrder";

//Secure the document
License myLicense =
apsDocumentManager.installDocumentSecurity(pdfDocument,secureDocument,
myPolicy.getId(),secureDocName,null,null);
```

Note: To policy-protect a document, your application requires access to an additional security component. For information about this component, contact Adobe Customer Support.

Removing policy security from a document

You remove policy security from a policy-protected document by using the `DocumentManager` object's `removeDocumentSecurity` method. This method requires the following arguments:

- A Java `File` object that represents the policy-protected document.
- A Java `File` object that represents the unsecured document. An existing file is overwritten.
- A `PackageReporter` object that is used to receive progress information. This optional parameter can be null.

Before removing security from a document, call the `DocumentManager` object's `isDocumentSecured` method to ensure that the document is secure. This method requires a Java `File` object that represents the document. The `isDocumentSecured` method returns `true` if the document is secure; otherwise, it returns `false`.

The following code example determines whether a document is secure. *If it is secure,* security is removed.

Example 5.2 ***Removing policy security from a document***

```
//Create a File object based on an existing document
File pdfDocument = new File("C:\\PurchaseOrder.pdf")

//Determine if the document is secure
boolean secured = apsDocumentManager.isDocumentSecured(pdfDocument);

if (secured == true){
   //Create an unsecured document file
   File unsecuredDocument = new File("C:\\unsecuredPurchaseOrder.pdf");

   //Remove the security
   apsDocumentManager.removeDocumentSecurity(pdfDocument,unsecuredDocument,
   null);
}
```

Switching document policies

Sometimes it is necessary to switch a document's policy. For example, assume that a document is secured with a policy that becomes outdated and a newer policy becomes available. In this situation, you can change the policy associated with a policy-protected document's license. For information, see "Changing a policy associated with a license" on page 40.

Retrieving a license from a policy-protected document

You call the `DocumentManager` object's `getDocumentLicense` method to retrieve a Policy Server license that is used to secure a document. This method requires a Java `File` object that represents the policy-protected document and returns a `License` object.

Before retrieving the license from a document, call the `DocumentManager` object's `isDocumentSecured` method to ensure that the document is secure. An exception is thrown if you attempt to retrieve a license from an unsecured document.

Once you retrieve a `License` object, you can perform tasks such as revoking a document. For information, see "Revoking and reinstating documents" on page 39.

The following code example retrieves a Policy Server license from a policy-protected document and displays the policy-protected document's file name.

Example 5.3 ***Retrieving a license from a policy-protected document***

```
//Create a File object based on an existing document
File pdfDocument = new File("C:\\PurchaseOrder.pdf")

//Determine if the document is secure
boolean secured = apsDocumentManager.isDocumentSecured(pdfDocument);

if (secured == true){
   //Retrieve the license from the policy-protected document
   License myLicense = apsDocumentManager.getDocumentLicense(pdfDocument);
   System.out.println("The name of the policy-protected document is "
   +myLicense.getDocumentName());

}
```

Tip: Document publishers can dynamically change policies for documents that they have published. The license for a document may be outdated if the license was changed after the document was published. If you want to ensure that you have the latest license, use the `getLicense` method of the `LicenseManager` class to retrieve the license from the server.

Revoking and reinstating documents

You can revoke and reinstate access capabilities for policy-protected documents by using the Policy Server API. For example, you can programmatically revoke the ability for a recipient to open a policy-protected document that has offline access. When you revoke a document, the change takes effect the next time the recipient synchronizes with Policy Server by opening the policy-protected document online.

The ability to revoke a document provides additional security. For example, assume a newer version of a document becomes available and you do not want anyone viewing the older version. In this situation, the older document can be revoked, and nobody can view the document unless it is reinstated.

To revoke or reinstate a document, you use a `LicenseManager` object. For information about creating this object, see "Creating a LicenseManager object" on page 16.

Revoking documents

You revoke a document by using the `LicenseManager` object's `revokeLicense` method. This method requires the following arguments:

- A string value that represents the Id of the license to revoke. You can get this value by calling the `License` object's `getId` method.
- An integer value that specifies the reason to revoke the document. This argument is a field of a static `License` object and is one of the following values: `License.DOCUMENT_REVISED`, `License.DOCUMENT_TERMINATED`, `License.GENERAL_MESSAGE`. For information about the meaning of these fields, see the API Reference.
- A Java `URL` object that directs the user who is trying to open the document to a resource on the Internet. For example, this object can direct a user to a newer version of the document. This argument is optional and may be null.

The following code example revokes a document. The license Id is retrieved by calling the `License` object's `getId` method. This value is passed to the `revokeLicense` method.

Example 5.4 ***Revoking a document***

```
//Retrieve the license from the policy-protected document
License myLicense = apsDocumentManager.getDocumentLicense(pdfDocument);

//Get the License Id value
String myLicId = myLicense.getId();

// Revoke the document
apsLicenseManager.revokeLicense(myLicId, License.DOCUMENT_TERMINATED, null);
```

Reinstating documents

You can reinstate a revoked document by using the `LicenseManager` object's `unrevokeLicense` method. This method requires a string value that represents the Id of the license used to revoke the document. The `unrevokeLicense` method does not have a return value.

The following code example reinstates a document.

Example 5.5 ***Reinstating a document***

```
// Assume the myLicenseId is used to revoke a document
apsLicenseManager.unrevokeLicense(myLicenseId);
```

Managing licenses

In addition to using a `LicenseManager` object to revoke and reinstate policy-protected documents, you can also use this object to manage licenses. Using this object, you can perform the following tasks:

- Change the policy associated with licenses
- Set alternative Id for licenses
- Retrieve licenses
- Updating the URL of a license

Changing a policy associated with a license

You use the `LicenseManager` object's `changeLicensePolicy` method to change a policy associated with a license. This ensures that a license uses the most up-to-date secure policy available. For example, assume a document is initially highly confidential and is only available to a few recipients (such as board members who belong to a company). After a while, the document becomes less confidential and the policy is changed so that the document is available to all employees.

If the document is online, the change takes affect immediately. However, if the document is offline, the change takes affect the next time a recipient synchronizes with Policy Server by opening the policy-protected document online.

The `changeLicensePolicy` method requires the following arguments:

- The Id of the license to which the policy change is made
- The Id of the new policy

The following code changes a policy associated with a license.

Example 5.6 ***Changing a policy associated with a license***

```
//Create a File object based on a policy-protected document
File myDocument = new File("C:\\securePurchaseOrder.pdf");

//Get the license of the policy-protected document
License myLicense = apsDocumentManager.getDocumentLicense(myDocument);

//Change the license's policy
apsLicenseManager.changeLicensePolicy(myLicense.getId(), newPolicy.getId());
```

Note: In this code example, assume that the `newPolicy` object represents the new policy. For information about creating a `Policy` object, see "Creating a Policy object" on page 19.

Setting an alternative Id for a license

You use the `LicenseManager` object's `setLicenseAlternateId` method to set an alternative identifier for a license. This method requires a string value that represents the alternative identifier. The default format of a license identifier is similar to the format of a Universal Unique Identifier (UUID). For example, the following string represents a license identifier:

```
ISC17168AA-3603-83A9-75F1-1C9497E72B30
```

It is easier to use an alternative identifier value than to use its default Id value. You can set a license's alternative identifier with a value that describes the license. If you attempt to set a license's alternative identifier with a value that is already in use, an exception is thrown.

The following code example sets a license's alternative identifier

Example 5.7 ***Setting a license's alternative Id***

```
// Set a license's alternative identifier
apsLicenseManager.setLicenseAlternateId("OctoberPurchaseOrderLicense");
```

Retrieving existing licenses

You retrieve existing licenses from Policy Server by calling the `LicenseManager` object's `getLicenses` method. This method returns an array of `License` objects. Before calling this method, create a `LicenseSearchFilter` object by using its public constructor. This object acts as a license filter that enables you to define search criteria.

You define the search criteria by calling methods that belong to the `LicenseSearchFilter` object. For example, call its `setLicenseIssueBeginDate` method to return all `License` objects issued after a specific date.

The following code example retrieves the first 20 Policy Server licenses issued after August 21, 2004.

Example 5.8 ***Retrieving existing licenses***

```
//Create a LicenseSearchFilter object
LicenseSearchFilter licenseSF = new LicenseSearchFilter();

//Create a Calendar object to pass to setLicenseIssueBeginDate
Calendar myCalendar2 = Calendar.getInstance();
myCalendar2.set(2004,8,21);
licenseSF.setLicenseIssueBeginDate(myCalendar2.getTime());

//Get the first 20 licenses issued after Aug 21, 2004
License [] allLicenses = apsLicenseManager.getLicenses(licenseSF,20);

for (int zz = 0; zz< allLicenseslength; zz++)
{
   License myLicense = (License)allLicenses[zz] ;
   System.out.println("The name of document that uses this license is
"+myLicense.getDocumentName());
}
```

Note: For more information about the `LicenseSearchFilter` object, see the API Reference.

Retrieving a specific license

You can retrieve a specific license from Policy Server by calling the `LicenseManager` object's `getLicense` method and passing the Id of the license to retrieve. This method returns a `License` object that corresponds to the Id value. If there are no licenses with the specified Id value, this method returns null.

You can call the `License` object's `getId` method to get a license Id. The following code example retrieves a license that has an Id value of `ISC17168AA-3603-83A9-75F1-1C9497E72B30`.

Example 5.9 ***Retrieving a specific license***

```
//Get a license that corresponds to ISC17168AA-3603-83A9-75F1-1C9497E72B30
License myLicense =
apsLicenseManager.getLicense("ISC17168AA-3603-83A9-75F1-1C9497E72B30");
System.out.println("The name of document that uses the license is
"+myLicense.getDocumentName());
```

Note: To retrieve a license by its alternative Id, you must use the `LicenseManager` object's `getLicenseByAlternateId` method. For information about an alternative Id, see "Setting an alternative Id for a license" on page 41.

Updating the URL of a license

You use the `LicenseManager` object's `updateLicenseRevocationUrl` method to update or remove the revocation URL for a license that is currently revoked. This method requires following arguments:

- A string that specifies the license Id of the revoked license
- A `URL` object that represents the URL of the new revocation URL and null if you want remove the URL that was previously set.

For more information about the `updateLicenseRevocationUrl` method, see the API Reference.

Querying license information

You can use the methods that belong to the `License` object to retrieve license information. For example, call the `License` object's `getIssueDate` method to determine the date on which the license was issued. For a complete list of all the methods you can use to retrieve license information, see the API Reference.

6 Working with Watermarks

This chapter explains how to use the Policy Server API to create and manipulate watermarks. Watermarks help ensure the security of a document by uniquely identifying the document and controlling copyright infringement. For example, you can create and place a watermark that states *Confidential* on all pages of a document.

The two Policy Server APIs that you use to work with watermarks are `Watermark` and `WatermarkManager`. This chapter discusses how to create a `Watermark` object. For information about creating a `WatermarkManager` object, see "Creating a WatermarkManager object" on page 16.

This chapter contains the following information:

Topic	Description	See
Creating watermarks	Describes how to create new watermarks.	page 44
Setting watermark attributes	Describes how to use a `Watermark` object to set watermark attributes.	page 45
Managing watermarks	Describes how use a `WatermarkManager` object to manage watermarks.	page 48
Querying watermark information	Describes how to retrieve watermark information, such as the name of the watermark.	page 51

Creating watermarks

To create a watermark, perform the following tasks:

1. Create a `Watermark` object.
2. Set the attributes for the watermark. For information, see "Setting watermark attributes" on page 45.
3. Register the watermark with Policy Server. For information, see "Registering watermarks" on page 49.

Creating a Watermark object

You create a `Watermark` object by calling the `InfomodelObjectFactory` object's `createWatermark` method. This method returns a `Watermark` object that is based on the `Watermark` interface. For information about an `InfomodelObjectFactory` object, see "Working with InfomodelObjectFactory objects" on page 17.

The following code example creates a `Watermark` object.

Example 6.1 ***Creating a Watermark object***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();
```

Note: Only administrators can register watermarks with Policy Server. As a result, there is no reason for anyone other than administrators to create watermark objects.

Setting watermark attributes

You can use the `Watermark` object to set the following watermark attributes (for example, you can define the name of a watermark by calling the `Watermark` object's `setName` method):

- Background
- Custom text
- Date information
- Horizontal position
- Name
- Opacity
- Rotation
- Scale
- User information
- Vertical position

Setting the background attribute

You determine whether a watermark is set in a document page's background or foreground by calling the `Watermark` object's `setBackground` method. If you specify `true`, the watermark is set in the page's background; otherwise, it is set in the foreground.

Before you set this attribute, call the `Watermark` object's `isBackground` method. This method returns true if the watermark is set in a document page's background. The following code example sets a watermark in a page's background, providing that this attribute is not already set.

Example 6.2 ***Setting the background attribute***

```
if (myWatermark.isBackground() != true)
{
   //Set the background attribute
   myWatermark.setBackground(true);
}
```

Setting the custom text attribute

You can set the text that a watermark displays by calling the `Watermark` object's `setCustomText` method. This method requires a string value that represents the text. The following code example sets the custom text attribute to Confidential.

Example 6.3 ***Setting the custom text attribute***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the custom text attribute
myWatermark.setCustomText("Confidential");
```

Setting the setDateIncluded attribute

You determine whether a watermark displays the date that a document was opened by calling the `Watermark` object's `setDateIncluded` method. If you specify `true`, the date will appear in the watermark. The following code example sets this attribute to `true`.

Example 6.4 ***Setting the setDateIncluded attribute***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the setDateIncluded attribute
myWatermark.setDateIncluded(true);
```

Setting the setHorizontalAlignment attribute

You set the `setHorizontalAlignment` attribute by calling the `Watermark` object's `setHorizontalAlignment` method. This method requires one of the following string values:

- HORIZONTAL_ALIGNMENT_LEFT
- HORIZONTAL_ALIGNMENT_CENTER
- HORIZONTAL_ALIGNMENT_RIGHT

The following code example sets this attribute to HORIZONTAL_ALIGNMENT_CENTER.

Example 6.5 ***Setting the setHorizontalAlignment attribute***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the setHorizontalAlignment attribute
myWatermark.setHorizontalAlignment("HORIZONTAL_ALIGNMENT_CENTER");
```

Setting the name attribute

You set the name of a watermark by calling the `Watermark` object's `setName` method. The only two restrictions are no two watermarks can have the same name and a name cannot exceed 255 characters. The following code example sets the name of a watermark.

Example 6.6 ***Setting the name attribute***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the name attribute
myWatermark.setName("Confidential");
```

Setting the opacity attribute

You set the `opacity` attribute by calling the `Watermark` object's `setOpacity` method and specifying an integer value that represents the opacity percentage. The higher the percentage, the easier it is to view the watermark.

The following code example sets the `opacity` attribute to 80 percent.

Example 6.7 *Setting the opacity attribute*

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the opacity attribute
myWatermark.setOpacity(80);
```

Setting the rotation attribute

You set the `rotation` attribute by calling the `Watermark` object's `setRotation` method and specifying an integer value that represents the number of degrees to rotate the watermark. Values are from 0 to 359, inclusive. The following code example sets this attribute to 180.

Example 6.8 *Setting the rotation attribute*

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the opacity attribute
myWatermark.setRotation(180);
```

Setting the scale attribute

You set the `scale` attribute by calling the `Watermark` object's `setScale` method and specifying an integer value that represents the scale percentage. The value 0 specifies the fit-to-page scale and is the default size of a watermark. Valid values are from 0 to 99, inclusive. The following code example sets this attribute to 10.

Example 6.9 *Setting the scale attribute*

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the scale attribute
myWatermark.setScale(10);
```

Setting the setUserIdIncluded attribute

You set the `setUserIdIncluded` attribute by calling the `Watermark` object's `setUserIdIncluded` method. This method requires a boolean value that specifies whether the watermark includes the Id of the user who opened the document. If you specify true, the user Id is included. Before setting this attribute, call the `isUserIdIncluded` method to determine if it is already set.

The following code example sets this attribute to `true`.

Example 6.10 *Setting the setUserIdIncluded attribute*

```
if (myWatermark.isUserIdIncluded() != true)
{
   //Set the setUserIdIncluded attribute
   myWatermark.setUserIdIncluded(true);

}
```

Setting the setUserNameIncluded attribute

You set the `setUserNameIncluded` attribute by calling the `Watermark` object's `setUserNameIncluded` method. This method requires a boolean value that specifies whether the watermark includes the name of the user who opened the document. If you specify true, the user name is included. Before setting this attribute, call the `isUserNameIncluded` method to determine if it is already set.

The following code example sets this attribute to true.

Example 6.11 ***Setting the setUserNameIncluded attribute***

```
if (myWatermark.isUserNameIncluded() != true)
{
   //Set the setUserNameIncluded attribute
   myWatermark.setUserNameIncluded(true);

}
```

Setting the setVerticalAlignment attribute

You set the `setVerticalAlignment` attribute by calling the `Watermark` object's `setVerticalAlignment` method. This method requires one of the following string values as an argument:

- VERTICAL_ALIGNMENT_TOP
- VERTICAL_ALIGNMENT_CENTER
- VERTICAL_ALIGNMENT_BOTTOM

The following code example sets this attribute to VERTICAL_ALIGNMENT_CENTER.

Example 6.12 ***Setting the setVerticalAlignment attribute***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the setVerticalAlignment attribute
myWatermark.setVerticalAlignment("VERTICAL_ALIGNMENT_CENTER");
```

Managing watermarks

You can manage a watermark by using a `WaterManager` object. Using this object, you can perform the following tasks:

- Register a watermark
- Retrieve a watermark
- Update a watermark
- Delete a watermark

Note: In the following code examples, the name of the `WatermarkManager` object is named `apsWaterManager`. For information about creating this object, see "Creating a WatermarkManager object" on page 16.

Registering watermarks

A watermark must be registered with Policy Server before it can be used. Only an administrator can register a watermark. Register a watermark after setting its attributes. For information, see "Setting watermark attributes" on page 45.

You register a watermark by calling the `WatermarkManager` object's `registerWatermark` method and passing a `Watermark` object that represents the watermark to register. This method returns a string value that specifies the watermark's Id.

The following code example registers a watermark with Policy Server.

Example 6.13 ***Registering a watermark with Policy Server***

```
//Create a Watermark object
Watermark myWatermark = InfomodelObjectFactory.createWatermark();

//Set the custom text attribute
myWatermark.setCustomText("Confidential");

//Set the name attribute
myWatermark.setName("Confidential");

//Set the setDateIncluded attribute
myWatermark.setDateIncluded(true);

//Register the watermark and display its Id
String waterId = apsWatermarkManager.registerWatermark(myWatermark);
System.out.println("The Id of the registered watermark is "+waterId);
```

Note: If you attempt to register the same watermark twice, an exception is thrown.

Retrieving existing watermarks

You can retrieve an existing watermark from Policy Server by using either of these methods:

- `getWatermarkByName`
- `getWatermarkById`

Both these methods belong to the `WatermarkManager` interface. The `getWatermarkByName` method requires a string value that specifies the watermark's name and returns a `Watermark` object.

The `getWatermarkById` method requires a string value that specifies the watermark's Id and returns a `Watermark` object. The format of a watermark Id is similar to the format of a Universal Unique Identifier (UUID). For example, the following string value represents a watermark Id:

```
899EE683-5088-61D7-5C7A-73934F68E629
```

The following code retrieves a watermark named Confidential and displays its Id.

Example 6.14 ***Retrieving an existing watermark***

```
//Retrieve a watermark named confidential and display its Id value
Watermark wm = apsWatermarkManager.getWatermarkByName("Confidential");

if (wm != null)
   System.out.println("The Id of the registered watermark is "+wm.getId());
else
   System.out.println("There is no watermark that is named Confidential");
```

Note: An SDKException is thrown if no watermark corresponds to the specified name or Id value.

Updating watermarks

You can update a watermark anytime after you modify it. Assume that you retrieve an existing watermark by calling the `getWatermarkByName` method and then modify its opacity attribute. Before the change takes effect, you must update the watermark by calling the `WatermarkManager` object's `updateWatermark` method and passing a `Watermark` object that represents the modified watermark. For information about the opacity attribute, see "Setting the opacity attribute" on page 46.

The following code example retrieves a watermark, modifies its opacity attribute, and then updates it.

Example 6.15 ***Updating a watermark***

```
//Retrieve a watermark named confidential
Watermark wm = apsWatermarkManager.getWatermarkByName("Confidential");

if (wm != null)
   System.out.println("The Id of the registered watermark is "+wm.getId());
else
   System.out.println("There is no watermark that is named Confidential");

//modify it's opacity attribute
wm.setOpacity(60);

//Update the watermark
apsWatermarkManager.updateWatermark(wm);
```

Deleting watermarks

You can delete a watermark by calling the `WatermarkManager` object's `deleteWatermark` method. This method requires a watermark Id that identifies the watermark. After the watermark is deleted, it can not be added to policies. However, policies that are already using the watermark can still do so.

The following code example deletes a watermark.

Example 6.16 ***Deleting a watermark***

```
//Delete a watermark
apsWatermarkManager.deletetWatermark("899EE683-5088-61D7-5C7A-73934F68E69");
```

Note: If you specify an invalid watermark Id, an exception is thrown.

Querying watermark information

You can use the methods that belong to the `Watermark` object to retrieve watermark information. For example, you can call the `Watermark` object's `getName` method to determine the watermark name. For a complete list of all the methods you can use to retrieve watermark information, see the API Reference.

7 Registering Event Handlers

This chapter explains how you can use the Policy Server SDK API to configure and register event handlers. When event auditing is enabled, Policy Server tracks Policy Server-related actions as they occur, such as applying a policy to a document or opening a policy-protected document.

This chapter contains the following information:

Topic	Description	See
Events and event handlers	Describes events and event handlers.	page 52
Registering event handlers	Describes how to register event handlers and specify the events that they process.	page 53
Unregistering event handlers	Describes how to unregister event handlers so that they no longer receive events for processing.	page 54
Retrieving event handlers	Describes how to retrieve objects that contain information about existing event handlers.	page 54
Modifying event handlers	Describes how to change the events that an event handler is subscribed to.	page 54
Retrieving subscribable events	Describes how to retrieve all events that an event handler subscribes to.	page 55

Events and event handlers

Policy Server can be configured to maintain an audit of the actions that users and the server component perform. These actions are referred to as events. Event handlers are applications that process the events. Policy Server passes event information to event handlers as the events occur.

Policy Server includes an event handler that enables users and administrators to view audited events. Information about the audited events are displayed in the Policy Server web pages.

Event types

Events fall into one of the following categories:

- Administrator events are actions related to an administrator, such as creating a new administrator account
- Document events are actions related to a document, such as closing a policy-protected document.
- Policy events are actions related to a policy, such as creating a new policy.
- Server events are actions related to Policy Server, such as synchronizing with the user directory.
- User events are actions related to a user, such as deleting a user account.

You cannot create new events. The fields of the `EventManager` interface define all of the available events. For a list of events, see the API Reference.

Creating event handlers

To create an event handler, you need to implement the `EventHandler` interface of the `com.adobe.edc.server.spi` Java package. Although the details about implementing the interface is beyond the scope of this guide, the API Reference includes information about it.

For information about developing custom event handlers, contact Adobe Customer Support.

Registering event handlers

You must register an event handler before it can track events, such as a policy-protected document being opened. To register an event handler, perform the following tasks:

1. Create an `EventHandlerDefinition` object by using its public constructor.
2. Define an integer array of event codes to which that event handler subscribes. Event codes are defined by fields that belong to the `EventManager` interface. For example, the `EventManager` object's `DOCUMENT_VIEW_EVENT` field defines an event that occurs when a policy-protected document is opened. You must use a static `EventManager` object to reference event codes.
3. Call the `EventHandlerDefinition` object's `setEventType` method and pass the integer array that defines the event codes.
4. Call the `EventHandlerDefinition` object's `setHandlerClass` method and pass a string value that specifies the custom event handler class. An event handler can only have one event handler class. If you attempt to register an event handler without defining an event handler class, an exception is thrown.
5. Call the `EventHandlerDefinition` object's `setHandlerInitData` method and pass a string value that specifies the location of data required to initialize the handler.
6. Call the `EventManager` object's `registerEventHandler` method and pass the `EventHandlerDefinition` object. This method does not have a return value. For information about creating an `EventManager` object, see "Creating an EventManager object" on page 15.

The following code example registers an event handler that uses an event handler class named `PostEventHandler`.

Example 7.1 ***Registering an event handler***

```
// Create an EventHandlerDefinition object
EventHandlerDefinition eventHandleDef = new EventHandlerDefinition();

//Define an integer array of event codes
int [] defineEvents
={EventManager.DOCUMENT_VIEW_EVENT,EventManager.DOCUMENT_FORM_FILL_EVENT,
EventManager.DOCUMENT_CLOSE_EVENT};

//Set the event codes, the handler class, and the initialization data
eventHandleDef.setEventType(defineEvents);
eventHandleDef.setHandlerClass("events.handler.PostEventHandler");
eventHandleDef.setHandlerInitData("http://localhost:8080/events/events");

//Register the event handler
apsEventManager.registerEventHandler(eventDef);
```

Note: In this code example, the event handler subscribes to three events: the document view event, document fill event, and document close event. You can specify `EventManager.ALL_EVENTS` in the event array, which results in the event handler subscribing to all events.

Unregistering event handlers

You can unregister an event handler by using methods that belong to the `EventManager` interface. After an event handler is unregistered, it must be registered again before it can be used. You can unregister all event handlers by calling the `EventManager` object's `unregisterAllEventHandlers` method. This method does not require any arguments and does not have a return value.

You can unregister a specific event handler by calling the `EventManager` object's `unregisterEventHandler` method. This method requires a string value that specifies the event handler class to unregister and does not have a return value.

Retrieving event handlers

You can retrieve all registered event handlers by using the `EventManager` object's `getEventHandlers` method. This method returns an array of `EventHandlerDefinition` objects, where each object represents a registered event handler. The event handler's class can be obtained by calling the `EventHandlerDefinition` object's `getHandlerClass` method.

The following code example retrieves all registered event handlers and displays each event handler's class.

Example 7.2 ***Retrieving event handlers***

```
// Create an array of EventHandlerDefinition objects
EventHandlerDefinition [] allEventHandlers =
apsEventManager.getEventHandlers();

//Iterate through the array
for (int xx=0; xx<allEventHandlers.length;xx++){
   EventHandlerDefinition eventHandlerOb = (EventHandlerDefinition)
allEventHandlers[xx] ;
   System.out.println("The name of the event handler's class is
"+eventHandlerOb.getHandlerClass());
}
```

Modifying event handlers

You modify the events that an existing event handler subscribes to by calling the `EventManager` object's `modifyEventsForHandler` method. This method requires a string value that specifies the event handler class to modify and an integer array of event codes for which that event handler subscribes to. As previously stated in this chapter, event codes are defined by fields that belong to the `EventManager` interface.

The specified events replace the previously registered events for this event handler. To add additional events, you must include all the events that the event handler previously subscribed to and the additional events.

The following code example adds two new events to an event handler. Assume that the event handler in this code example previously subscribed to the following three events: view event, form fill event, and close event.

Example 7.3 ***Modifying event handlers***

```
// Create an EventHandlerDefinition object
EventHandlerDefinition eventHandleDef = new EventHandlerDefinition();

//Define an integer array of event codes. Specify the original three
//events plus the addition two. Now there will be five events to which
//this event handler subscribes
int [] defineEvents
={EventManager.DOCUMENT_VIEW_EVENT,EventManager.DOCUMENT_FORM_FILL_EVENT,
EventManager.DOCUMENT_CLOSE_EVENT,
EventManager.DOCUMENT_MODIFY_EVENT,EventManager.DOCUMENT_COPY_CONTENT_EVENT}
;

//Set the event codes, the handler class, and the initialization data
eventHandleDef.setEventType(defineEvents);
eventHandleDef.setHandlerClass("events.handler.PostEventHandler");
eventHandleDef.setHandlerInitData("http://localhost:8080/events/events");

//Register the event handler
apsEventManager.registerEventHandler(eventDef);
```

Retrieving subscribable events

You can get all events to which an event handler can subscribe to by calling the `EventManager` object's `getSubscribableEvents` method. This method returns an array of `Event` objects, where each object represents an event to which an event handler can subscribe.

The `Event` interface, on which an `Event` object is based, consists of three methods:

- getCategory, which returns the event's category
- getName, which returns the event's name
- getType, which returns the event's type code

After you call the `getSubscribableEvents` method, you can iterate through the array of `Event` objects and retrieve the event's name, category, and type code. The following code example retrieves all events and displays each event's name, category, and type code.

Example 7.4 ***Retrieving subscribable events***

```
//Create an array of Event objects
Event [] allEvents = apsEventManager.getSubscribableEvents();

//Iterate through the array
for (int xx=0; xx<allEvents.length;xx++){
    Event eventOb = (Event) allEvents[xx] ;
    System.out.println("The name of the event is "+eventOb.getName());
    System.out.println("The category of the event is "+eventOb.getCategory());
    System.out.println("The type of the event is "+eventOb.getType());
}
```

Note: For more information about the `Event` interface, see the API Reference.

Index

L

M

O

P

R

S

U

V

W