Oracle® Fusion Middleware

Developing Oracle WebCenter Content: Imaging

12*c* (12.2.1.3.0)
E48250-02
March 2018

Oracle Fusion Middleware Developing Oracle WebCenter Content: Imaging, 12*c* (12.2.1.3.0)

E48250-02

Primary Author: Divya Ramabhadran

Contributors: Sonia Nagar, Brian Gray

Contents

Preface

Oracle WebCenter Content: Imaging integrates electronic document storage, retrieval, and annotation with business processes to facilitate document use across an enterprise. Documents are uploaded into a repository managed by Oracle WebCenter Content using an application within Imaging. Applications are predefined by you based on your business need. The application used to upload the document is chosen based on the type of document being uploaded. For example, one application would be used to upload an invoice and a different application would be used to upload a contract. The application determines the metadata that is associated with a document, as well as security permissions to the document and any document annotations. This guide details how to define applications and searches, connect to a workflow server to integrate with other business processes, and configure Imaging to best meet your company needs.

Audience

This document is intended for developers responsible for customizing Oracle WebCenter Content: Imaging functionality.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

The complete Oracle WebCenter Content documentation set is available from the Oracle Help Center at http://www.oracle.com/pls/topic/lookup?ctx=fmw122130&id=wccdocs.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

The following topic introduces the significant changes that are described in this guide, and provides pointers to additional information.

Significant Documentation Changes for 12c (12.2.1.3.0)

There are no significant changes to this guide for the 12*c* (12.2.1.3.0) release.

Part I

Getting Started with Customizing Oracle WebCenter Content: Imaging

Part I contains the following chapters:

1

Introduction to Oracle WebCenter Content: Imaging

This chapter provides an overview of Oracle WebCenter Content: Imaging, including the integration options, the common API model, software requirements, and a sample of the Imaging integration API.
This chapter includes the following sections:

1.1 Overview of Integration Options

Imaging provides the following two integration options: native Java API and Web services.

Native Java API

The Imaging native Java API offers a comprehensive set of Java classes providing access to all aspects of Imaging. It is the easiest option from environments that can directly reference Java code. Because it is implemented with client-side Java code, the API provides a number of convenience utilities for common tasks such as populating data structures, searching and sorting collections, and enumerating data types. Moreover, the native Java API acts as a wrapper for Web services. In this way, an integrator can use the Java API and need not worry about the underlying Web services implementation.

Web Services

Imaging functionality is available directly as Simple Object Access Protocol (SOAP) based Web services. These services allow access to Imaging functionality in non-Java environments that support calling Web services. All of the core functionality of the Imaging API is available in the Web service set.

Because the Imaging API is exposed as Web services, access to Imaging functionality is available to any programming environment supporting Web services. Access Imaging like any Web service by using the Imaging Web Services Description Language (WSDL) to generate implementation classes for those services.

Application Extension Framework

Oracle provides productized integrations with business applications like Oracle E-Business Suite. These integrations are enabled by the Application Extension Framework (AXF) component of Imaging. The AXF provides a command-driven, web services integration that allows administrators to configure and modify multiple business process solutions separate from the systems themselves. For more details

about the AXF and AXF imaging solutions, see About Application Extension Framework (AXF) in *Administering the Application Adapters for Oracle WebCenter*.

1.2 Common API Model

The Imaging API has been implemented based on service-oriented architecture design patterns. The components of the model can be broken down into the following two categories: services and objects.

Services

The services translate into Web service requests. These services do not contain data, but provide methods to accomplish common tasks such as creating a document. The services move data objects back and forth to the Imaging servers.

Objects

Objects contain only data. The only methods they provide are simple get and set operations for the object properties.

See the *Imaging Java API Reference* guide for detailed information about Imaging services, methods, and parameters. Code samples that demonstrate many key API features are provided throughout this guide.

1.2.1 Understanding Services

Imaging services can be divided into the following categories:

- **Definition**: The definition services provide the functionality to manage the structure of the Imaging system including applications, searches, and inputs.
- **Document**: The document services provide the functionality to manage the content within an Imaging system.
- **System**: The system services provide the functionality to manage the Imaging system.

The following common methods are available among these services when applicable:

Methods	Description
`list...()`	The list method is used to find definitions with which the current user can interact. The type of interaction, such as *view* or *modify*, is defined using the `Ability` enumeration and is provided as a parameter.
`get...()`	The get method is used to retrieve a definition object. This method includes an array of `SectionFlag` enumeration values used for requesting specific portions of the object.

`create...()`
`update...()`
`delete...()`
	The create, update, and delete methods provide standard maintenance functions for their respective objects within the limits of the current user context.

1.2.1.1 Definition Services

The following definition services provide basic creation and maintenance of the structure of the Imaging system.

- ApplicationService
- SearchService
- InputService
- ConnectionService
- ImportExportService

1.2.1.1.1 ApplicationService

An Imaging application represents a document repository with a uniquely configured set of metadata, privilege assignments, document lifecycle specifications, workflow integrations, auditing configurations, and other items necessary to properly manage documents and transactions. ApplicationService provides the facilities to find, create, modify, and delete applications. The definition of the application resides within an application object that contains multiple subobjects.

1.2.1.1.2 SearchService

Imaging is designed to provide storage for a large number of documents. Because it can be difficult for users to navigate a large number of documents by browsing folders, the Imaging system provides a comprehensive searching solution as a more efficient way to find desired content. SearchService is used to find, delete, and execute searches. Because search definitions are complex entities, it is best practice to create a search using the Imaging user interface and then save it in the system. Saved searches provide an effective mechanism to leverage complex queries through an integration. It only requires that you know the name of the search and the desired search parameter values.

1.2.1.1.3 InputService

InputService is used to find, create, modify, and delete input definition objects. InputService allows state modifications as well as toggling an input online/offline. It allows users to upload sample data as well as to get definition file information.

1.2.1.1.4 ConnectionService

ConnectionService is used to find, create, modify, and delete connection definition objects. Connections are used to connect Imaging to workflow servers and Content Server repositories.

1.2.1.1.5 ImportExportService

ImportExportService is used to create applications, searches and inputs in bulk within an Imaging system.

1.2.1.2 Document Services

The following document services are used to create and maintain document content in Imaging.

- DocumentService
- DocumentContentService

1.2.1.2.1 DocumentService

DocumentService is used to access, lock, move, copy, modify, and delete the documents within the Imaging repository. All document service method actions are bounded by the current user context and the associated document security configured in the application in which the document resides. DocumentService also maintains the annotations associated with a document and provides the ability to render documents into image formats.

1.2.1.2.2 DocumentContentService

DocumentContentService is used to upload documents and retrieve documents.

1.2.1.3 System Services

The following system services provide functionality relevant to the management of the Imaging system.

- LifecycleService
- LoginService
- PreferenceService
- SecurityService
- TicketService

1.2.1.3.1 LifecycleService

LifecycleService provides the ability to obtain information about the storage volumes that are available for document storage. The volume objects represent the storage media. Storage volumes are associated with documents in the application definition.

1.2.1.3.2 LoginService

LoginService establishes and terminates user sessions for Imaging. A session must be established before any of the other services may be used.

1.2.1.3.3 PreferenceService

PreferenceService provides the ability to store preference information at either the system or the user level. The system level provides a single configuration instance for an Imaging installation while the user level provides a unique configuration instance for each user. This service provides basic storage and retrieval of those preference settings. Note that if setting preferences from the API, you must be mindful of type. There is no validation against an incorrect setting passed through the API. Validation occurs only when preferences are set through the user interface.

1.2.1.3.4 SecurityService

SecurityService provides mechanisms to query the system for users and user groups that exist within the encompassing security environment. It also provides the ability to retrieve and define the security privileges that are beyond those security elements defined within each of the system definition objects.

1.2.1.3.5 TicketService

TicketService allows users to create or delete a ticket and perform a collective delete of expired tickets.

1.2.2 Understanding Data Objects

The system services provide management at the Imaging API level. Integrators can leverage concepts that span multiple Imaging API calls.

1.2.2.1 Identification

The NameID class provides the basic means of identifying an entity, such as an Application, Search, or Input, within the system. All entities have both a unique identifier which is numeric and a name which is represented by string data type. Either the numeric identifier or the name can be used to refer to specific entities within Imaging. The NameID class will hold the numeric, string, or both identifiers. If both identifiers are provided, the numeric ID is used. Id must be 0 (not provided) for Name to be used. This class is returned by all of the list() functions providing the caller with a list of both the numeric and string identifiers for the objects listed. The get() functions that accept a NameID allow the integrator to use either the numeric ID or the name to retrieve the desired entity.

1.2.2.2 Sections

Each object (application, input, search, connection, and document) is composed of multiple subobjects referred to as sections. Object sections include general properties, security grants, and audit history, among others. For convenience, the get() services accept an array of section indicators so that only a subset of the full object's content can be retrieved from Imaging or sent back to Imaging for update. These indicators are defined within the SectionFlag class of each object.

The following table lists the sections available for each definition object.

Section Flag	Application	Input	Search	Connection	Document
DESCRIPTION	X	X	X	X	
DETAILS				X	
DOCUMENTPERMISSIONS	X				
DOCUMENTSECURITY	X				
EXPRESSIONS			X		
FIELDDEFINITIONS	X				
FIELDVALUES					X
HISTORY	X	X	X	X	X
LIFECYCLEPOLICY	X				
MAPPINGS		X			
NAME	X	X	X	X	
PARAMETERS			X		

Section Flag	Application	Input	Search	Connection	Document
PERMISSIONS	X	X	X	X	X
PROPERTIES	X	X	X	X	X
RESULT_COLUMNS			X		
SECURITY	X	X	X	X	
SOURCE_PROPERTIES		X			
WORKFLOWCONFIG	X				

1.2.2.3 Properties

Each Imaging object contains a subobject called the properties object that defines the properties of that object. The properties object may contain additional subobjects that have their own attributes providing additional complex content.

1.2.2.4 Permissions

Many of the objects define a permissions subobject that specifies the permissions the current user has been granted in relation to the current instance of the object.

1.2.2.5 Security

The definition objects define security subobjects that are used to define what users or user groups have been assigned what privileges. These privileges cover the basic object maintenance actions such as creation, modification and deletion, as well as specific actions unique to the type such as search execution.

1.2.2.6 Audit Events

Many objects include a section for returning the audit history associated with that object. This history includes the actions taken by various users that have affected that object. Audited actions would include creation, modification, or viewing of that object. The audit history records the action that occurred, the user performing it, and the date upon which it occurred.

1.3 Requirements

Imaging clients not running on an Oracle WebLogic Server have the following requirements:

- JDK 1.6 or higher
- An installed and operational Imaging system

1.4 Sample of the Imaging Integration API

The following example is a simple demonstration of some of some basic functionality provided by the Imaging integration API. The example simply logs in to the Imaging system, lists viewable applications, and logs out.

Example 1-1 Listing Viewable Applications

```
package devguidesamples;

import java.util.List;
import java.util.Locale;

import oracle.imaging.Application;
import oracle.imaging.ApplicationService;
import oracle.imaging.BasicUserToken;
import oracle.imaging.ImagingException;
import oracle.imaging.NameId;
import oracle.imaging.ServicesFactory;
import oracle.imaging.UserToken;

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

      try { // try-catch
         UserToken credentials = new BasicUserToken("ipmuser", "ipmuserpwd");
         ServicesFactory servicesFactory =
            ServicesFactory.login(credentials, Locale.US, "http://ipmhost:16000/
imaging/ws");

         try { // try-finally to ensure logout
            ApplicationService appService = servicesFactory.getApplicationService();

            // List the viewable applications to confirm that "Invoices" exists
            List<NameId> appsList =
appService.listApplications(Application.Ability.VIEW);
            for (NameId appNameId: appsList) {
               System.out.println( appNameId );
            }
         }
         finally {
            if (servicesFactory != null) {
               servicesFactory.logout();
            }
         }
      }
      catch (ImagingException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Part II

Configuring the Imaging Client Side and Security

Part II contains the following chapters:

- Configuring the Class Path for the Imaging API
- Configuring Authentication and Security Policies

2 Configuring the Class Path for the Imaging API

This chapter describes how to configure the class path for the Imaging API, how to copy the API files to a client directory, and how to set the class path.
This chapter includes the following sections:

2.1 Configuring the Class Path for the Imaging API

The imaging API is contained in the .jar file *imaging-client.jar*. This is the only .jar reference required to obtain Imaging-specific classes for use in client code.

The imaging-client.jar file is dependent on a number of infrastructure classes for JAX-WS web services and Oracle Web Service Manager Security. These dependencies are automatically available when the client API is called from within an Oracle JRF enabled JEE container.

For standalone JSE clients, Imaging provides a bundled zip file containing all external dependencies required by imaging-client.jar. The procedure for configuring references in a standalone JSE environment is shown below. This procedure assumes that Oracle WebCenter Content is installed to $ORACLE_HOME on a server system, and the client system has no WebLogic or Oracle WebCenter Content components installed.

To configure references in a stand-alone JSE environment, do the following:

1. Copy $ORACLE_HOME/ipm/lib/ecm-client.zip from the server to a temporary directory on the client.
2. Extract ecm-client.zip to a directory on the client. The directory should contain a lib directory accessible to the client working environment. The .jar files contained within it will be referenced on the classpath at both compile and run time. A typical solution is to place it in a lib directory parallel to the client src and classes directories.)
3. Copy $ORACLE_HOME/ipm/lib/imaging-client.jar from the server to the lib directory on the client. For example, copy this into the same lib directory to which ecm-client was extracted.
4. In the classpath during compile and runtime, include imaging-client.jar and ecm-client.jar from with extracted ecm-client zip directory. For example, assuming imaging-client.jar and ecm-client.zip are both installed to a project lib directory within a typical Java project directory, the class path would appear as:

```
CLASSPATH = classes:lib/imaging-client.jar:lib/ecm-client.jar
```

The ecm-client.jar must be included from within the directory to which the ecm-client.zip was extracted, and all files in the zip must remain in their relative locations within that directory. The ecm-client.jar is a manifest only jar that references other jars using relative paths. Other jars contained in the zip need not be included on the classpath and only ecm-client.jar should be. Future releases may modify the contents of the zip, so referencing other unzipped jars is discouraged.

2.2 Copying the API Files to a Client Directory

To configure references in a stand-alone JSE environment, do the following:

1. Copy $ORACLE_HOME/ipm/lib/ecm-client.zip from the server to a temporary directory on the client.
2. Extract ecm-client.zip to a directory on the client. The directory should contain a lib directory accessible to the client working environment. The .jar files contained within it will be referenced on the classpath at both compile and run time. A typical solution is to place it in a lib directory parallel to the client src and classes directories.)
3. Copy $ORACLE_HOME/ipm/lib/imaging-client.jar from the server to the lib directory on the client. For example, copy this into the same lib directory to which ecm-client was extracted.

2.3 Setting the Class Path

In the classpath during compile and runtime, include imaging-client.jar and ecm-client.jar from with extracted ecm-client zip directory. For example, assuming imaging-client.jar and ecm-client.zip are both installed to a project lib directory within a typical Java project directory, the class path would appear as:

```
CLASSPATH = classes:lib/imaging-client.jar:lib/ecm-client.jar
```

The ecm-client.jar must be included from within the directory to which the ecm-client.zip was extracted, and all files in the zip must remain in their relative locations within that directory. The ecm-client.jar is a manifest only jar that references other jars using relative paths. Other jars contained in the zip need not be included on the classpath and only ecm-client.jar should be. Future releases may modify the contents of the zip, so referencing other unzipped jars is discouraged.

3

Configuring Authentication and Security Policies

This chapter provides an overview of how to configure authentication and security policies, how to provide SSL communication for basic authentication, how to apply OWSM security policies to Imaging web services, and how to reconfigure client-side security policies for Java API login.
This chapter contains the following sections:

3.1 About Configuring Authentication and Security Policies

Authentication and session management are handled differently depending on the integration method being used. When first installed, the Imaging Web Services are configured with no Oracle Web Service Manager security policies applied. When no security policies are applied, the services leverage either the HTTP Basic Authentication mechanism, or username token authentication. Note that Basic Authentication, where user credentials (user ID and password) are transmitted in the web service HTTP message header mechanism is not very secure because the user credentials are not encrypted in any way unless a Secure Socket Layer (SSL) transport mechanism is used.

3.2 Providing SSL Communication for Basic Authentication

If SSL is properly configured for the Imaging server instance, Imaging can be configured to force the use of SSL in all web service communication. This is done by setting the Imaging configuration MBean RequireBasicAuthSSL to true. By default, it is false.

> **Note:**
>
> The RequireBasicAuthSSL setting only applies when no HTTP Basic Authentication is in use because no OWSM security policies have been applied.

Using OWSM Security Policies

When higher degrees of security are desirable, Imaging web services support the following Oracle Web Services Management (OWSM) security policies.

- wss_username_token
- wss_username_token_over_ssl
- wss11_username_token_with_message_protection

When applying a security policy to the Imaging web services, remember that the same policy must be applied to all of the web services with the exception of the **DocumentContentService**. The DocumentContentService is designed to use streaming MTOM that is incompatible with OWSM security policies. Security for DocumentContentService first requires a separate, stateful login through the LoginService, which does leverage OWSM security policy. (This information is primarily significant for making direct web services calls. The proper login sequence occurs automatically when using the native Java API.)

3.3 Applying OWSM Security Polices to Imaging Web Services

Security policies are applied to Imaging web services from the WebLogic Server Administration Console using the following procedure.

1. Log in to Administration Console.
2. Click **Deployments**. The Summary of Deployments page is displayed.
3. Click the plus (+) icon next to **imaging** in the Name column of the Deployments table. The imaging deployment expands.
4. For each imaging web service under **Web Services** except *DocumentContentService*, do the following:
    a. Select the web service. The setting page for the service is displayed.
    b. Select the **Configuration** tab. The configuration tab becomes active.
    c. Select the **WS-Policy** tab. The WS-Policy tab becomes active.
    d. Click the web service port in the Service Endpoints and Operations column of the WS-Policy Files Associated With This Web Service table. The Configure the Policy Type for a Web Service page is displayed.
    e. Ensure **OWSM** is selected and click **Next**. Note that WebLogic polices are not supported. The Configure a WebService Policy page ID displayed.
    f. Choose a supported service policy from the Available Endpoint Policies field. Supported polices are listed in the Providing SSL Communication for Basic Authentication.
    g. Click the **right arrow** to move the selected policy to the Chosen Endpoint Policies field. Note that only one security policy should be selected.
    h. Click **Finish**. The Save Deployment Plan Assistant page is displayed.
    i. Click **OK** to save the deployment plan.
    j. Repeat step Applying OWSM Security Polices to Imaging Web Services for each web service except *DocumentContentService* until the same policy is applied for all services.
5. Click **Deployments** to return to the Deployments page.

6. Enable the check box next to **imaging** in the Name column of the Deployments table and click **Update**. The Update Application Assistant page is displayed with the new deployment plan specified next to **Deployment plan path**.
7. Click **Finish**. The new policies are applied and the deployment updated.

3.4 Reconfiguring Client-Side Security Policies for Java API Login

When OWSM security policies are applied to the Imaging web service, Java API code must use the WsmUserToken class to login rather than the BasicUserToken class. The WsmUserToken class is a helper class for configuring OWSM client side security polices, including a set of static constants for setting the correct client side policy. Depending on the policy being used, addition configuration setting may be required as well. Refer to OWSM document for complete details on the meaning of the various configuration options.

The code fragments in Example 3-1 demonstrate possible usages of the WsmUserToken class for various policy types.

Example 3-1 WsmUserToken Class for Various Policy Types

```
WsmUserToken userToken = new WsmUserToken ("weblogic", "weblogic");
userToken.setClientPolicy(WsmUserToken.USERNAME_TOKEN_POLICY);
ServicesFactory.login(userToken, wsurl);
```

Part III

Managing Documents in the Imaging Repository

Part III contains the following chapters:

4
Creating Documents

This chapter describes the basic tasks in Imaging, including how to create documents, list applications, get application properties and field definitions, upload document content, how to provide metadata for a document, and how to create a document sample.
This chapter includes the following sections:

4.1 About Creating Documents

The Imaging APIs ApplicationService and DocumentService interfaces are used to manage applications and documents in the imaging system. All documents must be associated with an imaging application. The application defines many default features that are then applied to the documents that are associated with the application. In this way, an application may be thought of as a template that defines a single document type. Typically, a single application is created for documents of a single type (for example, invoices, proposals, contracts, and so on).

This chapter details how to use basic methods used for creating documents in an application including mechanisms for list and getting existing application definitions and mechanisms for uploading and indexing documents. At the end of the chapter, Example 4-1 provides a code sample of the topics presented here.

4.2 Listing Applications

The Imaging API provides two mechanisms for enumerating the list of applications defined within and Imaging system: ApplicationService.listApplications and DocumentService.listTargetApplications. Each operation accepts an “Ability" parameter that specifies which applications to return based the user's security settings. However, they differ in one which security filter is used.

The ApplicationService.listApplications returns applications based on Application definition security. Ability.VIEW returns all applications to which the calling user has view permission. The Ability.MANAGE parameter returns all applications to which that user has either delete or modify permissions. The intended purpose of this operation is primarily for code written to manage application definitions.

The DocumentService.listTargetApplications returns applications based on the user's document permissions within the application, i.e., whether the user has either view

document or create document permissions for that application. This operation is the best choice when the client is working with directly documents.

Both of these operations return a java.util.List of NameId objects identifying both the numerical ID and the textual name for the application.

4.3 Getting Application Properties and Field Definitions

As with list applications, the Imaging API provides two distinct operations for getting details of the field defined for an application: ApplicationService.getApplication and DocumentService.getTargetApplication. These operations again differ by the permissions used to determine whether or not the user is allowed to get the requested application. ApplicationService.getApplication requires application view permissions. DocumentService.getTargetApplication requires document view permissions.

Application.getApplication operation is intended for use when managing the definition itself and provides a parameter for specifying which sections of the definition are desired. Sections are requested using a SectionSet, which is a container for passing in a list of SectionFlags. For example, in order to get the application's properties and field definitions sections, calling code would pass in a SectionSet defined as follows:

```
sectionSet =
   Application.SectionSet.of(Application.SectionFlag.PROPERTIES,
                             Application.SectionFlag.FIELDDEFINITIONS);
```

The DocumentService.getTargetApplication, however, returns a fixed section set that automatically includes both the properties and field definitions because these sections typically required when working with documents.

4.4 Uploading Document Content

Creating a document in an Imaging application is a two step process. In the first step, the documents binary data is uploaded to the Imaging server using the DocumentContentService.uploadDocument operation. This operation returns a unique token (and upload token) that is then used in a subsequent call to DocumentService.createDocument to index the document into the application. This uploadToken is valid until the user logs out or until it is used in either a createDocument or updateDocument call. It may only be used once.

The upload operation accepts data in the form a javax.activation.DataHandler, which in turn wraps the document content as a javax.activation.DataSource, typically a javax.activation.FileDataSource. The javax.mail.util package also contains a ByteArrayDataSource which can wrap the document content from and InputStream. However, the ByteArrayDataSource will load the entire document into memory on the client before performing the upload, so it should be used with caution for very large documents Please refer to the Javadoc for the javax.activation and javax.mail.util packages for complete details on the use of DataHandlers and DataSources.

4.5 Providing Metadata for a Document

The Document.FieldValue class in the Imaging API is used to provide document metadata when indexing a document. FieldValues are passed to createDocument as a java.util.List of Document.FieldValue instances. Each FieldValue in the list will map to a FieldDefinition in the application.

The Document.FieldValue object behaves similarly to NameId definition classes in that they can be defined to map to an application field definition by either Field ID or Field Name. If both are supplied, then the ID value supersedes the name value.

A Document.FieldValue also contains a value property. The type of the value must be compatible with the Imaging FieldType of the field definition. The Imaging type of the value is automatically determined by the Java type used. The following table lists the Imaging field types and the corresponding compatible Java types.

Imaging Field Type	Java Type
FieldType.Date	**java.util.Date**, java.util.GregorianCalendar
FieldType.Decimal	**java.math.BigDecimal**, float, decimal
FieldType.Number	**Integer**, Long,
FieldType.Text	**String**

In the table, the Java types in **bold** are the native types associated with the FieldType. The FieldValue will coerce other types in the table into the native type. Caution should be used when using types other than the native types since precision on the value may sometime be lost during the coercion process.

When FieldValues are use with createDocument, all field values that are defined as required must be supplied. For fields that are not required, it is also possible to deliberately set the value to *null* by including the FieldValue in the list but setting the FieldValue's value to null. When doing this, the FieldValue cannot determine the necessary field type based on the *null* Java type, so the FieldValue constructor accepting an Imaging FieldType must be used.

4.6 Create Document Sample

Example 4-1 demonstrates the basic concepts discussed in this section.

Example 4-1 text

```
package devguidesamples;

import java.math.BigDecimal;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Locale;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;

import oracle.imaging.Application;
import oracle.imaging.ApplicationService;
import oracle.imaging.BasicUserToken;
import oracle.imaging.Document;
import oracle.imaging.DocumentContentService;
import oracle.imaging.DocumentService;
import oracle.imaging.ImagingException;
import oracle.imaging.NameId;
import oracle.imaging.ServicesFactory;
import oracle.imaging.UserToken;
```

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

      try { // try-catch
         UserToken credentials = new BasicUserToken("ipmuser", "ipmuserpwd");
         ServicesFactory servicesFactory =
            ServicesFactory.login(credentials, Locale.US,
              "http://ipmhost:16000/imaging/ws");
         try { // try-finally to ensure logout
            DocumentService docService = servicesFactory.getDocumentService();
            DocumentContentService docContentService =
               servicesFactory.getDocumentContentService();
            NameId invoicesAppNameId = null;

            // List the viewable applications to confirm that "Invoices" exists
            List<NameId> appsList =
             docService.listTargetApplications(Document.Ability.CREATEDOCUMENT);
            for (NameId nameId: appsList) {
               if (nameId.getName().equals("Invoices")) {
                  invoicesAppNameId = nameId;
               }
            }
            if (invoicesAppNameId == null) {
               System.out.println("Invoices application not found.");
               return;
            }

            // Upload document content
            String fileName = "C:/PathToImages/invoice1234.tif";
            DataHandler fileData = new DataHandler(new FileDataSource(fileName));
            String uploadToken = docContentService.uploadDocument(fileData,
             "invoice1234.tif");

            // Index the document
            List<Document.FieldValue> fieldValues = new
               ArrayList<Document.FieldValue>();
            fieldValues.add(new Document.FieldValue("Invoice Number", 1234));
            fieldValues.add(new Document.FieldValue("Purchase Order", 4321));
            fieldValues.add(new Document.FieldValue("Vendor", "Acme Supply"));
            fieldValues.add(new Document.FieldValue("Amount", new
              BigDecimal("99.95")));
            fieldValues.add(new Document.FieldValue("Receive Date", new Date()));
            docService.createDocument(uploadToken, invoicesAppNameId,
               fieldValues, true);
         }
         finally {
            if (servicesFactory != null) {
               servicesFactory.logout();
            }
         }
      }
      catch (ImagingException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

5
Searching for Documents

This chapter has the following sections:

5.1 About Searching for Documents

The Imaging API provides the SearchService for enumerating and executing saved searches. The execution of a saved search requires the user to provide the NameId of the search and a List of SearchArguments objects for the search. The arguments form the variable portion of the WHERE clause in the search. These are passed to the SearchService. executeSavedSearch operation, which returns a Search.ResultSet. At the end of the chapter, Example 5-1 provides a code sample of the topics presented here.

5.2 Listing Saved Searches

The Imaging API provides the SearchService.listSearches methods for listing searches that are available for the logged in user. This operation accepts an **Ability** parameter that specifies which applications to return based the user's security settings. It returns applications based on search definition security. Ability.VIEW returns all searches to which the calling user has view permission. The Ability.MANAGE parameter returns all searches to which that user has either delete or modify permissions.

The operation returns a java.util.List of NameId objects identifying both the numerical ID and the textual name for the search.

5.3 Providing Search Arguments

A saved search may define one or more search parameters that Imaging will assemble into the WHERE clause for the search. In the Imaging API, the SearchArgument class is used to pass arguments for these parameters. The constructor for SearchArgument accepts the name of the parameter as well as SearchValue object specifying its value. The operator for the argument must be set using the SearchArgument.setOperator method.

Multiple SearchArguments are passed as a java.util.List. The order of the arguments in the list is not significant.

When calling SearchService.getSearchParameters and passing search NameId, SearchParameters are returned. This defines the parameters defined by the search and can help you build the SearchArguments necessary to execute the search. The SearchParameters also defines whether each parameter is required.

5.4 Executing a Search

The SearchService.executeSavedSearch method returns the results of the search in the form of a Search.ResultSet. The column labels are available in the Search.Result from the getColumns method, which returns an ordered java.util.List of Strings.

5.5 Parsing Search Results

The rows of the results are available from the Search.Results getResults method which returns a java.util.List of Search.Result objects. Each Search.Result represents a single document found by the search.

The search result columns are returned by the getColumnValues which is an ordered list of TypedValues. The order matches the order of the column labels list. System field values for the document are accessible from Document object returned by Search.Result.getDocument.

5.6 Execute Search Sample

Example 5-1 demonstrates the basic concepts discussed in this section.

Example 5-1 Sample Search Execution

```
package devguidesamples;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

import oracle.imaging.BasicUserToken;
import oracle.imaging.ImagingException;
import oracle.imaging.NameId;
import oracle.imaging.Search;
import oracle.imaging.SearchArgument;
import oracle.imaging.SearchService;
import oracle.imaging.SearchValue;
import oracle.imaging.ServicesFactory;
import oracle.imaging.TypedValue;
import oracle.imaging.UserToken;

public class ExecuteSearchSample {
   public static void main(String[] args) {
      try { // try-catch
         UserToken credentials = new BasicUserToken("ipmuser", "ipmuserpwd");
         ServicesFactory servicesFactory =
            ServicesFactory.login(credentials, Locale.US,
               "http://ipmhost:16000/imaging/ws");

         try { // try-finally to ensure logout
            SearchService searchService = servicesFactory.getSearchService();

            NameId invoiceSearchNameId = null;
```

```
                // List the viewable applications to confirm that "Invoices" exists
                List<NameId> searchList =
                   searchService.listSearches(Search.Ability.VIEW);
                for (NameId nameId: searchList) {
                   if (nameId.getName().equals("Invoices")) {
                      invoiceSearchNameId = nameId;
                   }
                }

                if (invoiceSearchNameId == null) {
                   System.out.println("Invoices search not found.");
                   return;
                }

                SearchValue searchValue = new SearchValue(SearchValue.Type.NUMBER, 1234);
                SearchArgument searchArgument =
                   new SearchArgument("Invoice Number", searchValue);
                searchArgument.setOperatorValue(Search.Operator.EQUAL);

                List<SearchArgument> searchArguments =
                   new ArrayList<SearchArgument>();
                searchArguments.add(searchArgument);

                Search.ResultSet resultSet =
searchService.executeSavedSearch(invoiceSearchNameId,searchArguments);

                // Display Column Headers
                System.out.print("DocumentId" + "  ");
                for (String column: resultSet.getColumns()) {
                   System.out.print(column + "  ");
                }
                System.out.println();

                // Display result Rows
                for (Search.Result row: resultSet.getResults()) {
                   System.out.println(row.getDocument().getId());
                   for (TypedValue typedValue: row.getColumnValues()) {
                      System.out.print(typedValue.getStringValue() + "  ");
                   }
                   System.out.println();
                }
             }
             finally {
                if (servicesFactory != null) {
                   servicesFactory.logout();
                }
             }
          }
          catch (ImagingException e) {
             System.out.println(e.getMessage());
          }
       }
    }
```

6
Retrieving Documents

This chapter has the following sections:

6.1 About Retrieving Documents

The Imaging API provides several document retrieval operations on the DocumentContentService. All of the retrieval methods accept the unique document identifier as input and return document content as a javax.activation.DataHandler.

When processing the DataHandler results, calling code should assume that the DataHandler is a wrapper around an open stream to the server and should process the results immediately, either by persisting the content to an output stream, such as a file, using the DataHandler.writeTo method, or by reading the input stream directly by using the DataHandler.getInputStream method. If the input stream is accessed directly, the caller must be sure to close the stream when the reading of the stream is complete.

Some of the retrieval methods that perform rendering operations on the original document in its native format, also accept additional parameters for controlling the rendering process. At the end of the chapter, Example 6-1 provides a code sample of the topics discussed.

6.2 Retrieving an Original Document

The original document is retrieved using the DocumentContentService.retrieve operation. This operation is the simple retrieval method and takes only the document ID as an argument. It returns the binary content of the document as a DataHandler. The Imaging render engine does no processing on the document in this case. Note, however, that this operation will return an exception if the document has secure annotations associated with it and the calling user does not have permissions to remove those annotations.

6.3 Retrieving a Rendition of a Document with Annotations

The document can be rendered along with its associated annotations using the DocumentContentService.retrieveRendition operation. This operation accepts a number of arguments including the documentId and a flag indicating whether or not to include annotations. It also includes a parameter for specify the page set that is to be rendered. Specifying multiple pages for this parameter will results in single, multi-page

document containing only those pages. Passing a null, or page zero, for this parameter will render the entire document. Refer the Javadoc for the retrieveRendition for complete details other parameters to the method.

The retrieveRendition method returns a Rendition class instance. The DataHandler containing the document content is accessible from the Rendition.getContent() method.

6.4 Retrieving Individual Pages from a Document

While the retrieveRendition method is used to return a single, possible multiple page rendition of the document, individual pages can be retrieved using the DocumentContentService.retrievePage. This method provides detailed control over how the page is rendered by accepting a RenderOptions class instance. RenderOptions provides options for specifying page rotation, page scaling, fit mode, output format, and others.

The retrievePage method returns a RenderResult class instance, which contains a List of RenderPage. The DataHandler content for page is accessible from RenderPage.getPageData().

6.5 Retrieve Document Sample

Example 6-1 demonstrates the basic concepts discussed in this section.

Example 6-1 Sample Document Retrieval

```
package devguidesamples;

import java.io.FileOutputStream;

import java.io.IOException;
import java.io.InputStream;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

import javax.activation.DataHandler;

import oracle.imaging.BasicUserToken;
import oracle.imaging.DocumentContentService;
import oracle.imaging.DocumentService;
import oracle.imaging.ImagingException;
import oracle.imaging.NameId;
import oracle.imaging.RenderOptions;
import oracle.imaging.RenderPage;
import oracle.imaging.RenderResult;
import oracle.imaging.Rendition;
import oracle.imaging.Search;
import oracle.imaging.SearchArgument;
import oracle.imaging.SearchService;
import oracle.imaging.SearchValue;
import oracle.imaging.ServicesFactory;
import oracle.imaging.UserToken;

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

```
      throws IOException {
      try { // try-catch
         UserToken credentials = new BasicUserToken("ipmuser", "ipmuserpwd");
         ServicesFactory servicesFactory =
            ServicesFactory.login(credentials, Locale.US, "http://ipmhost:16000/
imaging/ws");

         try { // try-finally to ensure logout
            SearchService searchService = servicesFactory.getSearchService();
            DocumentContentService docContentService =
                           servicesFactory.getDocumentContentService();

            // The find the document with invoice number 1234 using the Invoices
search
            List<SearchArgument> searchArguments = new ArrayList<SearchArgument>();
            SearchValue searchValue = new SearchValue(SearchValue.Type.NUMBER, 1234);
            SearchArgument searchArgument = new SearchArgument("Invoice Number",
searchValue);
            searchArgument.setOperatorValue(Search.Operator.EQUAL);
            searchArguments.add(searchArgument);
            Search.ResultSet resultSet =
               searchService.executeSavedSearch(new NameId("Invoices"),
searchArguments);
            if (resultSet.getTotalResults() == 0) {
               System.out.println("Document not found");
            }
            String documentId = resultSet.getResults().get(0).getDocumentId();
            String documentName =
resultSet.getResults().get(0).getDocument().getName();

            DataHandler fileData = null;
            FileOutputStream outputStream = null;

            // retrieve original native document content.
            fileData = docContentService.retrieve(documentId);
            outputStream = new FileOutputStream(documentName);
            fileData.writeTo(outputStream);
            outputStream.close();

            // Retrieve a document rendition with annotations
            Rendition rendition = docContentService.retrieveRendition(documentId,
                                                      true,
                                                      true,

RenderOptions.RenditionOutput.ORIGINALORTIFF,
                                                      null);

            fileData = rendition.getContent();
            outputStream = new FileOutputStream(documentName);
            fileData.writeTo(outputStream);
            outputStream.close();

            //Render a specific page to JPEG format.
            RenderOptions renderOptions = new RenderOptions();
            renderOptions.setPageNumber(2);
            renderOptions.setFormat(RenderOptions.OutputFormat.JPEG);
            RenderResult result = docContentService.retrievePage(documentId,
renderOptions);
            RenderPage page = result.getPages()[0];

            fileData = page.getPageData();
```

```
            outputStream = new FileOutputStream(documentName);
            fileData.writeTo(outputStream);
            outputStream.close();
         }
         finally {
            if (servicesFactory != null) {
               servicesFactory.logout();
            }
         }
      }
      catch (ImagingException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

7

Updating a Document

This chapter has the following sections:

7.1 About Updating a Document

The Imaging API provides the DocumentService.updateDocument operation for updating both a document's metadata as well as a document's content. This operation accepts both an uploadToken returned from a DocumentContentService.uploadDocument operation, and a List for FieldValue instances containing field value changes. Either or both of these parameters may be supplied meaning that the updated can either be a field value only update, a document content only update, or it can update both field values and content. At the end of the chapter, Example 7-1 provides a code sample of the topics presented here.

7.2 Uploading Revised Document Content

Updating document content in an Imaging application is a two step process. In the first step, the documents binary data is uploaded to the Imaging server using the DocumentContentService.uploadDocument operation. This operation returns a unique upload token that is then used in a subsequent call to DocumentService.updateDocument to index the document into the application. This uploadToken is valid until the user logs out or until it is used in either a createDocument or updateDocument call. It may only be used once.

7.3 Updating Metadata for a Document

The Document.FieldValue class in the Imaging API is used to provide document metadata when indexing a document. FieldValues are passed to udpateDocument as a java.util.List of Document.FieldValue instances. Each FieldValue in the list will maps to a FieldDefinition in the application.

The Document.FieldValue object behaves similarly to NameId definition classes in that they can be defined to map to an application field definition by either Field ID or Field Name. If both are supplied, then the ID value supersedes the name value.

A Document.FieldValue also contains a value property. The type of the value must be compatible with the Imaging FieldType of the field definition. The Imaging type of the value is automatically determined by the Java type used. The following table lists the Imaging field types and the corresponding compatible java types.

Imaging Field Type	Java Type
FieldType.Date	**java.util.Date**, java.util.GregorianCalendar
FieldType.Decimal	**java.math.BigDecimal**, float, decimal
FieldType.Number	**Integer**, Long,
FieldType.Text	**String**

In the table, the Java types in **bold** are the native types associated with the FieldType. The FieldValue will coerce other types in the table into the native type. Caution should be used when using types other than the native types since precision on the value may sometime be lost during the coercion process.

When FieldValues are use with updateDocument, the list need not contain every field defined in the document's application. Only those field values needing to change should be supplied. And field that is not supplied with be ignored when updating the document. For fields that are not defines as required, it is also possible to deliberately set a document field value to null by including the FieldValue in the list but setting the FieldValue's value to null. When doing this, the FieldValue cannot determine the necessary field type based on the *null* Java type, so the FieldValue constructor accepting an Imaging FieldType must be used.

7.4 Update Document Sample

Example 7-1 demonstrates the basic concepts discussed in this section:

Example 7-1 Sample Document Update

```
package devguidesamples;

import java.io.FileOutputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Locale;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;

import oracle.imaging.BasicUserToken;
import oracle.imaging.Document;
import oracle.imaging.DocumentContentService;
import oracle.imaging.DocumentService;
import oracle.imaging.ImagingException;
import oracle.imaging.NameId;
import oracle.imaging.RenderOptions;
import oracle.imaging.RenderPage;
import oracle.imaging.RenderResult;
import oracle.imaging.Rendition;
import oracle.imaging.Search;
import oracle.imaging.SearchArgument;
import oracle.imaging.SearchService;
import oracle.imaging.SearchValue;
import oracle.imaging.ServicesFactory;
import oracle.imaging.TypedValue;
```

```
import oracle.imaging.UserToken;

public class UpdateDocumentSample {
   public static void main(String[] args)
      throws IOException {
      try { // try-catch
         UserToken credentials = new BasicUserToken("ipmuser", "ipmuserpwd");
         ServicesFactory servicesFactory =
            ServicesFactory.login(credentials, Locale.US, "http://ipmhost:16000/
imaging/ws");

         try { // try-finally to ensure logout
            SearchService searchService = servicesFactory.getSearchService();
            DocumentService docService = servicesFactory.getDocumentService();
            DocumentContentService docContentService =
                   servicesFactory.getDocumentContentService();

            // The find the document with invoice number 1234 using the Invoices
search
            List<SearchArgument> searchArguments = new ArrayList<SearchArgument>();
            SearchValue searchValue = new SearchValue(SearchValue.Type.NUMBER, 1234);
            SearchArgument searchArgument = new SearchArgument("Invoice Number",
searchValue);
            searchArgument.setOperatorValue(Search.Operator.EQUAL);
            searchArguments.add(searchArgument);
            Search.ResultSet resultSet =
               searchService.executeSavedSearch(new NameId("Invoices"),
searchArguments);
            if (resultSet.getTotalResults() == 0) {
               System.out.println("Document not found");
            }
            String documentId = resultSet.getResults().get(0).getDocumentId();
            // update field values only.
            List<Document.FieldValue> fieldValues = new
ArrayList<Document.FieldValue>();
            fieldValues.add(new Document.FieldValue("Amount", new
BigDecimal("99.95")));
            docService.updateDocument(documentId, null, fieldValues, false);

            // update document content
            String fileName = "C:/PathToImages/NewInvoice1234.tif";
            DataHandler fileData = new DataHandler(new FileDataSource(fileName));
            String uploadToken = docContentService.uploadDocument(fileData,
"invoice1234.tif");
            docService.updateDocument(documentID, uploadToken, null, false);

            // update field values and document content at once
            fieldValues = new ArrayList<Document.FieldValue>();
            fieldValues.add(new Document.FieldValue("Receive Date", new Date())); //
now
            fileName = "C:/PathToImages/AnotherNewInvoice1234.tif";
            fileData = new DataHandler(new FileDataSource(fileName));
            uploadToken = docContentService.uploadDocument(fileData,
"invoice1234.tif");
            docService.updateDocument(documentID, uploadToken, fieldValues, false);
         }
         finally {
            if (servicesFactory != null) {
               servicesFactory.logout();
            }
         }
```

```
            }
            catch (ImagingException e) {
               System.out.println(e.getMessage());
            }
         }
      }
```

Part IV

Integrating Imaging Into Your Environment

Part IV contains the following chapters:

8

Using the Imaging API as Pure Web Services

This chapter has the following sections:

8.1 About Using the Imaging API as Pure Web Services

Although the Native Java API is the most convenient way to leverage Imaging services, it is mostly a set of proxy and utility classes that wrap calls to the Imaging web services set. All of the functionality exposed by the native Java API is available using direct web service calls as well. Use of the API through direct web services requires an in depth understanding of WSDL documents and of environment specific techniques for invoking web services.

8.1.1 Locating Web Service WSDLs

Understanding how to invoke Imaging web services starts with knowing where to access the WSDL documentation. The following is a complete list of the Imaging service WSDL locations. The host name and port number will vary depending on your installation.

- http://<*ipmhost*>:<*port*>/imaging/ws/ApplicationService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/ConnectionService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/DocumentContentService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/DocumentService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/ImportExportService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/LoginService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/PreferenceService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/SearchService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/SecurityService?wsdl
- http://<*ipmhost*>:<*port*>/imaging/ws/TicketService?wsdl

8.2 Using Web Services in Stateless Sessions

The majority of Imaging web services are capable of operating in either stateless or stateful mode. In stateless mode, authentication credentials passed in each service

request are used to transparently log the user in, perform the requested operation, and then log out before returning.

8.3 Using Web Services in a Stateful Session

In stateful operation, a call is first made to the *LoginService.login* operation to establish the user session with Imaging. Credentials to the login method are provided by the security policy currently in effect, or through HTTP Basic Auth if no policy is applied.

The jsessionid cookie returned by the log in operation is subsequently passed to call other services, thus maintaining session state from call to call. Note that web service security still requires that each call pass user credentials in order to comply with OWSM security policy enforcement. A call to LoginService.logout ends the user session.

As mentioned above, most Imaging services operate in either mode. The exception to this is the *DocumentContentService*. *DocumentContentService* operations are capable of leveraging a streaming Message Transmission Optimization Mechanism (MTOM) feature that is incompatible with OWSM security policies. Therefore, the stateful mode is required to wrap appropriate security around *DocumentContentService* operations.

8.4 Using the AXF Web Service

You can generate WSDL files for interfacing with the AXF Server services. The WSDL files provide the ability to pass data that can be understood by the AXF Server services, which enables access to the various commands within WebCenter Content.

8.4.1 Locating the AXF Web Service WSDL File

The AXF Web Service file can be found at the following location. The host name and port number will vary depending on your installation.

http://<*host*>:<*port*>/axf-ws2/AxfSolutionMediatorService?wsdl

> **Note:**
>
> The above URL is valid only for the IPM Server.

8.4.2 WSDL File Structure

WSDL files are formally structured with elements that contain a description of the data to be passed to the web service. This structure enables both the sending application and the receiving application to interpret the data being exchanged.

WSDL elements contain a description of the operation to perform on the data and a binding to a protocol or transport. This permits the receiving application to both process the data and interpret how to respond or return data. Additional sub elements may be contained within each WSDL element.

The WSDL file structure includes these major elements:

- **Data Types**: Generally in the form of XML schema to be used in the messages.

- **Message**: The definition of the data in the form of a message either as a complete document or as arguments to be mapped to a method invocation.
- **Port Type**: A set of operations mapped to an address. This defines a collection of operations for a binding.
- **Binding**: The actual protocol and data formats for the operations and messages defined for a particular port type.
- **Service and Port**: The service maps the binding to the port and the port is the combination of a binding and the network address for the communication exchange.

8.4.3 Data Type

The Data Type <types> defines the complex types and associated elements. Web services supports both simple data types (such as string, integer, or boolean) and complex data types. A complex type is a structured XML document that contains several simple types or an array of subelements.

The following code fragment for the AxfRequest set defines the CommandNamesapce, solutionNamespace request Parameters, UserContext and username elements and specifies that they are strings.

```
<xs:complexType name="axfRequest">
  <xs:sequence>
    <xs:element name="commandNamespace" type="xs:string"
    minOccurs="0" />
    <xs:element name="conversationId" type="xs:string"
    minOccurs="0" />
    <xs:element name="requestParameters">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="entry" minOccurs="0"
          maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="key" minOccurs="0"
                type="xs:string" />
                <xs:element name="value" minOccurs="0"
                type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="role" type="xs:string" minOccurs="0" />
    <xs:element name="solutionNamespace" type="xs:string"
    minOccurs="0" />
    <xs:element name="systemName" type="xs:string" minOccurs="0" />
    <xs:element name="userContext">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="entry" minOccurs="0"
          maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="key" minOccurs="0"
                type="xs:string" />
```

```
                <xs:element name="value" minOccurs="0"
                type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="username" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Similarly, the following code fragment for the AxfResponse set defines the ConversationId, Errorcode, ErrorMessage and response commands elements and specifies that they are strings.

```
<xs:complexType name="executeResponse">
  <xs:sequence>
    <xs:element name="response" type="tns:axfResponse"
    minOccurs="0" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="axfResponse">
  <xs:sequence>
    <xs:element name="conversationId" type="xs:string"
    minOccurs="0" />
    <xs:element name="errorCode" type="xs:string" minOccurs="0" />
    <xs:element name="errorMessage" type="xs:string"
    minOccurs="0" />
    <xs:element name="pid" type="xs:string" minOccurs="0" />
    <xs:element name="responseCommands" type="tns:responseCommand"
    nillable="true" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="responseCommand">
  <xs:sequence>
    <xs:element name="command" type="xs:string" minOccurs="0" />
    <xs:element name="value" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

8.4.4 Message

The Message <message> defines the data as arguments to be mapped to a method invocation.

```
<message name="execute">
  <part name="parameters" element="tns:execute" />
</message>
<message name="executeResponse">
  <part name="parameters" element="tns:executeResponse" />
</message>
```

8.4.5 Port Type

The Port Type <portType> defines a collection of operations for a binding. The DocInfo.wsdl file provides the DocInfoSoap and the DocInfo operation name (method name) with I/O information for processing the message.

```
<portType name="AxfSolutionMediatorWS">
  <operation name="execute">
    <input message="tns:execute" />
    <output message="tns:executeResponse" />
  </operation>
</portType>
```

8.4.6 Binding

The binding <binding> defines the actual protocol and data formats for the operations and messages for the particular port type.

```
<binding name="AxfSolutionMediatorPortBinding" type="tns:AxfSolutionMediatorWS">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="execute">
    <soap:operation soapAction="execute" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
```

8.4.7 Service and Port

The service <service> maps the binding to the port. The port is the combination of a binding and the network address for the communication exchange. The port is used to expose a set of port types (operations) on the defined transport.

```
<service name="AxfSolutionMediatorService">
  <port name="AxfSolutionMediatorPort" binding="tns:AxfSolutionMediatorPortBinding">
    <soap:address location="http://&lt;MachineName&gt;:&lt;Port&gt;/axf-ws2/
AxfSolutionMediatorService" />
  </port>
</service>
```

9 Integrating Imaging with BPEL

This chapter has the following sections:

9.1 About Integrating Imaging with BPEL

For additional information on workflow integration points, connection configuration, security and fault reporting, see Integrating with a Workflow in *Administering Oracle WebCenter Content: Imaging*.

9.2 Invoking Imaging Web Services from a BPEL Process

One possible integration point between Imaging and BPEL is a service call into BPEL from Workflow Agent to create a new process instance using document metadata stored in Imaging. It is also possible for the BPEL process to make web service calls back to Imaging. Such calls can retrieve the latest document metadata, additional metadata, or update document metadata to synchronize changes made during the execution of the process instance.

Calls to any Imaging web service from BPEL will use the same general procedure:

1. On the composite design diagram, select the Web Service service adapter from the **Component Palette** and drag it to the External References swim lane.
2. Enter a name for the service and the WSDL URL for the service. Imaging web service WSDLS take the form:

   http://<*host*>:<*port*>/imaging/ws/<*service*>?wsdl

   Where <*service*> would be one of the possible Imaging Web Service end points (ApplicationService, DocumentService, etc.).
3. On the composite diagram, link the BPEL process to the newly create external reference.
4. Open the BPEL process diagram. In this diagram, the external reference added to the composite will appear in the Partner Links swim lane.
5. Add an **Invoke** activity to the diagram and link it to the web service partner link.
6. In the invoke properties dialog, select the desired operation, and then click the **+** button next to the **Input** and **Output** variables to link the **Invoke** activity.
7. Click **OK**. The dialog box closes.
8. Add a Transform activity to the BPEL process diagram before the Invoke activity.

9. In the properties for the Transform Activity, select an appropriate source variable from the process, and select the input variable of the Invoke activity created in step 5 as the Target Variable.
10. Click the **+** next to the **Mapper File** to define the transformation mapping of process data into the web service input payload.
11. If output is expected to be returned from the Imaging web service invoke, a Transform or Assign activity can be used after the Invoke to transfer data from the Invoke output variable back into process variables.

9.3 Updating Imaging Metadata from a BPEL Process

As mentioned above, the transform definition from BPEL instance variables to Imaging web service input variables is a complex topic that is dependent on the specific schemas involved. However, because updating document metadata is a common use case for BPEL to Imaging interaction, Example 9-1 shows how the transform might be defined for a DocumentService.updateDocument operation used to modify document field values.

Example 9-1 Invoking DocumentService.updateDocument Sample

In this example, a purchase order document is indexed into Imaging and a BPEL process instance is created in an approval process. During the execution of the approval process, the instance is approved or denied. At the end of the process, Imaging needs to be updated with the approval status and the name of the user setting the status. This example assumes the following configuration.

Application Definition contains the following fields:

- PurchaseOrder: id=1, type string
- ApprovedBy: id=2, type string
- ApprovedStatus: id=3, type string

The BPEL Process variable is defined as follows:

```
<element name="process">
  <complexType>
    <sequence>
      <element name="docId" type="string"/>
      <element name="docURL" type="string"/>
      <element name="poNumber" type="string"/>
      <element name="approvedBy" type="string"/>
      <element name="approvedStatus" type="string"/>
    </sequence>
  </complexType>
</element>
```

An external partner link is defined using the document service WSDL:

```
http://host:port/imaging/ws/DocumentService?wsdl.
```

The updateDocument operation payload type is defined as follows:

```
<xs:complexType name="updateDocument">
  <xs:sequence>
    <xs:element name="documentId" type="xs:string" minOccurs="0"/>
    <xs:element name="uploadToken" type="xs:string" minOccurs="0"/>
    <xs:element name="fieldValues" type="tns:FieldValue"
```

```
          minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="updateAnnotations" type="xs:boolean"/>
  </xs:sequence>
</xs:complexType>
```

For modifying document field values, the elements that are significant are the documented elements and the fieldValues element. The FieldValue type and the type's references are defined as follows:

```
<xs:complexType name="FieldValue">
  <xs:complexContent>
    <xs:extension base="tns:baseId">
      <xs:sequence>
        <xs:element name="value" type="tns:TypedValue" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="id" type="xs:long" use="required"/>
    </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="TypedValue">
  <xs:simpleContent>
  <xs:extension base="xs:string">
    <xs:attribute name="type" type="tns:FieldType"/>
  </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="FieldType">
   <xs:restriction base="xs:string">
     <xs:enumeration value="TEXT"/>
     <xs:enumeration value="NUMBER"/>
     <xs:enumeration value="DECIMAL"/>
     <xs:enumeration value="DATE"/>
   </xs:restriction>
 </xs:simpleType>
```

Although the XSD for FieldValue looks rather complicated, the following sample XML instance of FieldValue demonstrates that it is actually fairly simple.

```
<fieldValues name="ApprovedStatus", id="3">
   <value type="TEXT">
    APPROVED
   </value>
</fieldValues>
```

The name and ID attributes in the fieldValues node are the name and ID of the document field value to be modified. In practice, providing one or the other is enough to identify the field. The value element provides the new value of the document field, and the type attribute, which is one of TEXT, NUMBER, DECIMAL, or DATE) indicates the Imaging type of data being provided.

The updateDocument type XSD indicates that the fieldValues has an unbounded maxOccurs attribute. In fact, the service expects that there be one instance of the element for each document field value that is being modified. Field Values that are not being modified need not be supplied.

Finally, the XSL transform in the BPEL process must assign the docId element from the BPEL instance variable to the documentId node in updateDocument and transform the ApprovedBy and ApprovedStatus values in the BPEL process variable into two FieldValue elements. The transform is defined as follows:

```
<xsl:template match="/">
  <tns:updateDocument>
    <documentId>
      <xsl:value-of select="/client:process/client:docId"/>
    </documentId>
    <xsl:for-each select="/client:process/client:approvedBy">
      <fieldValues>
        <xsl:attribute name="name">
          <xsl:text>ApprovedBy</xsl:text>
        </xsl:attribute>
        <value>
          <xsl:attribute name="type">
            <xsl:text>TEXT</xsl:text>
          </xsl:attribute>
          <xsl:value-of select="/client:process/client:approvedStatus"/>
        </value>
      </fieldValues>
    </xsl:for-each>
    <xsl:for-each select="/client:process/client:approvedStatus">
      <fieldValues>
        <xsl:attribute name="name">
          <xsl:text>ApprovedBy</xsl:text>
        </xsl:attribute>
        <value>
          <xsl:attribute name="type">
            <xsl:text>TEXT</xsl:text>
          </xsl:attribute>
          <xsl:value-of select="/client:process/client:approvedStatus"/>
        </value>
      </fieldValues>
    </xsl:for-each>
  </tns:updateDocument>
</xsl:template>
```

10

Accessing User Interface Functions Through URL Tools

This chapter has the following sections:

The examples used in this chapter assume that there is an existing Imaging application, that it has one or more saved documents, and that it has a search previously saved.

10.1 About Accessing User Interface Functions Through URL Tools

URL tools are a set of URLs in the Imaging user interface that provide direct access to specific user interface functions, such as executing a search or viewing a document. These tools are exposed through a specific access page and are supported as official APIs into the application. Imaging 11g URL tools are implemented in a manner similar to the previous Imaging 10g URL tools.

This section covers basic information about how URL tools are implemented in Imaging 11g. They are useful for determining what URL tools are installed on a specific server, or for maintaining installed URL tools. Because of the centralized nature of URL tool implementations in an imaging-ui project, it is easy to determine what tools are available on a particular code base. Additionally, through examination of the code, it is possible to determine the expected parameters and behavior of a particular URL tool.

10.2 Using URL Tools

The access point for the URL tools is currently the "UrlTools.jspx" page found in the following location: http://<*server*>:<*port*>/imaging/faces/Pages/UrlTools.jspx

The following tools are currently supported:

- **Search URL Tool**: provides direct access to the search UI tool.
- **Viewer URL Tool**: provides direct access to the viewer UI tool.
- **Upload URL Tool**: provides direct access to the upload UI tool.
- **User Preferences URL Tool**: provides direct access to the user preferences UI tool.

10.2.1 Supported URL Tool Parameters

The following URL parameters are used by all the tools and can be added as a parameter to any of the URL tools. Semicolon delimited Parameter Names indicate that multiple parameter names mean the same thing. This is for backward compatibility with the names used in Imaging 10g.

Parameter Name	Description	Valid Values	Default
HideBanner	Indicates the banner should be shown or not shown. By default, the banner is shown. Not used by the ViewDocument tool.	**1** or **true**: Hide the banner **0** or **false** (default): Show the banner	**0** - Show the banner on all URL tools unless otherwise noted
HideWorkcenter	Indicates the Navigation Pane of the UI should not be shown. Not used by the ViewDocument tool.	**1** or **true**: Hide the Navigation Pane **0** or **false**: Show the Navigation Pane	1 - Hide the Navigation Pane on all URL Tools (unless otherwise noted)
LockBanner	Indicates the banner should be locked or not locked. Locking the banner means that it is hidden and there is no way to expand it. HideBanner=0 will override the existence of this parameter on the URL.	**1** or **true**: Lock the banner **0** or **false**: Unlock the banner	0 - Unlock banner on all URL Tools (unless otherwise noted)
LockWorkcenter	Indicates the Navigation Pane should be locked or not locked. Locking the Navigation Pane means that it is hidden and there is no way to expand it. HideWorkcenter=0 will override the existence of this parameter on the URL.	**1** or **true**: Lock the Navigation Pane **0** or **false**: Lock the Navigation Pane	1 - Lock the Navigation Pane on all URL tools (unless otherwise noted)
ToolName	Indicates which tool should be used to process the request.	Currently there are only two tools. They are ExecuteSearch (or AWSER) and ViewDocument (or AWVWR).	None
skin	Indicates which skin to use.	Any value that is deployed with the application. Typically: • blafplus-rich • blafplus-medium • blafplus • fusion • fusion-11.1.1.3.0 • skyros	The current user's preference setting is used.

10.2.2 Supported URL Tools

The following URL tools are supported:

- Search URL Tool
- Viewer URL Tool

- Upload URL Tool
- User Preferences URL Tool
- Original Document Download Using the Native Viewer

10.2.2.1 Search URL Tool

The Search URL tool (ExecuteSearch) exposes the Search Results user interface as a directly accessible tool. The following is a summary of the URL parameters that are expected by the ExecuteSearch tool. Any remaining parameters are assumed to be field values that should be inserted into the search conditions.

Parameter Name	Description	Valid Values	Default
SearchId	This is the ID of the search that you want to run. If this is not specified then SearchName will be used. If there is no SearchId and no SearchName specified an error will be presented to the user.	Any values of Search Ids that exist in the Imaging system and the current user has access to.	None
SearchName	If SearchId is not found in the Parameters then SearchName will be looked for. If there is no SearchId and no SearchName specified an error will be presented to the user.	Any values of Search Names that exist in the Imaging system and the current user has access to. Note that the hexadecimal (base 16) code should be substituted for any special ASCII characters used in the search name. For example, an ampersand used in a search named **This&That** should be defined in the URL as http://machinename:port/imaging/faces/Pages/UrlTools.jspx?ToolName=AWSER&SearchName=This%26That	None
ClearSearches	Removes all the searches that have been run to this point in the session.	**1** or **true**: Clear the searches	False or 0

Parameter Name	Description	Valid Values	Default
_ipmOperator.<FieldName>	Specifies the Operator for the field identified by <FieldName>. For example a field named *Company* would use *_ipmOperator.Company*. This parameter is optional and only used when Picklist Operators are used in a search.	String values for the Search.Operator enum in the Java API. Options for text type fields include: • BEGINS_WITH • ENDS_WITH • EQUAL_TEXT • NOT_EQUAL_TEXT • CONTAINS Options for number, decimal, and date field types include: • EQUAL • GREATER_THAN • GREATER_THAN_OR_EQUAL • LESS_THAN • LESS_THAN_OR_EQUAL • NOT_EQUAL	If no value is provided then none is specified to the API and the API will behave as if none was specified.
ORAIPM_EMPTY_URL_PARAMETER	Removes the default value of a search field if one exists.	ORAIPM_EMPTY_URL_PARAMETER	None

When a parameter is entered as a search field in the URL, and that search condition allows the user to select the operator for the condition, then the operator defaults to the value specified for the parameter in the saved search unless a different one is entered by the user. Example 10-1 and Example 10-2 are sample URLs for running a search using the ExecuteSearch URL Tool.

Example 10-1 Running a Search

In this example, the search named Find HR Docs returns a result listing of all documents where the Employee Name field equals Jon Doe.

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?
ToolName=ExecuteSearch&SearchName=Find+HR+DOCS&EmployeeName=Jon+Doe
```

Example 10-2 Removing a Search Field Default Value

The ExecuteSearch search tool uses the default value of a search field if one exists. If you want to remove the default value you can add a URL parameter that indicates an empty URL parameter. You do this by specifying a value of "ORAIPM_EMPTY_URL_PARAMETER" for the parameter. For example, if the field "AText80" normally has a default value that should be removed when running the search, specify the following URL:

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?
ToolName=ExecuteSearch&SearchName=Search
%20Documents&AText80=ORAIPM_EMPTY_URL_PARAMETER
```

Note that by default the ExecuteSearch tool locks down the banner and the navigation pane of the Imaging user interface. This tool also locks down the form for the search results.

10.2.2.2 Viewer URL Tool

The Viewer URL tool exposes the Imaging Viewer UI tool as a directly accessible tool. The following is a summary of the URL parameters that are expected by the ViewDocument tool.

Parameter Name	Description	Valid Values	Default
showHistory	Causes the history pane of the viewer to be shown or hidden	**1** or **true**: Show History **0** or **false**: Hide history	False
showProperties	Causes the Properties pane of the viewer to be shown or hidden	**1** or **true**: Show properties **0** or **false**: Hide properties	False
showStickyNotes	Causes the Sticky Notes pane of the viewer to be shown or hidden	**1** or **true**: Show Sticky Notes **0** or **false**: Hide Sticky Notes	False
DocumentId	The document id of the document that should be shown to the user. If this is missing the view of the document will fail with an error.	Any valid document id obtained through searching or as a result of indexing a document through the user interface or the Web service API.	None
supportingKey	The key for the supporting information that should be shown	Any valid supporting information key value.	None
folder	The name of the folder the viewer should be placed in. This is an optional value that allows different sets of documents to be accumulated into segregated set in the UI cache.	Any string value that can be used as a name of a folder.	If this is not specified the folder name will be "default". This is the same folder the search results place viewed documents.
showTabs	Indicates if the tabs should be shown allowing the user to switch between documents in a folder.	1 or true - Hide the tabs 0 or false - Show the tabs	False
closeAllTabs	Tells the URL Tool to close all the tabs in the folder that the document will be opened in.	1 or true - Close all the documents in the folder. 0 or false - Do not close any of the documents in the folder.	False
forceHideProperties	Causes the Properties pane of the viewer to close. This option overrides the system default and any user preferences.	1 or true - Force close the Properties pane. 0 or false - Do not force close the Properties pane.	False

Parameter Name	Description	Valid Values	Default
forceHideStickyNotes	Causes the StickyNotes pane of the viewer to close. This option overrides the system default and any user preferences.	1 or true - Force close the StickyNotes pane. 0 or false - Do not force close the StickyNotes pane.	False
forceHideHistory	Causes the History pane of the viewer to close. This option overrides the system default and any user preferences.	1 or true - Force close the History pane.0 or false - Do not force close the History pane.	False
HideBanner	Causes the banner in the viewer to be shown or hidden	1 or true - Hide the banner 0 or false - Show the banner	True

Example 10-3 is a sample URL for opening a document in the viewer using the ViewDocument URL Tool.

Example 10-3 Opening a Document for Viewing

In this example, the document with the ID of 123.RPO_456 and is placed in a folder named EBS1. The parameter showTabs=0 suppresses the document tabs in the viewer to prevent users from switching to other documents in the folder.

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?
ToolName=ViewDocument&DocumentId=123.RPO_456&folder=EBS1&showTabs=0
```

10.2.2.3 Upload URL Tool

The Upload URL tool exposes the Upload user interface as a directly accessible tool. The following is a summary of the URL parameters that are expected by the UploadDocument tool. Any remaining parameters are assumed to be field values that should be used to automatically populate the Application Fields in the Upload form.

Parameter Name	Description	Valid Values	Default
AppId	This is the ID of the Application into which you want to index a document. If this is not specified then AppName will be used. If there is no AppId and no AppName specified an error will be presented to the user.	Any values of Search Ids that exist in the Imaging system and the current user has access to.	None
AppName	If AppId is not found in the Parameters then AppName will be looked for. If there is no AppId and no AppName specified an error will be presented to the user.	Any values of Search Names that exist in the Imaging system and the current user has access to.	None
ShowFieldsWithValues	Normally all fields that are specified on the URL will be removed from the Upload form. If this parameter is set to true then those fields will be displayed along with the other fields.	1 or true - Clear the searches 0 or false - Do not clear the searches	False

Example 10-4 and Example 10-5 are sample URLs used to upload a document using the UploadDocument URL Tool.

Example 10-4 Uploading a Document Using AppId

In this example the Upload tool loads and displays the application with an ID of 157. The fields MyTextField and MyNumberField are populated automatically and removed from the list of fields available to the user.

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?
ToolName=UploadDocument&AppId=157&MyTextField=Text%20Value&MyNumberField=123
```

Example 10-5 Uploading a Document Using AppName

In this example, the Upload tool loads and displays the Application with name of My App. The fields MyTextField and MyNumberField would be populated automatically but are not removed from the list of fields available to the user because the ShowFieldsWithValues parameter is set to true.

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?
ToolName=UploadDocument&AppName=My%20App&MyTextField=Text
%20Value&MyNumberField=123&ShowFieldsWithValues=true
```

By default the UploadDocument tool locks down the banner and the navigation pane of the Imaging user interface.

> **Note:**
>
> The required format for a date field is a standard ISO date and time format which is represented by the following:
>
> ```
> yyyy-MM-dd'T'HH:mm:ss.SSSZ
> ```
>
> For example:
>
> ```
> 2022-12-30T00:00:00.000-0700
> ```

10.2.2.4 User Preferences URL Tool

The User Preferences URL tool exposes the Preference Page user interface as a directly accessible tool. There are currently no UserPreferences URL parameters other than the tool name. Example 10-6 is a sample URL that can be used to access the User Preferences URL Tool.

Example 10-6 Loading User Preferences

This loads the User Preferences user interface tool showing all of the user's Imaging user preferences. When using the User Preferences Tool the *Close* button is removed from the user interface:

```
http://<server>:<port>/imaging/faces/Pages/UrlTools.jspx?ToolName=UserPreferences
```

10.2.2.5 Original Document Download Using the Native Viewer

There is an option to download and open the original document in its native viewer directly using a URL. This option is particularly useful for MS Office documents, especially Excel spreadsheets, or PDF documents containing embedded fonts which sometimes do not render as well in the Imaging viewer.

Example 10-7 is a sample URL for downloading a document in its native viewer.

Example 10-7 Downloading a Document in the Native Viewer

In this example, the document with the ID of 3.IPM_028745 is opened in its native viewer.

```
http://<server>:<port>/imaging/renderimage/originaldoc/3.IPM_028745
```

11
Making REST Paged Rendition Requests

This section explains how to format an URL to request single page renditions of documents via REST (Representational State Transfer). You can use REST requests to simplify HTML display of a document by allowing the REST URL to be embedded in the SRC attribute of an IMG tag. This is useful for displaying document pages regardless of native application or browser support.

Format the REST URL as shown below, using the parameters described in Table 11-1. Also see the example URLs provided.

11.1 REST URL Format

```
http[s]://<hostname:port>/imaging/renderimage[/#pct][/#deg][/fitmode][/crop#T#L#H#W]
[/annotations[true|false]][/page#][/download][/version#]/<documentId>.<format>
```

> **Note:**
>
> Include the parameter name in the same position (before or after) as specified in the REST URL format. For example, the number precedes the parameter name for *pct* and *deg* parameters, but follows the name for the *page* parameter.

REST URL Examples

Example URL	Description
http://myserver:16000/imaging/renderimage/IPM7_170001.JPEG	Retrieves a JPEG of the 1st page of document ID IPM7_170001
http://myserver:16000/imaging/renderimage/50pct/page2/IPM7_170001.PNG	Retrieves a PNG of the 2nd page of a document at 50% scale
http://myserver:16000/imaging/renderimage/fitbest/crop0T0L100H100W/page5/IPM7_170001.JPEG	Fits the 5th page of a document into a 100-pixel square
http://myserver:16000/imaging/renderimage/page5/download/IPM7_170001.GIF	Downloads page 5 of a document as a gif
http://myserver:16000/imaging/renderimage/180deg/IPM7_170001.JPEG	Displays the 1st page of a document rotated 180 degrees
http://myserver:16000/imaging/renderimage/20pct/270deg/annotationsfalse/page3/download/version2/IPM7_170001.JPEG	Downloads page 3 of version 2 of a document at 20% of its original height, rotated 270 degrees, with annotations turned off

Table 11-1 REST URL Parameters

Parameters	Description
pct	Specify the percent (as an integer) by which to scale the document. For example, 100 keeps the size unchanged, and 50 scales the document to 50% of its original size.
deg	Specify the rotation (in degrees) by which to rotate the document. Available rotations include: 0, 90, 180, and 270.
fitmode	To crop pages, specify a fit mode for this parameter (fitwidth, fitheight, or fitbest) and a crop rectangle using the crop parameter. The page will fit the fit mode based on the specified crop dimensions. (Note that this parameter is ignored if the crop parameter is not specified.)
crop	To crop pages, specify a crop rectangle's dimensions. The page will fit the specified fit mode (fitwidth, fitheight, or fitbest) based on the specified crop rectangle's dimensions. Use the format #T#L#H#W, which corresponds to a java.awt.Rectangle's top, left, height, and width properties. For example, for a rectangle where top=10, left=20, height=100, and width=200, specify the following: crop10T20L100H200W (Note that this parameter is ignored if the fitmode parameter is not specified or if fitmode is specified as fitscale.)
annotations	Specify true to apply annotations to the rendered page, or false to omit them.
page	Specify the page to display. For example, specify 2 to display the second page of a multi-page TIFF file. Note that the page is returned to the caller as raw page data. It includes header information that specifies the proper mime type and suggested file name.
download	If this parameter is included, the client downloads the file in the Open/Save/Cancel browser dialog box, and the rendition is zipped for return to the caller. If not included, the page is rendered only.
version	Specify the document version to render.
documentId	Specify the document ID to render (provided, for example, by the web interface).
format	Specify the format in which to render pages. Supported formats include: TIFF, PNG, GIF, and JPEG.