Oracle® Fusion Middleware

Developer's Guide for Content Integration Suite

11*g* Release 1 (11.1.1)

E10608-01

May 2010

Oracle Fusion Middleware Developer's Guide for Content Integration Suite, 11*g* Release 1 (11.1.1)

E10608-01

Primary Author: Will Harris

Contributor: Adam Stuenkel, David Wyman, Ron van de Crommert

Contents

Preface

The Oracle Fusion Middleware Developer's Guide for Content Integration Suite (CIS) provides general information on CIS, details on the Universal Content and Process Management API (UCPM API), an explanation of the various service APIs (SCS, SIS, and Common), information on extending commands, and instructions for migrating from previous versions.

Audience

This guide is intended for application developers and programmers. It includes conceptual and other information about the UCPM API, describes command invocation and execution, and explains how commands can be extended.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit `http://www.oracle.com/accessibility/support.html` if you are hearing impaired.

Related Documents

For more Oracle Enterprise Content Management Suite developer information, see the following documents in the Oracle ECM documentation set:

- *Oracle Fusion Middleware Content Integration Suite (CIS) Java API Reference*
- *Oracle Fusion Middleware Developer's Guide for Remote Intradoc Client (RIDC)*
- *Oracle Fusion Middleware Remote Intradoc Client (RIDC) Java API Reference*
- *Oracle Fusion Middleware JCR Adapter Guide for Content Server*
- *Oracle Fusion Middleware JCR Adapter Java API Reference*

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.
/	This guide uses the forward slash (/) to separate directories. Depending on your operating system, you may need to change the separation markers when defining directories.

1 Introduction

The Content Integration Suite (CIS) API offers access to Oracle Content Server by exposing the content server services and data in a unified object model. The Universal Content and Process Management (UCPM) API is modeled into a set of services APIs, which are API calls that communicate with the target server and the returned value objects from the server.

This chapter contains the following sections:

1.1 CIS Architecture

CIS has a layered architecture that allows for its deployment in a number of different configurations. The architecture, at its core, is based on the standard J2EE Command Design Pattern. The layers on top of the commands provide the APIs that are exposed to the end user.

CIS uses the Universal Content and Process Management (UCPM) API, which uses the SCS API for communication to Oracle Content Server. The SCS API wraps communication from the content server into an object model that allows access to the individual object metadata.

The UCPM API allows application developers to focus on presentation issues rather than being concerned with how to access content server services (IdcCommand services). It comprises a set of command objects which encapsulate distinct actions that are passed to the UCPM API and then mapped to the content server. These commands include common content management functions such as search, check-out, and workflow approval. Each command is tied to one or more service calls. The UCPM API command objects have been developed in accordance with the J2EE Command Design Pattern.

This infrastructure is deployable in any J2EE-compliant application server or stand-alone JVM application. When deployed, the UCPM API leverages the features in the environment, whether this is a J2EE application server or non-J2EE server.

The UCPM API encapsulates content server business logic and validates the parameters of the incoming calls. It also handles communication with the content server, encapsulates socket communication logic (opening, validating, and streaming bits through the socket), and provides a strongly typed API to the available services.

1.2 Internationalization / Character Encoding

Oracle recommends that encoding for CIS should be set to the same encoding as the Java Virtual Machine running Oracle Content Server. However, if CIS is communicating with multiple Oracle Content Server instances in different languages, then the ISCSContext.setEncoding method can be used to set the encoding to match that of the JVM running CIS.

1.3 Deprecated FixedAPI

The Fixed API available in pre-11*g* releases of CIS for communication with the Image Server has been deprecated. Calling getFixedAPI() throws an error.

2 Understanding the UCPM API

The Universal Content and Process Management (UCPM) API offers access to Oracle Content Server instances by exposing their services and data in a unified object model. The UCPM API is modeled into a set of services APIs that communicate with the target server.

This chapter contains the following sections:

Note: Refer to *Oracle Fusion Middleware Content Integration Suite (CIS) Java API Reference* for information on the Class/Interface, Field, and Method descriptions.

2.1 Accessing the UCPM API

The UCPM API offers access to Oracle Content Server instances by exposing their services and data in a unified object model. The UCPM API is modeled into a set of services APIs, which are API calls that communicate with the target server, and into ICISObject objects, which are the value objects returned from the server.

The UCPM API is available on the ICISApplication class via the getUCPMAPI() method. The getUCPMAPI() method returns a reference to the IUCPMAPI object, allowing access to all UCPM API objects. The public interface IUCPMAPI is the locator for the getActiveAPI object; **getActiveAPI()** returns a reference to the SCSActiveAPI object. The SCS API classes communicate with, and handle content stored on, the content server.

2.2 UCPM API Methodology

The UCPM API is stateless; all method calls pass in the necessary state to the method. This means that you can share the reference to the CISApplication class across threads.

- **ISCSContext** for the SCS API. The ISCSContext interface is the context object used when communicating with the content server.
- **ICISCommonContext** for calling some of the CIS APIs. The ICISCommonContext interface identifies which adapters to query and what user information to use.

The first parameter for all methods is an IContext bean. The IContext bean holds context information, such as username and session ID, which is used in the underlying service APIs to identify the user invoking the given command.

The UCPM API is a service-oriented API that returns value objects, implemented as ICISObject objects (name changed from the 7.6 API). However, calling methods on the value objects themselves do not modify content on the server; one must call the UCPM API and pass in the value object as a parameter before the changes can be applied.

Example:

```
SCSActiveAPI activeAPI = m_cisApplication.getUCPMAPI ().getActiveAPI ();
ISCSDocumentID documentID = (ISCSDocumentID) m_cisApplication.getUCPMAPI ().
  createObject(ISCSDocumentID.class);
documentID.setDocumentID("10");
ISCSDocumentInformationResponse docResponse =
  activeAPI.getDocumentInformationAPI ().
  getDocumentInformationByID(m_context, documentID);
ISCSContent content = docResponse.getDocNode();

// call does not change object on server
  content.setTitle ("New Title");

// now item is updated on server after this call
  activeAPI.getDocumentUpdateAPI ().updateInfo (m_context, content);
```

2.3 CIS Initialization

Content Integration Suite (CIS) is initialized by accessing the CISApplicationFactory class, which resides in the com.stellent.cis.impl package.

This section contains the following topics:

- "Initialization" on page 2-2
- "SCSInitializeServlet" on page 2-3

2.3.1 Initialization

CIS initialization should happen once per application. The CIS APIs are stateless and the initialized CISApplication instance can therefore be safely shared between threads.

To initialize CIS, a number of properties must be defined. The cis.config.type should be `server` and the cis.config.server.type should be `standalone`. The adapter configuration file contains XML-formatted configuration information for communicating with the content servers.

Initialization Examples

Initializes the system and reads the adapterconfig.xml file from the classpath:

```
cis.config.type=server
cis.config.server.type=standalone
cis.config.server.adapterconfig=classpath:/adapterconfig.xml
```

Code example:

```
ICISApplication application;
  URL xmlRes = new File ("adapterconfig.xml").toURL()
  Properties properties = new Properties();
  properties.setProperty(ICISApplication.PROPERTY_CONFIG_TYPE, "server");
  properties.setProperty(ICISApplication.PROPERTY_CONFIG_SERVER_ADAPTER_CONFIG,
    xmlRes.toExternalForm());

properties.setProperty(ICISApplication.PROPERTY_CONFIG_SERVER_TYPE, "standalone");
application = CISApplicationFactory.initialize(properties);
```

Property Definitions

The properties are defined in the following table.

Property	Description
cis.config.type	Should be set to `server`.
cis.config.server.type	Should be set to `standalone`.
cis.config.server.adapterconfig	The URL pointing to the adapter configuration file. In addition to standard URLs, this can be in classpath form (classpath:/) or file form (file:/)
cis.config.server.temporarydirectory	The location of the temporary directory used for file transfers, streaming, and file caching.

2.3.2 SCSInitializeServlet

The SCSInitializeServlet is a convenient way to initialize a CISApplication instance from within a web application. Any of the properties described can be used by the SCSInitializeServlet. The SCSInitializeServlet can be configured externally via a properties file. The `cis.initialization.file` property can be set with a path (either a web-relative path or a classpath reference), to a property file containing the initialization properties. This allows you to easily externalize the initialization to a properties file.

By default, if SCSInitializeServlet finds no properties in the web.xml file, it will attempt to load a properties file from the WAR and then from the classpath using the default value `/cis-initialization.properties`. Thus, if you place a file called cis-initialization.properties in your classpath (that is, in the same directory as the adapterconfig.xml file), that file will be read during startup.

This properties file can hold all the standard initialization properties as defined in the CISApplication class. This allows you to move the configuration of how CIS initializes outside the scope of the EAR/WAR file.

Server Property Definitions

The server properties are defined below. The defaults can be overridden by creating a file named cis-initialization.properties and saving the file to the server-ear directory of the unbundled CIS distribution file (this is the directory containing the adapterconfig.xml file).

Property	Description
cis.config.type	Must be set to server (default).
cis.config.server.adapterconfig	The URL pointing to the adapterconfig.xml file. In addition to standard URLs, this can also be in classpath form (that is, classpath:/)
cis.config.server.type.options.ejb=true	Default is true (EJBs enabled).
cis.config.server.type.options.rmi=true	Default is true (MI enabled).

Initialization Process

At startup, the SCSInitializeServlet servlet begins the CIS server initialization process. It attempts to load various properties from the web.xml file and classpath (that is, cis-initialization.properties). It then passes those properties to the static method CISApplicationFactory.initialize(…). This section describes the order of operation.

SCSIntitialize init(ServletConfig)

Called by the web application container during initialization of the web application.

1. Load properties from web.xml.
2. Load properties from classpath: /cis-initialization.properties.
3. Call CISApplicationFactory.initialize(properties).
4. Via the CISWebHelper class, it sets the CISApplication and the command application instance as an attribute on the servlet context.

CISApplicationFactory initialize(properties)

Called by the init(…) method of an object instance of SCSInitialize.
Calls CISApplicationFactory.initializeCisApplication(properties).

CISApplicationFactory initializeCisApplication(properties)

Determines if CIS should be started in client or server mode.
Calls CISApplicationFactory.initializeServer(properties).

CISApplicationFactory initializeServer(properties)

Called by CISApplicationFactory.initialize(properties).

1. Creates the adapter config URL (used to eventually load the adapterconfig).
2. Loads IsolatedJarClassloader.
3. Calls CISApplication.initialize(properties).

2.4 Integration in a Web Environment

This is what you would put in the web.xml if you wanted the SCSInitializeServlet to start up and register the CIS application:

```
<servlet id="scsInitialize">
  <servlet-name>scsInitialize</servlet-name>
  <display-name>SCS Initialize Servlet</display-name>
  <servlet-class>com.stellent.cis.web.servlets.SCSInitializeServlet
    </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

If you add a new JavaServer Page called search.jsp, it would look like the following:

```
<%-- JSTL tag library --%>
<%@ taglib uri="/WEB-INF/tlds/c.tld" prefix="c" %>

<%--  CIS Application object placed in servlet context by the SCSInitialize
servlet. Get the CIS Application and make a query --%>

<%
ICISApplication cisApplication =
  (ICISApplication) request.getSession().getServletContext().
    getAttribute ("CISApplication");
ISCSSearchAPI searchAPI =
  cisApplication.getUCPMAPI ().getActiveAPI ().getSearchAPI ();

// create a context
ISCSContext context =
  cisApplication.getUCPMAPI ().getActiveAPI ()._createSCSContext ();
  context.setUser ("sysadmin");

// execute the search
ISCSSearchResponse response =
  searchAPI.search (context, "dDocAuthor <substring> 'sysadmin'", 20);
%>
<!-- model the search results as desired -->
```

The SCSInitializeServlet places the initialized CISApplication class as an attribute in the javax.servlet.ServletContext with the name CISApplication. The CISApplication instance is available to the entire web application and is thread-safe; the one instance can be shared across the application.

2.5 Classloading

The UCPM 8.0.0 API uses a custom class loader to isolate dependencies on specific libraries from any application that uses the CIS API. Refer to the Java 2 Platform API specification for more information:

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html

This section contains the following topics:

- "Custom Class Loader" on page 2-5
- "Classloader Usage" on page 2-6

2.5.1 Custom Class Loader

Our implementation of this paradigm is found in the com.stellent.cis.impl.IsolatedJarClassLoader object. This custom class loader allows for a jar to have a nested set of jar files that serve as the library. This is the structure of the nested jar files:

```
+ cis-application-8.0.0.jar
    +-- com/stellent/cis/...
    +-- META-INF
    +-- lib/
        +-- spring-1.1.5.jar
        +-- log4j-1.0.3.jar
        +-- ...
```

All libraries that live in the /lib directory will be the classpath for the CIS objects. The classloader will query this local directory first before loading files from the parent classloader.

However, all of the jar files cannot be isolated as the consuming client needs to have access to some API classes so they can be imported into their application space. Therefore, only our interfaces are exposed into the application space; keeping our implementation and associated dependencies isolated. Since the class loader for the dependencies and implementation has access to the parent loader, it can access the same version of the interfaces as the application using the APIs. This also implies that a client will only be able to access the interfaces of the UCPM APIs. Any attempt to create an implementation class using the new keyword will result in a ClassNotFoundException.

2.5.2 Classloader Usage

To use this classloader, use the new com.stellent.cis.impl.CISApplicationFactory object to initialize the system. This object will automatically detect and use the IsolatedJarClassLoader if required. This means that you can still deploy CIS in the original format, with all the class files and libraries at the application level, if you so desire.

In the current version, CIS only needs the cis-client-8.0.0.jar file in the classpath; no other libraries are needed. Once cis-client-8.0.0.jar is in the application classpath, you initialize CIS using the following code:

```
// the initialization properties (as defined in ICISApplication)
  Properties properties = new Properties ();
  ICISApplication cisApplication = CISApplicationFactory.initialize(properties);
```

Once you have a reference to com.stellent.cis.ICISApplication, you can interact with the APIs. The difference is that now you only have access to interface objects (everything in com.stellent.cis.client) and not the implementation objects.

If you implement custom commands, your public interfaces will also need to be in the classpath, and the implementation classes packaged in the cis-client-8.0.0.jar.

2.6 Object Creation

The UCPM 8.0.0 API uses a customized classloader to hide library dependencies and implementation classes. Only the client interface classes are exposed to the user. However, this implies that you cannot use the new keyword to instantiate UCPM API objects. Therefore, in the UCPM API framework, use the generic `_create` methods available on the IUCPMAPI object to tell the system to instantiate an instance of the given object.

Since objects are mutable and only the interfaces are exposed, use the createObject method and set the properties that you need:

```
ISCSDocumentID docID =
  (ISCSDocumentID)ucpmAPI.createObject(ISCSDocumentID.class);
docID.setDocumentID("20");
```

2.7 Interacting With the UCPM API

With an initialized CISApplication instance, the getUCPMAPI () method (of the CISApplication object) returns a reference to the IUCPMAPI object, allowing access to all UCPM API objects. The IUCPMAPI interface is the locator for the various API objects in the UCPM API. The IUCPMAPI object has methods to get references to the Active API; **getActiveAPI ()** returns a reference to the SCSActiveAPI object for communicating with the content server. This allows you to access the necessary APIs and begin making calls through the UCPM API to the target server.

Calling API Objects Using Newly Instantiated ICISObject Objects

Many UCPM API calls take in an ICISObject or an interface that inherits from ICISObject. For example, in the ISCSDocumentInformationAPI (SCS API) the getDocumentInformationByID () method takes as a parameter a ISCSDocumentID object.

The fully qualified method name is:

```
ICISApplication.getUCPMAPI ().getActiveAPI ().getDocumentInformationAPI ()
```

In such cases, to obtain a reference to a valid object to pass in as a parameter, you can either retrieve the object reference from another ICISObject or create a new instance of the ICISObject using the generic createObject method available in the UCPM API. In CIS 11*g*R1, a customized classloader is used to hide our library dependencies and implementation classes. Only our client interface classes are exposed to the user. However, this implies that you cannot use the new keyword to instantiate UCPM API objects. Therefore, in the CIS API framework, you use the generic create method available on the IUCPMAPI object to tell the system to instantiate an instance of the given object. The createObject method will let you create a new instance of any ICISObject.

For example, if you wanted to query for document information, but only had the document ID, you would do the following:

```
ISCSDocumentInformationAPI documentInfoAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentInformationAPI ();

// create the document ID
ISCSDocumentID documentID =
  (ISCSDocumentID) m_cisApplication.getUCPMAPI ().
    createObject (ISCSDocumentID.class);
  documentID.setDocumentID("12345");

ISCSDocumentInformationResponse docResponse =
  documentInfoAPI.getDocumentInformationByID(m_context, documentID);
```

For any API that requires an ICISObject, you can use the createObject method that allows you to create a new instance of the API object. The createObject methods is a client-side method; it does not make a call to the target server. It is to be treated as a constructor for the ICISObject implementations.

If you needed to build up a more complex object such as a new content item to be checked into the content server, you would need to create several objects and populate the data:

```
// Create an empty content object
ISCSContent content =
  (ISCSContent) m_cisApplication.getUCPMAPI ().createObject(ISCSContent.class);

// Create an empty content ID object, and then give it a content ID
ISCSContentID contentID = (ISCSContentID) m_cisApplication.getUCPMAPI ().
  createObject(ISCSContentID.class);
  contentID.setContentID("my_document");
```

```
// Set all of the properties of the content item required for check in
  content.setContentID(contentID);
  content.setAuthor (context.getUser ());
  content.setTitle ("Document Title");
  content.setSecurityGroup ("Public");
  content.setType ("ADACCT");
  content.setProperty ("xCustomProperty", "Value for custom property");
```

2.8 Interface IContext

The interface IContext is the generic context used for communication with the Command APIs. This interface handles contextual information to determine the current caller identity, the target adapter, and so on.

The context should be populated with the username and adapter name. The adapter name is determine by the adapterconfig.xml file and the username can be any valid user ID for the target server.

IContext has the sub-interfaces SCSContext (ISISContext extends the IContext interface); **ISCSContext** is the context object used for the SCS APIs and represents a users operating context when communicating with the content server.

The context object can be created by using the `_create` method. Thus, ISCSContext can be created from SCSActiveAPI.

```
// create an ISCSContext
  ISCSContext context =
  m_cisApplication.getUCPMAPI ().getActiveAPI ()._createSCSContext ();
```

Once the context is created, it should be populated with a username and the adapter name. This can be done by using the accessor methods on the IContext bean.

```
  context.setUser ("sysadmin");
  context.setAdapterName ("myadapter");
```

The CIS API will take either an ICISCommonContext or an IContext object. ICISCommonContext is a special kind of context that is used as a container for ISCSContext and ISISContext. It is required in APIs that federate information between a number of different adapters; it identifies which adapters to query and what user information to use. In instances where the call only operates against one adapter at a time, a single IContext is required.

```
// create an ICommonContext
  ICISCommonContext m_commonContext =
  m_cisApplication.getUCPMAPI ().getCommonAPI ()._createCommonContext ();
```

Once the ICISCommonContext adapter is created, multiple adapters can be added to it. This is done using the ICISCommonContext.addContext() method. Any number of adapters can be added; all the adapters added to the ICISCommonContext are then used individually during a Common API call.

The same ISCSContext object can be used for multiple queries and across threads. In a web application context, the easiest method is to add the IContext object to the session and retrieve it from the session for each query.

An sample web application has been provided (located in /SDK/Samples/WebSample) which has a login method that first validates the username against the content server and, if successful, adds the IContext object to the HttpSession object. Refer to the LoginActionHandler class in the src/com/stellent/sdk/web/sampleapp/handlers/login directory for more details.

2.9 Interface ICISObject

The interface ICISObject is the base interface for all objects with metadata in the UCPM API. Thus, all UCPM API objects are inherited from ICISObject. The interface ICISObject allows for the retrieval and setting of the object properties. The objects returned from calls to the UCPM API are value objects in the form of beans (reusable software components) that encapsulate data from the server call, not live objects. Updating or modifying the objects in any fashion will not affect server data; only by directly calling a method on a given UCPM API can the server data be modified.

This section contains the following topics:

- "Property Accessors" on page 2-9
- "Property Object Types" on page 2-10
- "Property Collections" on page 2-10

2.9.1 Property Accessors

Most implementations of ICISObject have their own specific property accessor methods. However, all properties can be retrieved by calling the getProperty () method on the ICISObject. The ICISObject.getProperty () method will return an ICISProperty object. From this object you can get the property value or property information using these methods:

- **getValue()** returns the property value. If the property has a null value, calling getValue () will result in a null reference.
- **getDescriptor()** returns the property descriptor that describes the contents of the property value.

```
// use the response object from the previous example - retrieve the content object
  ISCSContent content = docResponse.getDocNode ();

// get the title property
  String title = content.getTitle ();

// get the title by using the ICISObject getProperty method
  title = content.getProperty ("title").getValue ().getStringValue ();
```

The ICISObject property methods may throw a PropertyRetrievalException if an error occurs during the lookup of a given property. Since the PropertyRetrievalException is a RuntimeException, it does not have to be caught directly in your code. Common cases for the exception to be thrown is when a property is asked for but does not exist or when the property value contains invalid data. You can catch this exception and query the exception class for more details on the specific reason for the error.

ISCSProperty also allows for setting property values back into the property object. To do this, you can use an appropriate set method or call setProperty () and pass in the bean property name (both are valid and both will set the property value on the target object).

```
// set the title - using the content object from the previous example
  content.setTitle ("My New Title");

// set using the setProperty method
  content.setProperty ("title", "My New Title");
```

2.9.2 Property Object Types

A property object type is determined by the return value of the property method on the ICISObject. When using the generic getProperty() method, the ISCSPropertyValue has methods to get both the value of the property and the value as a specific object type (for example, boolean, float, long, and so on).

The ISCSPropertyValue is retrieved via the getValue() method on the returned ISCSProperty object.

When setting a property via the generic setProperty() method, it is important that the property value passed into the method is of the correct type or can be converted to the appropriate type via simple BeanUtils property conversion.

Apache BeanUtil is a utility for populating bean properties from the org.apache.commons project.

If we take a look at the ISCSContent object, the property readOnly is type boolean. Therefore, in the following example, the first three methods will successfully set the property value and the last method will not:

```
// correct
  content.setReadOnly (true);
  content.setProperty ("readOnly", Boolean.TRUE);
  content.setProperty ("readOnly", "true");

// incorrect
  content.setProperty ("readOnly", "not a boolean");
```

Since the setProperty () method takes an object as the second parameter, the boolean encapsulation must be used. Also, as mentioned, the method uses the BeanUtils property conversion and therefore the string `true` converts to the boolean value TRUE. As shown in the example above, passing a property value that cannot be converted (for example, not a boolean) will result in an exception.

2.9.3 Property Collections

The available list of properties can be retrieved using the getProperties() method on the ICISObject interface. This will return all of the available properties for a given object.

```
// using the content item from the previous example
  Collection properties = content.getProperties ();

// iterate through the collection
for (Iterator it = properties.iterator (); it.hasNext (); ) {
  ISCSProperty property = (ISCSProperty)it.next ();
  String name = property.getDescriptor ().getName ();
  ICISPropertyValue value = property.getValue ();
  if (value != null) {
     System.out.println (name + " = " + value.getStringValue ());
  }
}
```

2.10 Adapter Configuration File

The adapter configuration file (adapterconfig.xml) contains XML-formatted configuration information for communicating with your Oracle Content Server instance. It specifies for the CIS layer which servers to open communications with.

A single connection to a server is called an adapter; any number of adapters can be configured in the adapterconfig.xml file. The adapterconfig.xml file is required to initialize the CISApplication instance.

This section contains the following topics:

- "Adapter Element" on page 2-11
- "Config Element" on page 2-11

2.10.1 Adapter Element

Each adapter configuration is a separate element in the XML markup. The adapter element has four attributes as shown in the following table:

Adapter Attributes	Description
type	Should be `scs` for a connection to Oracle Content Server.
default	If true, then this is the default adapter for this type. Only one default adapter for a given type is allowed.
name	The adapter name.

A sample adapter element is shown below:

```
<adapter type="scs" default="true" name="myadapter">
```

2.10.2 Config Element

The config element includes a set of property elements that define the adapter-specific properties. These configuration elements are explained below.

SCS Adapter Configuration Elements

An SCS adapter communicates with the content server. The configuration element for the SCS adapter has four general attributes as shown in the following table:

Property Name	Description
post	The port of the content server.
host	The hostname or IP address of the content server.
type	These values may be used: **socket**: Uses the content server socket communication layer. **mapped**: Uses shared directories to transfer the files for file upload and download. **web**: Uses HTTP requests to transfer files; requires a content server username and password for Basic HTTP Authentication (file download only).

A sample SCS configuration element is shown below:

```
<adapter name="myadapter" type="scs" default="true">
  <config>
    <property name="host">localhost</property>
    <property name="port">4444</property>
    <property name="type">socket</property>
    <property name="version">75</property>
  </config>
  <beans template=
  "classpath:/META-INF/resources/adapter/adapter-services-scs.jxml"/>
</adapter>
```

By default, the content server socket communication layer is used to stream files to and from the content server. However, for high-volume check in or file retrieval, you can set the **mapped** or **web** optimized file transfer options.

A mapped transfer loads the files from a shared directory on the content server; this results in much faster file transfers and does not tie up a socket that could be used for other requests. To use mapped transfer, you must define these properties:

Property Name	Description
contentServerMappedVault	The content server vault directory as seen from the application server.
appServerMappedVault	The application server vault directory as seen from the content server.

A web transfer uses HTTP requests to the content server web server to download files. To use web transfer, you must define these properties:

Property Name	Description
contentServerAdminID	The content server administrator ID to use to authenticate against the content server.
contentServerAdminPassword	The content server administrator password to use to authenticate against the content server. This password is encrypted.

A sample SCS configuration element using web transfer is shown below:

```
<adapter type="scs" default="true" name="myadapter">
  <config>
    <property name="port">4444</property>
    <property name="host">localhost</property>
    <property name="type">web</property>
    <property name="contentServerAdminID">sysadmin</property>
    <property name="contentServerAdminPassword">idc</property>
  </config>
</adapter>
```

3 Understanding the SCS API

The UCPM API is modeled into a set of services APIs, which are API calls that communicate with the content server.

This chapter contains the following sections:

- "Accessing the SCS API" on page 3-1
- "Understanding the SCS API Objects" on page 3-2
- "Understanding the SCS API Servlets" on page 3-8
- "Using the SCS APIs" on page 3-12

Note: Refer to *Oracle Fusion Middleware Content Integration Suite (CIS) Java API Reference* for information on the Class/Interface, Field, and Method descriptions.

3.1 Accessing the SCS API

The UCPM API is available on the CISApplication class via the getUCPMAPI () method. The getUCPMAPI () method returns a reference to the IUCPMAPI object, allowing access to all UCPM API objects. The public interface IUCPMAPI is the locator for the SCS, SIS, and CIS API objects. The SCS API is available via getActiveAPI (), which returns a reference to the SCSActiveAPI object.

The fully qualified method name is:

```
CISApplication.getUCPMAPI ().getActiveAPI ()
```

The SCS API comprises the following:

- **ISCSSearchAPI**: This is the command API implementation of the search commands.
- **ISCSFileAPI**: Deals with the retrieval of files, and the dynamic conversions of files, from the content server.
- **ISCSWorkflowAPI**: Deals with the workflow commands such as approval and rejection, viewing a user's workflow queue, and interacting with the content server workflow engine.
- SCS Document APIs (**ISCSDocumentCheckinAPI** and **ISCSDocumentCheckoutAPI**), which deal with active content in the content server, including checking in and out of content, content information, and deletion of content.

- Various APIs for the implementation of the administrative commands, component commands, and so on.

The interface ICommandFacade is the entry point into the command interface. It allows for interaction with the command layer, including command retrieval, registration, and execution. Commands are referenced by name, where a name can be any string. A name consisting of the dot character (".") will be treated in a hierarchy, where the first segment is the top-level category, and the next segment is the second-level category, and so on. Commands can either be retrieved by their full command name or by browsing all available commands.

The fully qualified class name is

```
com.stellent.command.ICommandFacade
```

Example using ISCSDocumentCheckinCommandAPI:

```
ISCSDocumentCheckinCommandAPI commandAPI =
  (ISCSDocumentCheckinCommandAPI)m_commandFacade.
    getCommandAPI ("document.checkin");
```

3.2 Understanding the SCS API Objects

The SCS API is responsible for formulating requests to the content server. SCS API calls translate into one or more IDC Service (Content Server Service) calls.

This section contains the following topics:

- "Interface ISCSObject" on page 3-2
- "Interface ICISTransferStream" on page 3-3
- "Interface ISCSServerBinder" on page 3-4
- "Interface ISCSServerResponse" on page 3-7
- "Interface ISCSRequestModifier" on page 3-7

3.2.1 Interface ISCSObject

The ISCSObject is the base interface for all objects in the SCS API. It inherits from ICISObject and adds some specific functionality relative to the content server objects. It allows access to the ISCSServerResponse object that created the object, and it also allows access to a collection of properties that have been modified since the object was initialized via the getModifiedProperties () method.

The ICISObject class name is new for UCPM 8.0.0 API.

ISCSObject objects have the concept of native property names. Specifically, properties that are available on ISCSObject are available by two different names: the Java property name and the content server native name. For example, to get the title of a ISCSContent item, the following three methods are equal:

```
String title = content.getTitle ();
title = content.getProperty ("title").getValue ().getStringValue ();
title = content.getProperty ("dDocTitle").getValue ().getStringValue ();
```

The content server supports a metadata model that can be extended. ISCSObject objects can have more properties than those exposed via the getProperty () methods. The ISCSObject implementations expose the most common properties, but other properties, such as the extended metadata, are only available via the getProperty ()

method. Also, the getProperties () method will list all the properties on the object, including properties without a corresponding **getter** or **setter** method.

```
for (Iterator it = content.getProperties ().iterator (); it.hasNext (); ) {
  ISCSProperty property = (ISCSProperty)it.next ();
  ISCSPropertyDescriptor descriptor = property.getActiveDescriptor ();
  if (descriptor.isBeanProperty ()) {
    System.out.println ("Property is available via get or set: " +
                         property.getDescriptor().getName ());
  } else {
    System.out.println ("Property is a hidden or extended property: " +
                         property.getDescriptor().getName ());
  }
    System.out.println ("Native property name: " + descriptor.getNativeName ());
}
```

The properties returned from ISCSObject implement the ISCSProperty interface, which adds the getActiveDescriptor () method. The ISCSPropertyDescriptor adds the `beanProperty` and `nativeName` properties to the available properties on an item.

The beanProperty property determines if the current property object has an available **getter** or **setter** method; if the property is false, this property object is available only via the getProperty () method.

The nativeName property returns the content server property name for the given property.

Date Objects

Date fields in the SCS API are handled as Java Date objects in Coordinated Universal Time (UTC time). All dates passed into the various properties of ISCSObject must be in UTC time. All date objects returned from the SCS API are in UTC time and need to be converted into the appropriate time zone for the particular application.

```
Date releaseDate = content.getReleaseDate ();

// convert from UTC time to Pacific Time Zone
  Calendar calendar = Calendar.getInstance ();
  calendar.setTime (releaseDate);
  calendar.setTimeZone (TimeZone.getTimeZone ("America/Los_Angeles"));
// use calendar to display date...
```

3.2.2 Interface ICISTransferStream

File streams to and from the content server have changed. In an effort to keep with the interface-only approach to the CIS APIs, all streams are sent via the ICISTransferStream interface. The interface represents the actual physical stream object and additionally some metadata, including filename, content length and mime-type. Since the ICISTransferStream is an interface, it cannot extend InputStream directly. Therefore, when using this object, you must first obtain the stream via a call to ICISTransferStream.getStream() and then manipulate the stream appropriately.

Some commands that in previous versions would return an InputStream now return ICISTransferStream objects. For example, if calling getFile() in the FileAPI, to access the stream from the content server, your code would look like:

```
ICISTransferStream transferStream = fileAPI.getFile (context, documentID);
InputStream inputStream = transferStream.getInputStream();
```

The implementation of ICISTransferStream contains all the necessary plumbing to transfer the stream to and from the command client to command server. Since

InputStream objects are not directly serializable, it does some extra work to put streams into places where the server can access them. All of the logic is hidden from the user of the API.

The stream instance can be obtained from the root IUCPMAPI interface using the method createTransferStream. That returns an empty instance of the stream container, which you can then use the accessors methods to set the stream properties. For example, below we create a transfer stream and point it at a local file:

```
// create the stream implementation
ICISTransferStream transferStream = ucpmAPI.createTransferStream ();

// point it at a file
transferStream.setFile (new File ("mytestdoc.doc"));
```

If you had a stream in memory already rather than a file handle, and you wanted to check in the content in that stream into the content server, you would need to specify all of the attributes for the stream such as a filename, content type, and the length of the stream.

```
ICISTransferStream transferStream = ucpmAPI.createTransferStream ();
  transferStream.setFileName ("sample.txt");
  transferStream.setInputStream (inputStream);
  transferStream.setContentType ("text/plain");
  transferStream.setContentLength (length);
  checkinAPI.checkinFileStream (context, content, transferStream);
```

3.2.3 Interface ISCSServerBinder

The CIS 8.0.0 API provides a new object, ISCSServerBinder, which is the root object used for all communication to and from the content server. The ISCSServerBinder object encapsulates a message both to and from the content server. It is a collection of properties, result sets, files, option lists and other specific type of information needed by the content server.

All API calls into the content server will create an instance of the ISCSServerBinder. Each API call, for example ISCSSearchAPI.search(), will use the supplied ISCSObject parameters to populate a binder and possibly add in other particular information. Likewise, all responses from the content server that are not streams are ISCSServerResponse objects which extend ISCSServerBinder.

As all ISCSObjects are collections of arbitrary metadata, the ISCSServerBinder is a collection of a number of objects metadata contained within one object. A particular ISCSServerBinder might contain the metadata for a content server query (see ISCSSearchQuery), metadata concerning a piece of content, and a list of objects dealing with user data. As each ISCSObject contains the particular metadata for its object, the ISCSServerBinder is responsible for the metadata of many objects.

Properties

As ISCSServerBinder extends the core ISCSObject interface, it has the ability to get and set arbitrary properties via the getProperty and setProperty methods. These arbitrary properties are usually where arguments for a particular content server are placed. They can be added directly, via the setProperty method as shown in the following code example.

```
// create an empty binder
ISCSServerBinder binder =
  (ISCSServerBinder)getUCPMAPI ().createObject (ISCSServerBinder.class);
```

```
// set some properties
  binder.setProperty ("dDocTitle", "test");
  binder.setProperty ("dSecurityGroup", "Public");
```

Alternatively, properties can be set using the mergeObject functionality available on the ISCSObject interface. The following example shows creating another object, setting some properties on that object, and then using merge to put those properties into the server binder.

```
// create an empty content item
ISCSContent content = (ISCSContent)getUCPMAPI ().createObject (ISCSContent.class);

// set some properties
content.setTitle ("test");
content.setSecurityGroup ("Public");

// merge into binder; this copies all the properties from content into the binder
binder.mergeObject (content);
```

The above two examples are identical: they both result in setting the content item title (dDocTitle) and security group (dSecurityGroup) in the ISCSServerBinder object. However, by using the second method, we abstract ourselves from having to deal with the specifics of content server naming. The ISCSContent object handles the mapping of standard Java properties into content server metadata.

Result Sets

A result set represents a collection of rows returned from a content server query. This is exposed in ISCSServerBinder via the getResultSet and setResultSet methods. A result set in the SCS API is then exposed as a homogeneous list, containing a type of object the represents a single row of the result set. Many items returned from content server queries come back as result sets. As all the result sets are lists of ISCSObject objects, the items from the result sets can be used in other calls. For example, look at the following code snippet where a search is executed and the first item then has its contents updated:

```
// create an simple query
ISCSSearchQuery query =
  (ISCSSearchQuery)getUCPMAPI ().createObject (ISCSSearchQuery.class):
  query.setQueryText ("dDocName <substring> 'test' ");

// execute a search
ISCSSearchResponse response =
  getUCPMAPI ().getActiveAPI ().getSearchAPI ().search (context, query);

// search results come back as a result set of ISCSSearchResult items
ISCSSearchResult result = (ISCSSearchResult)response.getResults ().get (0);

// change the title and check it in
result.setTitle ("new title");
getUCPMAPI ().getActiveAPI ().getDocumentUpdateAPI ().
  updateInfo (context, result);
```

File Objects

The ISCSServerBinder allows files to be sent to the content server via the addFile method. This method takes an ICISTransferStream. The resulting file is sent to the content server, along with the binder, during the request. Adding a file is similar to adding a property:

```
// create an empty stream
ICISTransferStream stream = getUCPMAPI ().createTransferStream ();

// point the stream at a local file
stream.setFile (new File ("testfile.txt"));

// add the stream to the binder
serverBinder.addStream ("myFile", stream);
```

When the above binder is sent to the content server, the stream will be transferred along with the binder. Inside the content server, the stream will be available under the "myFile" key which was specified when adding the stream to the binder.

Object Copying and Casting

Each ISCSObject in the SCS API has an object that holds the data in a low-level format compatible with Oracle Content Server. This object, referred to as a data object, is used by all ISCSObject implementations. This implies that any ISCSObject can be mutated to any other type of ISCSObject. There are two methods that expose this functionality: the castObject and copyObject methods available on the ISCSObject interface.

Both methods take in a single parameter: a class type representing the type of object that should be created. A call to **castObject** will result in the creation of a new object that points at the same backing data of the object it was invoked against. This implies that changes to the original object will be reflected in the object returned from the castObject call as well. A call to **copyObject** will result in a copy of the backing data being made, allowing the newly created object to act independently from the original object.

For example, imagine there is a custom content server service called "MY_DOC_INFO" which is similar to the standard "DOC_INFO" but it does some extra business logic processing. However, the returned binder from the "MY_DOC_INFO" call is very close to the "DOC_INFO" call. Since there is no explicit API call in the SCS API to call this "MY_DOC_INFO" service, we have to use the generic executeIDCService call. But we can use the castObject method to change the return type into something more user friendly:

```
// build our custom call
ISCSServerBinder binder =
  (ISCSServerBinder)getUCPMAPI ().createObject (ISCSServerBinder.class);
  binder.setService ("MY_DOC_INFO");

// create a document ID and add it to the binder
ISCSDocumentID documentID =
  (ISCSDocumentID)getUCPMAPI ().createObject (ISCSDocumentID.class);
  documentID.setDocumentID ("12345");
  binder.mergeObject (documentID);

// execute the call
ISCSServerResponse response =
  (ISCSServerResponse)getUCPMAPI ().getAdministrativeAPI ().
    executeIDCService (context, binder);

// use the cast to change it to a ISCSDocumentInformationResponse
ISCSDocumentInformationResponse infoResponse =
  (ISCSDocumentInformationResponse)response.
    castObject (ISCSDocumentInformationResponse.class);

// use the info response as usual
System.out.println ("Title: " + infoResponse.getDocNode ().getTitle ());
```

As mentioned above, the castObject call links the two objects by sharing the same backing data. In the above example, any changes made to the response object would be reflected in the infoResponse object as well:

```
// set a property on the response
response.setProperty ("customProperty", "customValue");

// value is then available in the infoResponse object
String value = infoResponse.getPropertyAsString ("customProperty");
```

If copyObject was called instead, the response and infoResponse would be independent of each other. The castObject creates a smaller memory footprint than copyObject, since the result from a castObject call does not create a new backing data object.

3.2.4 Interface ISCSServerResponse

The result of a call to the SCS API is usually an ISCSServerResponse object. The ISCSServerReponse is the base interface for all the response objects. It encapsulates the response from the content server for the last request. Most methods have specific implementations of this interface, which provide properties that are specific to those responses. See the JavaDocs for the specific response objects, and the properties available to each.

In the current release, some calls that previously returned references to an InputStream now return a ICISTransferStream object instead. See Interface ISCSRequestModifier below on how to use the new transfer stream interface.

3.2.5 Interface ISCSRequestModifier

All requests to the content server via the SCS API result in the creation of an ISCSServerBinder object. In certain situations, it becomes necessary to change the binder contents for a given API call to match the requirements of a specific content server. The ISCSRequestModifier interface is designed for this very purpose: to augment a call to the content server with custom modifications.

All APIs in the SCS API have a corresponding method that takes as the first parameter an ISCSRequestModifier object. Look at the API for ISCSSearchAPI:

```
/**
   * Command the implements searching against the content server.
  * @param SCSContext the context object representing the current user
  * @param searchQuery the content server query object
  */

public com.stellent.cis.client.api.scs.search.
  ISCSSearchResponse search (com.stellent.cis.client.api.scs.context.
    ISCSContext SCSContext,
    com.stellent.cis.client.api.scs.search.ISCSSearchQuery searchQuery)
  throws com.stellent.cis.client.command.CommandException;

/**
  * Command the implements searching against the content server.
  * @param requestModifier modify the request
  * @param SCSContext the context object representing the current user
  * @param searchQuery the content server query object
  * @see com.stellent.cis.server.api.scs.commands.search.SearchCommand
  */
```

```
public com.stellent.cis.client.api.scs.search.
  ISCSSearchResponse search (com.stellent.cis.client.api.scs.
    ISCSRequestModifier requestModifier, com.stellent.cis.client.api.scs.context.
    ISCSContext SCSContext, com.stellent.cis.client.api.scs.search.I
    SCSSearchQuery searchQuery)
    throws com.stellent.cis.client.command.CommandException;
```

The second API takes all the parameters of the first, with the additional ISCSRequestModifier argument. The standard Search API, and all its logic, can be used and custom logic added. For example, imagine a custom component installed on the content server that will do some extra processing if it find the "myCustomProperty" property set during a search query. To do this with CIS, we can use the ISCSRequestModifier to change the binder as follows:

```
// build a search query
ISCSSearchQuery query =
  (ISCSSearchQuery)getUCPMAPI ().createObject (ISCSSearchQuery.class);
  query.setQueryText ("dDocName <substring> `test`");

// build a request modifier
ISCSRequestModifier modifier =
  (ISCSRequestModifier)getUCPMAPI ().createObject (ISCSRequestModifier.class);

// access the binder off the modifier and add in our custom data
modifer.getServerBinder().setProperty ("myCustomProperty", "customValue");

// execute the search as normal
getUCPMAPI ().getActiveAPI ().getSearchAPI ().search (modifier, context, query);
```

Now the binder we modified will be used during the search call. Our custom property value will get sent along with the standard search call. This same method can be used to set any properties, add files, set result sets or even override which service call is being made on the content server.

3.3 Understanding the SCS API Servlets

The SCS API requires that a number of servlets be available to the system while operating in a J2EE/Web environment and running in server mode.

This section contains the following topics:

- "Servlet Descriptions" on page 3-8
- "SCS Servlet Parameters" on page 3-9
- "Servlet Security" on page 3-10
- "Servlets and API Interaction" on page 3-11

3.3.1 Servlet Descriptions

This table lists the servlet names and the appropriate configuration information needed in the web.xml file for a given web application:

Fully Qualified Name	Mapping	Description
SCSFileDownloadServlet com.stellent.web.servlets.SCSFileDownloadServlet	/getfile	Allows clients to retrieve files from the content server.

Fully Qualified Name	Mapping	Description
SCSCommandClientServlet com.stellent.web.servlets.SCSCommandClientServlet	/scscommandclient	Publishes the CIS server configuration information for CIS clients.
SCSFileTransferServlet com.stellent.web.servlets.SCSFileTransferServlet	/scsfiletransfer	Allows UCPM APIs to transfer files to a CIS client.
SCSInitialize com.stellent.web.servlets.SCSInitialize	N/A	Initializes the CIS Application instance. Should be set as a LoadOnStartup servlet.
SCSDynamicConverterServlet com.stellent.web.servlets.SCSDynamicConverterServlet	/getdynamicconversion/*	Executes a dynamic conversion and streams the result to the client.
SCSDynamicURLServlet com.stellent.web.servlets.SCSDynamicURLServlet	/scsdynamic/*	Retrieves dynamic files from the content server; used when rewriting the dynamically converted document URLs.

3.3.2 SCS Servlet Parameters

This section provides a description of the parameters for these servlets:

- "SCSFileDownloadServlet" on page 3-9
- "SCSDynamicConverterServlet" on page 3-10
- "SCSDynamicURLServlet" on page 3-10

This section does not specify any of the available security parameters, which are detailed in "Servlet Security" on page 3-10. All calls made to the content server use the identity as specified in the servlet security section.

3.3.2.1 SCSFileDownloadServlet

Property	Required	Description
adapterName	true	The adapter name to query for the document.
dDocName	n/a	The content ID of the document to retrieve.
rendition	false	The content rendition; valid only when specifying the dDocName.
revisionSelection	false	The revisionSelection to use when selecting content; valid only when specifying the dDocName.
forceStream	false	If true, the contents are streamed from the content server via the GET_FILE call regardless of optimized file transfer settings for the adapter; defaults to false.

3.3.2.2 SCSDynamicConverterServlet

Property	Required	Description
adapterName	true	The adapter name to query for the document.
contentID	Either contentID or documentID is required.	The content ID (dDocName) of the document to retrieve.
documentID	n/a	The document ID (dID) of the document to retrieve.
rendition	false	The rendition of the document to retrieve; valid only if contentID is specified.
revisionSelectionMethod	false	The revisionSelectionMethod to use to select the document; valid only if contentID is specified.
viewFormat	false	The view format of the conversion (that is, Native or WebViewable).
useAlternate	false	If true, use the alternate file for conversion; default is false.

3.3.2.3 SCSDynamicURLServlet

Property	Required	Description
adapterName	true	The adapter name to query for the document; not passed in as a parameter, but, rather, specified as the last segment on the URL: /cis-server/scsdynamic/<adaptername>?...
fileUrl	true	The relative path to the content server file to retrieve.

3.3.3 Servlet Security

All servlets, except for SISFileDownloadServlet and SCSInitializeServlet, make UCPM API calls and therefore must have a user context. By default, they will use the HttpServletRequest.getUserPrincipal() method to determine the user ID and pass that ID via the ISCSContext object to the UCPM API call. This behavior can be overridden by specifying a couple of initialization parameters to the servlet:

- **principalLookupAllowed**: If set to TRUE, the servlet will look for a user ID in the configured scope. The default scope is `session`.
- **principalLookupScope**: The scope of the lookup. Valid if principalLookupAllowed is TRUE. The defined scope will be used to call the getAttribute () method to discover the name of the current user; can be either `request`, `session`, or `application`. The default is `session`.
- **principalLookupName**: The name of the scoped parameter that holds the user ID. Valid if principalLookupAllowed is TRUE. The default is `principal`.
- **getUserPrincipalEnabled**: If set to FALSE, no call will be made to the HttpServletRequest.getUserPrincipal () method to determine the user ID. The default is TRUE.

- **principal**: The default user ID if no user ID can be determined. The default is `guest`.

To determine the current user ID, the servlets will first check the status of the principalLookupAllowed flag. If TRUE, it looks up the name of the user by determining the scope as set by the parameter principalLookupScope. With the current scope, the getAttribute () method is called, using principalLookupName as the parameter. If it is unable to locate a principal, it then checks the status of the getUserPrincipalEnabled flag. If that flag is TRUE, it calls the HttpServletRequest.getUserPrincipal () method. If that returns null, it uses the default principal to execute the request.

Without any changes to the servlet, the default behavior is to check the HttpServletRequest.getUserPrincipal () method and then use the default, if necessary. The other checks on the request, session, and application are done only if specified in the **init-param** of the servlet definition in the web.xml file.

3.3.4 Servlets and API Interaction

The ISCSFileAPI.getDynamicConversion() method performs a dynamic conversion of the given document (assuming the Dynamic Converter component is installed on the content server). The getDynamicConversion() call will also rewrite the returned URLs, so that they point back to the CIS servlets (as opposed to pointing directly to the content server) and display properly in the Web/Portal environment when they are rendered.

The rewritten URLs point back to SCSDynamicURLServlet, which then retrieves the item from the content server, via the SCS API, and streams it back to the client. The servlet determines the user ID for the context by the method described in "Servlet Security" on page 3-10.

Since the servlet determines the user ID, the user who executed the getDynamicConversion() call might not have the same user ID as the user clicking a link on the rendered HTML. This would be the case if the HttpServletRequest.getUserPrincipal() user ID does not match the ISCSContext user ID

In that event, the SCSDynamicURLServlet can be directed to look for a user parameter on the session by customizing it via the methods described under "Servlet Security" on page 3-10. Alternatively, SCSDynamicURLServlet can call the getDynamicConversion() and pass in an ISCSConvertedUrlInfo object that allows a user to optionally add parameters to the URL, which can then be used by your application to identify the context.

For example, if your application stored the current User ID in a session attribute named `stellentPrincipal`, you would modify the web.xml for the SCSDynamicURLServlet (and other servlets, as necessary) as follows:

```
<servlet>
  <servlet-name>scsdynamic</servlet-name>
  <servlet-class>com.stellent.web.servlets.SCSDynamicURLServlet</servlet-class>
  <init-param>
    <param-name>sessionPrincipalAllowed</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>sessionPrincipalName</param-name>
    <param-value>stellentPrincipal</param-value>
  </init-param>
</servlet>
```

3.4 Using the SCS APIs

The SCS Search, SCS File, SCS Document, and SCS Workflow APIs are discussed and sample code provided. These APIs perform task such as searching, checking in and out of content, and workflow approval and rejection.

It is assumed that you have initialized a CISApplication instance (referred to as m_cisApplication) and created a context object (referred to as m_context). Additional samples can be found in the SDK/Samples/CodeSamples/ directory.

This section contains the following topics:

- "SCS Search API" on page 3-12
- "SCS File API" on page 3-12
- "SCS Document APIs" on page 3-13
- "SCS Workflow API" on page 3-14

3.4.1 SCS Search API

The ISCSSearchAPI is the command API implementation of the search commands. You can use ISCSSearchAPI to search the content server using the following code:

```
// get a handle to the SCS Search API
ISCSSearchAPI searchAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI ().getSearchAPI ();
ISCSSearchResponse searchResponse =
  searchAPI.search (m_context, "dDocTitle <substring> 'HR'",  25);

// iterate all results
for (Iterator it = searchResponse.getResults ().iterator (); it.hasNext (); ) {
  ISCSSearchResult searchResult = (ISCSSearchResult)it.next ();

// print out the title and author
System.out.println ("Found result: " + searchResult.getTitle () + " by " +
  searchResult.getAuthor ());
}
```

3.4.2 SCS File API

The ISCSFileAPI deals with the retrieval of files, and the dynamic conversions of files, from the content server. A file can be retrieved simply by passing in the ID for the content. Alternatively, different versions of the file can be retrieved by using the optional ISCSFileInfo object to obtain references to the Web and Alternate versions of the file.

```
// get the SCS File API
ISCSFileAPI fileAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI ().getFileAPI ();

ICISTransferStream transferStream =
  fileAPI.getFile (m_context, content.getDocumentID ());

InputStream stream = transferStream.getInputStream();
// do something with the stream...
```

You can also use the _createFileInfo() method to get an ISCSFileInfo object. This object has several properties, which allow one to further select which rendition of a file to retrieve. The following sample uses the fileinfo object to get the Web Viewable rendition of a file. A similar process can be used to get the Alternate rendition.

```
// get the web viewable version of the file
ISCSFileInfo fileInfo =
  (ISCSFileInfo) m_cisApplication.getUCPMAPI ().createObject(ISCSFileInfo.class);
  fileInfo.setRendition ("Web");

// get the file
ICISTransferStream transferStream =
  fileAPI.getFile (m_context, content.getDocumentID (), fileInfo);
  InputStream stream = transferStream.getInputStream();
// do something with the stream...
```

The SCS File API can be used to generate HTML renditions of the content via the Dynamic Converter component of the content server (you must have the Dynamic Converter component installed).

In a similar fashion to the getFile () calls, you can either call getDynamicConversion () with an ID to retrieve the HTML conversion, or you can use the ISCSFileInfo and ISCSConvertedFileInfo objects to pass information into the API to process conversion rules and apply explicit templates.

```
ICISTransferStream transferStream =
  fileAPI.getDynamicConversion (m_context, content.getDocumentID ());
// process the stream...
```

The following sample combines the above features in one method that dynamically converts the alternate rendition of a given content object by using a custom conversion template.

```
// create the converted file bean and set our properties
ISCSConvertedFileInfo convertedInfo = fileAPI.__createConvertedFileInfo ();
convertedInfo.setConversionLayout ("custom_layout");
convertedInfo.setRendition ("Alternate");

// execute the dynamic conversion
ICISTransferStream transferStream =
  fileAPI.getDynamicConversion (m_context, content.getDocumentID (),
    convertedInfo);
// do something with the stream...
```

3.4.3 SCS Document APIs

The SCS Document APIs deal with content in the content server, including the checking in and out of content, content information, and the deletion of content.

This section provides a description of these APIs:

- "ISCSDocumentCheckinAPI" on page 3-13
- "ISCSDocumentCheckoutAPI" on page 3-14

3.4.3.1 ISCSDocumentCheckinAPI

This API deals with the check-in of all content to the content server. For a simple check-in of a file from disk, the following code will work:

```
// get the checkin api
ISCSDocumentCheckinAPI checkinAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckinAPI ();

// create an empty content object with the specified content ID
ISCSContent content =
(ISCSContent) m_cisApplication.getUCPMAPI ().createObject(ISCSContent.class);
```

```
ISCSContentID contentID =
(ISCSContentID) m_cisApplication.getUCPMAPI ().createObject(ISCSContentID.class);
  contentID.setContentID("my_test_file");
  content.setContentID(contentID);
  content.setAuthor (m_context.getUser ());
  content.setTitle ("Custom Title");
  content.setSecurityGroup ("Public");
  content.setType ("ADACCT");

// get the file stream
File myFile = new File ("c:/test/testcheckin.txt");
ICISTransferStream transferStream =
  m_cisApplication.getUCPMAPI ().createTransferStream();
  transferStream.setFile(myFile);

// execute the checkin
checkinAPI.checkinFileStream (m_context, content, transferStream);
```

In many deployments of the content server, there are required extended properties that need to be set for a new piece of content. These properties can be set on the content object via the setProperty() call available to all ICISObject objects. For example, some custom properties can be set as follows:

```
// set an extended property
  content.setProperty ("xCustomProperty", "Custom Value");
```

You can use the setProperty() method to set all the properties as opposed to calling the **setter** methods. You can use either the JavaBean name (for example, *title*) or the native content server property name that the JavaBean property corresponds to (that is, *dDocTitle*). In the next sample, we will set the title property in three ways, all equivalent:

```
// set via a standard property setter
  content.setTitle ("My Title");

// set a standard property using the JavaBean property name
  content.setProperty ("title", "My Title");

// set a property using the native content server property name
  content.setProperty ("dDocTitle", "My Title");
```

3.4.3.2 ISCSDocumentCheckoutAPI

This API deals with checking out content from the content server. Content items are identified by their ID.

```
// get the checkout api
ISCSDocumentCheckoutAPI checkoutAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckoutAPI ();

// checkout the file
checkoutAPI.checkout (m_context, content.getDocumentID ());
```

3.4.4 SCS Workflow API

The ISCSWorkflowAPI deals with the workflow commands such as approval and rejection, viewing a user's workflow queue, and interacting with the content server workflow engine. The following sample code shows an example of querying the workflow engine for the workflows currently active in the system:

```
// get the workflow API
ISCSWorkflowAPI workflowAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI().getWorkflowAPI ();
ISCSWorkflowResponse workflowResponse =
  workflowAPI.getActiveWorkflows (m_context);

// iterate through the workflows
for (Iterator it = workflowResponse.getActiveWorkflows().iterator();
  it.hasNext (); ) {
    ISCSWorkflow workflow = (ISCSWorkflow)it.next ();
    String name = workflow.getName ();
    String status = workflow.getWorkflowStatus ();
    System.out.println ("SCS workflow: " + name + "; status = " + status);
}
```

The most common interaction with workflows is to reject them or approve them and advance them to the next step in the workflow. The following code illustrates how to get a user's personal workflow queue and approve all workflows pending:

```
// get the workflow API
ISCSWorkflowAPI workflowAPI =
  m_cisApplication.getUCPMAPI ().getActiveAPI().getWorkflowAPI ();

// get the workflow queue
ISCSWorkflowQueueResponse queueResponse =
  workflowAPI.getWorkflowQueueForUser (m_context);
for (Iterator it = queueResponse.getWorkflowInQueue().iterator(); it.hasNext();) {
ISCSWorkflowQueueItem queueItem =
  (ISCSWorkflowQueueItem)it.next();

// approve the workflow
workflowAPI.approveWorkflow(m_context, queueItem.getDocumentID ());
}
```

Index