VERSION 7.6.1

Stellent® Content Integration Suite

UCPM API Developer's Guide

STELLENT™

Copyright

Contents

CHAPTER 1

Introduction

The Universal Content and Process Management (UCPM) API offers access to the Stellent server products 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 ISCSObject objects, which are the returned value objects from the server.

This guide provides conceptual and other information about the UCPM API, describes command invocation and execution, and explains how commands can be extended.

In this section:

- Software development kit (SDK)
- CIS Administration Application Help system

Notes

- The information in this guide is subject to change as product technology evolves and as hardware and operating systems are created and modified.
- Due to the technical nature of browsers, databases, web servers, and operating systems, Stellent, Inc. cannot warrant compatibility with all versions and features of third-party products.

Software development kit (SDK)

Stellent Content Integration Suite includes the UCPM API Software Development Kit (SDK), which comes bundled with a number of samples and build scripts that allow you to quickly get up and running; calling and customizing the CIS APIs.

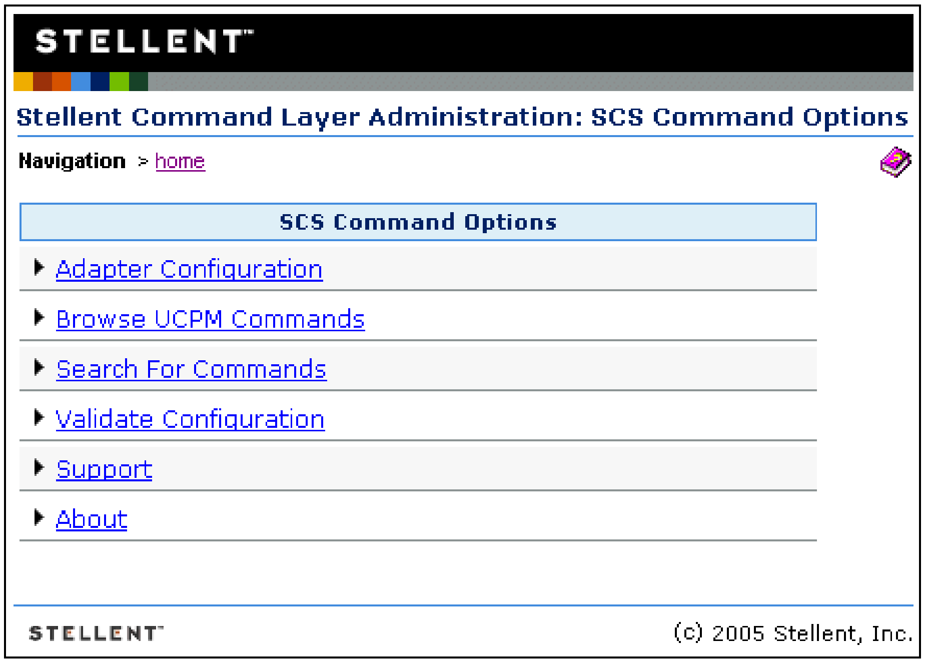
The following table lists the major components of the SDK, located in the <install_dir>/SDK/ directory inside the main distribution zip file.

Directory	Description
CommandBuilder	Ant scripts and support files for extending the CIS APIs.
library	The necessary libraries used by the SDK components.
Samples/Code Examples	Samples that show how to execute various pieces of the UCPM API.
Samples/ DirectoryCommands	The code used to demonstrate extending the CIS layer; referenced in the CommandBuilder SDK document.
Samples/ StellentWebSample	A sample web application that demonstrates the SDK working with the Active, Fixed, and Common APIs.
WebAppTemplate	A template for web applications built with CIS. Copy these files into your own WAR deployment; CIS libraries and initialization is already set up in this template.

CIS Administration Application Help system

You may need assistance with the CIS Administration Application, which is the administration interface for Stellent Content Integration Suite. Both online Help and the complete CIS Installation Guide can be accessed from any of the CIS Administration Application screens.

- Each dialog box in CIS Administration Application Help has a **Help** icon, which you can click to view a topic that describes the functionality of that dialog box.
- Once you have opened any Help topic, click the **Show Navigation** button in the navigation bar to view the complete CIS Installation Guide online.



Stellent Command Layer Administration: SCS Command Options

Navigation > home

SCS Command Options

- Adapter Configuration
- Browse UCPM Commands
- Search For Commands
- Validate Configuration
- Support
- About

(c) 2005 Stellent, Inc.

The online Help can be viewed in a web browser such as Internet Explorer or Netscape Navigator, with conventional web browser controls (Back, Forward, Refresh, and so on) used to navigate the Help system.

The online Help contains the following navigation options:

- **Contents**: The Contents tab contains an expandable list of Help topics. Click a book icon to expand or collapse that section of the Help system and then select the desired Help topic to view a topic.
- **Index**: The Index tab provides immediate access to individual Help topics by keyword.
- **Search**: The Search tab provides a full-text search of the Help system. Type the word or phrase you are looking for and then click **Go** (or press Enter on your keyboard).
- **Favorites**: The Favorites tab provides a means to save topics that you have found useful and anticipate viewing again. (When viewing a topic, click **Favorites** and then click **Add** to add the topic to your Favorites list.) The Favorites tab does not display if your browser does not support the Java applet used by the Help system.

The online Help also includes a toolbar with the following options

Toolbar	Definition
🔄	Highlights the title of the topic currently being viewed in the Contents tab. **Note**: If you open a Help topic by clicking the Help button in a dialog box, the Show Navigation button will display, instead.
<	Displays the topic that precedes the currently displayed topic. (The order of topics can be seen on the Contents tab.)
>	Displays the topic that follows the currently displayed topic. (The order of topics can be seen on the Contents tab.)
✉	Opens a new, blank email message addressed to Stellent Technical Support, with the name of the Help topic in the Subject line. Please use this feature to provide feedback to Stellent on the topic itself or how the topic is documented.
🖨	Prints the currently displayed topic by using your web browser print feature.
📄	Opens a PDF version of the CIS Installation Guide. You can read the PDF version online or print it out, if you prefer.
S	Connects to the Stellent web site (www.stellent.com), where you can learn more about Stellent products, upcoming events, and technical support.

Notes

- The Help systems use a secure Java applet to display the Contents, Index, Search, and Favorites tabs. If the Java applet is not installed or enabled in the web browser used to view Help, a JavaScript rendition of Help will display, instead. If both Java and JavaScript are disabled in the web browser, the browser will not be able to display Help.
- The Java applet is not supported by Internet Explorer on UNIX, Netscape 4.x on Macintosh, Netscape 6.x on any platform, or by any machine running a newer version of the Java Runtime Environment (JRE) 1.3.1_02. For these browsers and operating systems, the JavaScript rendition of Help will display (assuming it is not disabled in the browser).

CHAPTER 2

Understanding the UCPM API

The UCPM API offers access to Stellent servers 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 ISCSObject objects, which are the value objects returned from the server.

In this section:

- Accessing the UCPM API
- UCPM methodology
- CIS initialization
- Spring framework
- Integration environments
- Common method objects
- Interacting with the UCPM API
- Properties and the ISCSObject interface
- Adapter configuration file

Note: See the JavaDocs for information on the Class/Interface, Field, and Method descriptions. The complete CIS JavaDoc is located in the /docs/cis-javadoc-all.zip file (the /docs/cis-javadoc-ucpm.zip file contains only the UCPM API information).

Accessing the UCPM API

The UCPM API offers access to Stellent servers 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 ISCSObject objects, which are the value objects returned from the server.

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. Public interface IUCPMAPI is the locator for these API objects:

- **getActiveAPI ()** returns a reference to the SCSActiveAPI object. The "Active API" classes communicate with and handle content stored on Stellent Content Server.
- **getFixedAPI ()** returns a reference to the SCSFixedAPI object. The "Fixed API" classes communicate with and handle content stored on Stellent Image Server.
- **getCommonAPI ()** returns a reference to the SCSCommonAPI object. The "Common API" classes communicate with both Stellent Content Server and Stellent Image Server, by providing a common set of APIs that are shared between both the Active API and the Fixed API. These include searching, contribution, and information querying.

UCPM 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.

The UCPM API—including the Active, Fixed, and Common implementations—are all called in a similar fashion. The first parameter for all methods is an IContext bean:

- **ISCSContext** for the Active API. The interface ISCSContext is the context object used when communicating with Stellent Content Server.
- **ISISContext** for the Fixed API. The interface ISISContext is the context object used when communicating with Stellent Image Server.
- **ISCSCommonContext** for calling some of the Common APIs. The interface ISCSCommonContext identifies which adapters to query and what user information to use.

The IContext bean holds context information, such as username and session ID, that 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 ISCSObjects. 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 ();
ISCSActiveDocumentID documentID = activeAPI._createActiveDocumentID ("10");
ISCSActiveContent content = activeAPI.getDocumentInformationAPI ().
                            getDocumentInformation (m_context, documentID);

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

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

CIS initialization

Stellent Content Integration Suite (CIS) is initialized by accessing the CISApplication class, which resides in the com.stellent.cis package. The CISApplication class has a number of static initialization methods which allows you to initialize CIS in server or client mode:

Note: Also see "Properties and the ISCSObject interface" on page 22.

- In server mode, CIS can execute commands to the various Stellent servers on its own; it has no other dependencies.
- In client mode, CIS connects to another CIS server instance, and all CIS requests are sent to the CIS server to be processed. This allows for one CIS server to centralize the configuration of the various Stellent services that it will communicate with, and allows the server to share caching and other server-side resources between the CIS clients.

A CIS instance initialized in server mode can execute commands on its own; a CIS instance initialized in client mode must connect to another CIS server instance to execute commands.

CIS initialization should happen once per application. The CIS APIs are stateless and therefore the initialized CISApplication instance can be safely shared between threads. CIS can be run inside a web application, as a standalone EAR file, or in a standalone JVM process. The initialization is slightly different for each configuration.

Server mode

To initialize the CIS instance in server mode, the initializeServer () methods are used. The initializeServer () methods requires that the *adapterConfig* parameter be defined. The *adapterConfig* parameter is the path to the adapter configuration file (adapterconfig.xml).

Note: The adapter configuration file contains XML-formatted configuration information for communicating with the various content servers and image servers. See "Adapter configuration file" on page 24 for more information.

Optionally, you may specify the server type using the *serverType* parameter. If no server type is specified, the class will attempt to determine the server type automatically.

For example, to initialize in standalone mode (standalone JVM / no CIS clients):

```
CISApplication.initializeServer (CISApplication.TYPE_STANDALONE,
    new File ("adapterconfig.xml").toURL() );
```

To initialize for a WebLogic application server:

```
CISApplication.initializeServer (CISApplication.TYPE_WEBLOGIC,
    new File ("adapterconfig.xml").toURL() );
```

To initialize and allow CISApplication to determine the server type (if the server type cannot be determined, it defaults to standalone):

```
CISApplication.initializeServer (new File ("adapterconfig.xml").toURL() );
```

Client mode

To initialize the CIS instance in client mode, the intializeClient () methods are used. However, there must be a server mode running that accepts CIS clients and the connection method must then be determined by the type of environment.

For a connection to a CIS server running on a J2EE server, use the intializeClient () method that takes in a JNDI context object that is connected to the target server. This method requires that the *context* parameter be defined. The *context* parameter is the initialized context object to query the CIS Configuration EJB

Alternatively, you can use an initialized RMIRegistry instance to connect to a CIS server running on a Tomcat environment. This intializeClient () method requires that the *registry* parameter be defined. The *registry* parameter is the initialized registry where the CIS configuration instance is located

For clients running in the same J2EE server (or cluster) as the CIS instance, you can simply create the default InitialContext and pass that into the initializeClient () call to connect to the running CIS server:

```
InitialContext ctx = new InitialContext ();
CISApplication cisApplication = CISApplication.initializeClient (ctx);
```

To connect to a J2EE server running on a remote machine, you need to specify the appropriate arguments to the InitialContext to point it at the correct server. For example, the code below connects a CIS client to a WebLogic host running on a remote server:

Note: You must include the WebLogic libraries in order to connect to a WebLogic server if running in an external client outside the J2EE server.

```
Hashtable connectionArgs = new Hashtable ();
connectionArgs.put (Context.PROVIDER_URL, “t3://servername:7001”);
connectionArgs.put (Context.INITIAL_CONTEXT_FACTORY,
  “weblogic.jndi.WLInitialContextFactory”);
InitialContext ctx = new InitialContext (connectionArgs);
CISApplication application = CISApplication.initializeClient (ctx);
```

When running against a remote J2EE instance, as is the case of WebSphere, the JNDI names are sometimes different (as you are no longer operating in the java:comp/env context).

Programmatically, you can implement the IJNDILookupStrategy interface and handle the mapping of the JNDI names to the appropriate names for your special case. There is already an implementation of this for WebSphere, because connecting to a remote WebSphere instance requires a complete path in the cluster for each of the JNDI names. For example, the following code connects to a remote WebSphere instance using a WebSphere naming strategy:

```
Hashtable connectionArgs = new Hashtable ();
connectionArgs.put (Context.PROVIDER_URL,
    “corbaloc:iiop:servername:9080”);
connectionArgs.put (Context.INITIAL_CONTEXT_FACTORY,
    “com.sun.jndi.cosnaming.CNCtxFactory”);
InitialContext ctx = new InitialContext (ctx);
IJNDILookupStrategy lookupStrategy =
    new WebsphereRemoteLookupStrategy (“servername”, “server1”);
CISApplication application =
    CISApplication.initializeClient (ctx, lookupStrategy);
```

The client can also initialize without specifying any parameters. In this case, the CISApplication class looks in the JNDI tree for a configuration object placed by the CIS server instance. This is a useful mechanism for initializing multiple CIS clients that are in different web applications but deployed in the same application server as the CIS server.

```
CISApplication.initializeClient ();
```

Spring framework

CIS uses the Spring Framework, based upon the Inversion of Control (IoC) and Dependency Injection (DI) patterns, which allows developers to wire classes together in ways that reduces coupling. It does this by providing a central and automated way to build associations between classes thereby reducing unnecessary dependencies between those classes.

Extending CIS with the Spring Framework

CIS uses the Spring Framework to dynamically configure itself at runtime. At startup, CIS attempts to load the configuration file: /META-INF/resources/cis-client-services-custom.xml from the classpath. Creating this file, adding entries to it, and adding it to the classpath will cause the Spring Framework to load your classes and inject their dependencies. This file provides an additional point of integration with CIS.

To learn more details on the Spring Framework and Spring syntax see:

http://www.springframework.org/

scripting and processing engine

CIS uses an open-source tool for turning XML into executable code. This Java and XML based scripting and processing engine, called Jelly (an Apache/Jakarta project), allows developers to script their XML. For input, it reads in an XML document and can produce dynamic events which can be turned back into XML. If you are familiar with the Apache project Velocity, then the syntax will be familiar to you.

During initialization, CIS uses Jelly to preprocess XML configuration files. These files then get passed to the Spring Framework. The Spring Framework uses the XML to create the required Java beans and their associations.

To learn more about Jelly see:

http://jakarta.apache.org/commons/jelly

CIS Initialization strategy

When we talk of CIS initialization, we usually mean one of two ways: server and client. The server component of CIS receives command requests, executes those requests and returns the results to the client. The client component of CIS creates the command request and sends it to the server for execution. The server communicates directly with Stellent Content Server while the client communicates directly to the server. The cis-server application fills the role of "server" while the cis-admin application fills the role of "client."

Both the server and client initialize similarly with one important difference: the client gets its Spring injected configuration from the server application while the server application loads its configuration from the classpath.

CIS Server initialization

At startup, the servlet (SCSInitializeServlet) begins the CIS server initialization process. It attempts to load various properties from the web.xml file and classpath (i.e., cis-initialization.properties). It then passes those properties to the static method CISApplication.initialize(...).

The following table describes the order of operations:

Class / Method	Description
SCSIntitialize init(ServletConfig)	Called by the web application container during initialization of the web application. Load properties from web.xml Load properties from classpath: /cis-initialization.properties Call CISApplication.initialize(properties) [see below] Via the CISWebHelper class, it sets the CISApplication and the command application instance as an attribute on the servlet context
CISApplication initialize(properties)	Called by the init(…) method of an object instance of SCSInitialize. Calls CISApplication.initializeServer(properties)
CISApplication initializeServer(properties)	Called by CISApplication.initialize(properties) Creates the adapter config URL (used to eventually load the adapter config) Creates temporary directory Determines Server type (i.e. weblogic, websphere, etc) Determines whether server should support options like: (EJB, JMS, JCA, RMI) Calls initializeServer (serverType, adapterConfigUrl, temporaryDirectory)
CISApplication initializeServer (serverType, adapterConfigUrl, temporaryDirectory)	Called by CISApplication.initializeServer(properties) CISApplication.initializeServer(properties) Creates bean services URL (cis-services.jxml) ▪ /META-INF/resources/server/cis-services.jxml Calls initializeServer (serverType, beanServicesURL, adapterConfigURL, temporaryFileDirectory)
CISApplication initializeServer (serverType, beanServicesURL, adapterConfig, temporaryFileDirectory)	Called by CISApplication.initializeServer (serverType, adapterConfigUrl, temporaryDirectory) Creates a new CommandServer object Initializes the Command Server object with the bean services URL Create a new adapter config factory and initialize it ▪ Parse the adapter config (adapterconfig.xml) ▪ Create the adapters. ▪ For SCS adapters use: /META-INF/resources/adapter/adapter-services-scs.jxml Creates a new CISApplication obj passing in the commandServer and serverType. The CISApplication constructor is responsible for printing a message to the console that the server has been initialized.

Class / Method	Description
CommandServer new ( )	Created by CISApplication.initializeServer (serverType, beanServicesURL, adapterConfig, temporaryFileDirectory) Creates a JellyResourceLoader Loads the cis-resources resource bundle (for Internationalized server messages)
CommandServer initialize(beanSvcsURL)	Called by CISApplication.initializeServer (serverType, beanServicesURL, adapterConfig, temporaryFileDirectory) Creates the Jelly enabled XML file: cisinit.jxml Copies the bean services URL file to the cisinit.jxml Initializes itself using the cisinit.jxml file. This basically loads the bean definitions contained in the cisinit.jxml file. Have Spring Load Bean Definitions ▪ Sets the bean factory ▪ Sets the Jelly enabled bean definition reader. ▪ Loads the bean definitions ▪ Process the cisint.jxml file. Here Jelly gets used to evaluate the cis-services.jxml file (through the use of JellyResourceLoader, JellyResourceDecorator, JellyHelper). ▪ The bean services file (cis-services.jxml) causes the following files also to get evaluated: /META-INF/resources/server/cis-command-services.jxml /META-INF/resources/shared/cis-api-services.jxml /META-INF/resources/commandbeans-lwapi.xml /META-INF/resources/command-services-lwapi.xml /META-INF/resources/commandbeans-active.xml /META-INF/resources/command-services-active.xml /META-INF/resources/commandbeans-fixed.xml /META-INF/resources/command-services-fixed.xml /META-INF/resources/commandbeans-common.xml /META-INF/resources/command-services-common.xml /META-INF/resources/commandbeans-custom.xml Sets itself as the system bean factory (within the Spring Framework) Sets itself as a class variable

CIS Client Initialization

For the CIS administration application, the CIS client application gets initialized via a call from CommandAppInitializer (a Struts PlugIn). The CommandAppInitializer loads a properties file (/WEB-INF/cis-initialization.properties). It also holds the path to (/WEB-INF/cis-administrationapp-services.xml).

These files get passed to the static method CISApplication.initialize(properties) where it determines if this is a client or a server initialization.

The following table describes the order of operations:

Class / Method	Description
CommandAppInitializer init(ActionServlet, ModuleConfig)	Called by struts Loads the cis-initialization.properties resource bundle Calls CISApplication.initialize(properties) Sets the CISApplication as an instance of WebApplication Sets WebApplication as an attribute on the servlet context
CISApplication initialize(properties)	Called by SCSInitialize Calls CISApplication.initializeClient(properties)

Class / Method	Description
initializeClient(properties)	Called by CISApplication.initialize(properties) Using the properties, it determines how to connect to server (JNDI, RMI, URL, default) Unless specified in the properties, it calls initializeClient() Returns an instance of CISApplication
initializeClient ( )	Called by initializeClient(properties) Creates a new CommandClient Calls CommandClient.connect() Creates a new CISApplication passing in the command client object (CommandClient).
CommandClient new ( )	Created by CISApplication.initializeClient (properties) Creates a JellyResourceLoader Loads the cis-resources resource bundle (for Internationalized client messages)
CommandClient connect ( )	Loads /META-INF/resources/client/cis-client-services.jxml ▪ Creates the JellyResourceLoader Initializes the command client with the bean services URL (i.e. cis-client-services.jxml)
CommandClient connect(context, lookupStrategy)	Called by CommandClient connect() Gets configuration bean via a JNDI lookup: ▪ 'stellent/command/configuration' On the server side, the CommandConfigurationBean via the CommandConfigReader loads the 'cis-client-services.jxml' file: ▪ /META-INF/resources/client/cis-client-services.jxml Calls initialize(configString). The configString is the string representation of the XML configuration that was obtained from the server.

Class / Method	Description
CommandClient connect(configString)	Called by CommandClient connect(context,lookupStrategy) Creates the temporary client configuration file: cisinit.jxml Copies the configString to that file Calls initialize (resourceLocation, refresh)
CommandClient initialize(resourceLocation, refresh)	Called by CommandClient.connect(configString) Initializes itself using the cisinit.jxml file. This basically loads the bean definitions contained in the cisinit.jxml file. Have Spring Load Bean Definitions ▪ Sets the bean factory ▪ Sets the Jelly enabled bean definition reader. ▪ Loads the bean definitions ▪ Process the cisint.jxml file. Here Jelly gets used to evaluate the cis-services.jxml file (through the use of JellyResourceLoader, JellyResourceDecorator, JellyHelper). The bean services file (cis-services.jxml) causes the following files also to get evaluated: /META-INF/resources/client/cis-client-j2ee-services.xml /META-INF/resources/shared/cis-api-services.jxml /META-INF/resources/commandbeans-lwapi.xml /META-INF/resources/command-services-lwapi.xml /META-INF/resources/commandbeans-active.xml /META-INF/resources/command-services-active.xml /META-INF/resources/commandbeans-fixed.xml /META-INF/resources/command-services-fixed.xml /META-INF/resources/commandbeans-common.xml /META-INF/resources/command-services-common.xml /META-INF/resources/commandbeans-custom.xml /META-INF/resources/server/adapter-services-scs.jxml

Integration environments

CIS can be integrated into a web environment, an application server environment, or a standalone environment. Each method is described in detail below.

Web environment

A sample template for a web application is available in the /SDK/WebAppTemplate/ directory. This template is complete and ready for use with CIS 7.6. It contains the necessary entries in the web.xml file for initializing the CIS application. It also contains the CIS-related servlets and the necessary JAR files for CIS (located in the /WEB-INF/lib/ sub-directory). The JavaServer Pages Standard Tag Library (JSTL) tags are included, too.

The /WEB-INF/ sub-directory contains the adapterconfig.xml file, which is the adapter configuration file that is read at initialization and defines for the CIS layer the Stellent server(s) to communicate with. By default, there is one adapter, called *myadapter*, that points to a content server on localhost at port 4444.

Note: See "Adapter configuration file" on page 24 for more information.

The /WEB-INF/ sub-directory can immediately be bundled, as is, into a WAR file (using a zip tool such as WinZip) and then deployed into any Servlet 2.3–compatible servlet engine, such as Tomcat 4.x or 5.x.

When writing code in the template, you can use the already initialized CISApplication instance to obtain UCPM API objects for communicating with the Stellent servers.

For example, if we 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 --%>
<%
    CISApplication cisApplication = (CISApplication) request.getSession().
                               getServletContext().getAttribute ("CISApplication");
    SCSActiveSearchAPI searchAPI = cisApplication.getUCPMAPI ().
                               getActiveAPI ().getSearchAPI ();
    //create a context
    ISCSContext context = cisApplication.getUCPMAPI ().
                               getActiveAPI ()._createSCSContext ();
    context.setUser ("sysadmin");

    //execute the search
    ISCSActiveSearchResponse 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.

Application server environment

Once CIS is deployed in the target application server, you can communicate with the CISApplication by initializing the CISApplication as a "client." To do this, you need to:

1. Assemble the client application (EAR, WAR, etc.).
2. Place all the libraries from the "standalone" version into the client application.
3. In the application startup code, initialize the CISApplication instance, using the intializeClient() method:

```
CISApplication.intializeClient()
```

   This should be done only once; the CISApplication instance can be shared throughout the application

   Typically, you can use CISApplication.intializeClient() without any parameters if CIS is also deployed on the same application server.

4. Make the CISApplication instance available to the entire application; typically, in web applications this is done by using the ServletContext.setAttribute() method.

All client applications must have the correct version of the Stellent CIS libraries, which are available in the “standalone” directory of the CIS distribution. After CIS is initialized, calls can be made to the CIS layer, as in the previous example.

Standalone environment

CIS can be integrated into a “standalone” application. A standalone application is an application that runs outside the context of a J2EE application server or servlet engine (i.e., a command line or Swing application).

The first step is to take all the libraries from the standalone directory and place them in your application classpath. Then, copy the adapter configuration file (adapterconfig.xml) from the standalone directory into your application environment, where it can be accessed at runtime and you can modify it according to your Stellent server configuration.

Note: See “Adapter configuration file” on page 24 for more information.

The CISApplication instance then needs to be initialized. The initialized CISApplication instance is thread-safe and can be shared by the application. Initialization can be done with the following lines of code:

```
//load the adapterconfig.xml file
File adapterConfig = new File ("adapterconfig.xml");

//initialize the CISApplication in standalone mode
CISApplication cisApplication = CISApplication.initializeServer
                                (CISApplication.STANDALONE, adapterConfig.toURL());
```

Common method objects

The common method objects are IContext (generic context used for communication with the Command APIs) and ICommandExecutionProperties (used to set command execution flags and asynchronous flags).

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, etc.

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 two sub-interfaces: SCSContext and ISISContext (both ISCSContext and ISISContext extend the IContext interface).

- **ISCSContext** is the context object used for the Active APIs and represents a users operating context when communicating with the content server.
- **ISISContext** is the context object used for Fixed APIs and represents a users operating context when communicating with the image server.

The context objects can be created by using the _create methods on their corresponding APIs. Thus, ISCSContext can be created from the SCSActiveAPI, and ISISContext can be created from the SCSFixedAPI.

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

//create an ISISContext
ISISContext context =
        m_cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
```

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 Common API will take either an ISCSCommonContext or an IContext object. ISCSCommonContext 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
ISCSCommonContext context =
          m_cisApplication.getUCPMAPI ().getCommonAPI ()._createCommonContext ();
```

Once the ISCSCommonContext adapter is created, multiple adapters can be added to it. This is done using the ISCSCommonContext.addContext() method. Any number of adapters can be added; all the adapters added to the ISCSCommonContext 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.

Note: We have provided an sample web application (located in /SDK/Samples/StellentWebSample) that has a login method that first validates the username against the content server and, if successful, adds the IContext object to the HttpSession object. See the LoginActionHandler class in the src/com/stellent/sdk/web/sampleapp/handlers/login directory for more details.

The IContext object is also populated with a globally unique session ID when it is created. The session ID can be useful for identifying one session from another and stays the same for the duration of the context object. The session ID can be set by the user if the target application wishes to supply its own session ID. The only requirement is that the identifying session ID string be unique.

ICommandExecutionProperties

The interface ICommandExecutionProperties handles properties that can affect the execution of a given command via a Command API method. All available APIs have two versions: one takes an ICommandExecutionProperties object as its first parameter, the other does not. The ICommandExecutionProperties are optional and further define the execution path the command will take once it is passed into the command layer. The ICommandExecutionProperties object can be used to set command execution flags and asynchronous flags.

Command execution flags

The ICommandExecutionProperties object can modify the behavior of an API by calling setValidateCommand (). If the isValidateCommand () method returns false, the Command Bean's validation method will not be called. The default value for the flag is true.

Available methods allow the setting of custom, API-specific properties on the ICommandExecutionProperties object. These properties are interpreted different by the underlying APIs.

- In the Active API, any available properties via the method ICommandExecutionProperties.getProperties() are sent to the content server upon execution. This allows the user to specify extra parameters that the command might not include.
- In the Fixed API, the ICommandExecutionProperties.getProperties() are ignored.

Asynchronous flags

The available APIs can execute asynchronously (execute in another process and allow for the method to return immediately). The ICommandExecutionProperties object allows someone to make an asynchronous call by calling the setAsynchronous () method on the ICommandExecutionProperties object.

```
//perform a checkin asynchronously
ICommandExecutionProperties properties =
          m_cisApplication.getCommandFacade ().createExecutionProperties ();
properties.setAsynchronous (true);

//get the checkin API
ISCSActiveDocumentCheckinAPI checkinAPI =
           m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckinAPI ();

//perform the checkin (assume other parameters were created)
checkinAPI.checkinFileStream (properties, context, content, contentID, stream);

//since this is asynchronous, the return value is null...
```

The asynchronous invocation is available only when CIS is deployed on an application server that supports the Java Messaging Service (JMS). If the ICommandExecutionProperties object is passed to a CIS instance that does not support JMS (that is, if it is deployed on a standalone JVM), then setAsynchronous () is ignored and it will execute synchronously. Only through an explicit call to a UCPM API method that takes an ICommandExecutionProperties object will an asynchronous invocation take place.

After the asynchronous command is executed, the command result object is broadcasted on a JMS topic with the JNDI name stellent/jms/CommandResultTopic. Listening to this topic will allow you to retrieve the results of the asychronous operation. The result are sent as a javax.jms.ObjectMessage class, with the internal object being a ICommandResult object. This allows you access to both the command that executed the query and the object that resulted from that execution. if an error occurred during execution, calling the ICommandResult.getException ()

method will return the error that occurred during processing of the command. If there was an error, then the ICommandResult.getResult () method will return null.

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 interface IUCPMAPI is the locator for the various API objects in the UCPM API. The IUCPMAPI object has methods to get references to the Active, Fixed, and Common APIs. This allows you to access the necessary APIs and begin making calls through the UCPM API to the target server.

- **getActiveAPI ()** returns a reference to the SCSActiveAPI object (for communicating with the content server).
- **getFixedAPI ()** returns a reference to the SCSFixedAPI object (for communicating with the image server).
- **getCommonAPI ()** returns a reference to the SCSCommonAPI object (for calling some of the Common APIs).

Calling API objects using newly instantiated ISCSObject objects

Many UCPM API calls take in an ISCSObject or an interface that inherits from ISCSObject. For example, in the ISCSActiveDocumentInformationAPI (Active API) the getDocumentInformation () method takes as a parameter a ISCSActiveDocumentID object.

Note: The fully qualified method name is "CISApplication.getUCPMAPI ().getActiveAPI ().getDocumentInformationAPI ()".

In such cases, to obtain a reference to a valid object to pass in as a parameter, you either can retrieve the object reference from another ISCSObject or create a new instance of the ISCSObject using the one of the appropriate _create methods available on the APIs. The _create methods allow you to create a new ISCSObject and populate it.

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

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

//create the document ID
ISCSActiveDocumentID documentID =
                    m_cisApplication.getUCPMAPI ().getActiveAPI ().
_                   createActiveDocumentID ("12345");
ISCSDocumentInformationResponse response =
                    documentInfoAPI.getDocumentInformation(context, documentID);
```

For any API that requires an ISCSObject, there should be a _create method available that allows you to create a new instance of the API object. These _create methods are all client-side methods; they do not make calls to the target server. They are to be treated as "constructors" for the ISCSObject implementations.

Properties and the ISCSObject interface

The interface ISCSObject is the base interface for all objects with metadata in the UCPM API. Thus, all UCPM API objects are inherited from ISCSObject. The interface ISCSObject allows for the retrieval and setting of the object properties. The objects returned from calls to the UCPM API are value objects—that is, 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.

Property accessors

Most implementations of ISCSObject have their own specific property “accessor” methods. However, all properties can be retrieved by calling the getProperty () method on the ISCSObject. The ISCSProperty.getProperty () method will return an ISCSProperty 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 and retrieve the content object
ISCSActiveContent content = response.getContent ();

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

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

The ISCSObject 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");
```

Property object types

A property object type is determine by the return value of the property method on the ISCSObject. 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 (e.g., boolean, float, long, etc.).

Note: 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.

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

If we take a look at the ISCSActiveContent 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, we must use the boolean encapsulation. 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 (e.g., "not a boolean") will result in an exception.

Property collections

The available list of properties can be retrieved using the getProperties () method on the ISCSObject interface. This will return all of the available properties for a given object. This allows you to build generic views of an object regardless of the API from which it originated (Active, Fixed or Common).

```
//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 ();
    ISCSPropertyValue value = property.getValue ();
    if (value != null) {
        System.out.println (name + " = " + value.getStringValue ());
    }
}
```

In the above example, the value object was checked for a null reference. If the property does not have a value, the ISCSPropertyValue will be null. Another way to deal with this situation is to call ISCSObject.getProperties() with a filter. For example, we can ask for only those properties that have a value (i.e. are non-null):

```
//retrieve only the properties with values
Collection properties = content.getProperties (ISCSObject.FILTER_NULL_VALUE);
```

Conversely, we can ask for the properties that do not have a value, as in the next example:

```
//now retrieve only the properties without a value
Collection nullProperties =
            content.getProperties (ISCSObject.FILTER_NULL_VALUE, true);
```

The property filters can also be combined using the OR operator into a bitmask that contains all the filters to use. For example, if we want to filter out the read-only and null properties, we can use the following code:

```
Collection customProperties  =
           content.getProperties
           (ISCSObject.FILTER_NULL_VALUE | ISCSObject.FILTER_READ_ONLY);
```

And, as in the previous examples, we can ask for the converse and return only those properties that were null and read-only, by using the other version of the getProperties () method that takes in a boolean operation as in the following example:

```
Collection customProperties  =
             content.getProperties
             (ISCSObject.FILTER_NULL_VALUE | ISCSObject.FILTER_READ_ONLY, true);
```

Adapter configuration file

The adapter configuration file (adapterconfig.xml) contains XML-formatted configuration information for communicating with your content server and image server instances. 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.

If your CIS instance is deployed in a Web environment in server mode, you can use the CIS Administration Application to configure an adapter:

1. Browse to the CIS Administration Application (http://localhost:port/scscis).
2. Click **Adapter Configuration**.
3. Click **Create new SCS Adapter** or **Create new SIS Adapter** and enter the appropriate values. These will vary, depending on your installation.
4. If you need assistance, click **Help**.

Note: See “Secure Socket Layer (SSL) Communication” in the CIS Installation guide for information on enabling Secure Socket Layer (SSL) communication

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	The type of adapter; “scs” represents a connection to a content server; “sis” represents a connection to an image server; if not specified, the default is “scs”.
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.
resourceName	Required only for adapters of type “scs”. The name of the server resource used by the adapter. If deployed in a non-J2EE environment, this is the name of a resource pool that handles the server requests. Otherwise, this is the JNDI name of the JCA adapter that handles the server requests. EDIT--- the name of the resource used to talk to the content server. On J2EE servers, this is the JCA JNDI name; on non-J2EE servers, this is the name of the resource pool used to handle the connections.

A sample adapter element is shown below:

```
<adapter type="scs" default="true" name="myadapter" resourceName="pool1">
```

Config element

Within the config element is a set of property elements that define the adapter-specific properties. The SCS or “active” adapter uses different configuration elements that the SIS or “fixed” adapter. The configuration elements for both are explained below.

SCS adapter configuration elements

An SCS or “active” adapter communicates with Stellent Content Server (SCS). The config element for a SCS adapter has four general attributes as shown in the following table:

Property name	Description
post	The port of the content server (usually 4444).
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 tranfer files; requires a content server username and password for Basic HTTP Authentication (file download only).

A sample SCS configuration element is shown below:

```
<adapter type="scs" default="true" name="myadapter" resourceName="pool1">
    <config>
        <property name="port">4444</property>
        <property name="host">localhost</property>
        <property name="type">socket</property>
    </config>
</adapter>
```

Note: The attributes “port”, “host”, and “type” are associated with a content server adapter. An image server adapter would require the “soapUrl”, “uploadUrl”, and “downloadUrl” attributes. See “SIS adapter configuration elements” on page 26 for additional information.

By default, tthe content server socket communication layer is used to stream files to and from the content server. However, for high-volume checkin 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’s 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.

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

```
<adapter type="scs" default="true" name="myadapter" resourceName="pool1">
    <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>
```

SIS adapter configuration elements

An SIS or “fixed” adapter communicates with Stellent Image Server (SIS). The config element for an SIS adapter has five attributes as shown in the following table:

Property name	Description
soapUrl	The image server SOAP URL is the URL to the WSDL that CIS uses to communicate with the image server.
uploadUrl	The image server upload URL is the URL to the image server that accepts requests to post data streams. Used by CIS to contribute file streams from the user.
downloadUrl	The image server download URL is the URL to the image server that accepts requests for data streams. Used by CIS to return file streams to the user.
licenseView	The lcense view is the type of view into the system the user will have can be "Web View" or "Full Seat".
style	Used to determine the style of licensing. Currently, this value can either be left blank or be "CheckTrust", which indicates whether a trusted relationship should be used.

To create an SIS adapter, you must have Stellent Image Server version 7.5 or later installed and the Web SDK running and accessible from the machine where CIS is installed.

CIS communicates with the image server through the use of Web services. Consequently, the protocol used is the Simple Object Access Protocol (SOAP), and the application-level protocol definition is the Web Services Description Language (WSDL).

A sample SIS configuration element is shown below:

```
<adapter type="sis" name="fixedAdapter">
    <config>
        <property name="soapUrl">
        http://localhost/acordeweb/services/Acorde2.asmx
        </property>

        <property name="uploadUrl">
        http://localhost/AcordeWeb/UTInfrastructure/WebServiceFileReceiver.aspx
        </property>

        <property name="downloadUrl">
        http://localhost/AcordeData/AcordeSessionData
        </property>

        <property name="licenseView"></property>
        <property name="style"></property>

    </config>
</adapter>
```

Sample adapter configuration file

The following is a full example of an adapterconfig.xml file that can be used to communicate with a content server and an image server, both running on localhost using the default ports:

```
<?xml version="1.0" ?>
<config>
```

```
<adapter type="scs" default="true" name="myadapter" resourceName="pool1">
  <config>
    <property name="port">4444</property>
    <property name="host">localhost</property>
    <property name="type">socket</property>
  </config>
<beans template="classpath:/META-INF/resources/adapter/adapter-services-scs.jxml"/>
</adapter>

<adapter name="fixedAdapter">
  <config>
    <property name="soapUrl">
     http://localhost/acordeweb/services/Acorde2.asmx
    </property>
    <property name="uploadUrl">
     http://localhost/AcordeWeb/UTInfrastructure/WebServiceFileReceiver.aspx
    </property>
    <property name="downloadUrl">
     http://localhost/AcordeData/AcordeSessionData
    </property>
    <property name="licenseView"></property>
    <property name="style"></property>
  </config>
</adapter>
</config>
```

CHAPTER 3

Understanding the Active API

The UCPM API is modeled into a set of Services APIs, which are API calls that communicate with the target server, and into ISCSObject objects, which are the returned value objects from the server.

The Active API is a branch of the UCPM API that deals exclusively with Stellent Content Server. The term ”active” refers to the nature of the content residing within the content server which tends to be active in nature.

In this section:

- Acccessing the Active API
- Understanding the Active API objects
- Understanding the Active API servlets
- Using the Active APIs

Important: The current version of CIS uses the UCPM API which is not compatable with CIS 7.2. See “Migrating from Version 7.2” on page 73 for more information.

Note: See the JavaDocs for information on the Class/Interface, Field, and Method descriptions. The complete CIS JavaDoc is located in the /docs/cis-javadoc-all.zip file (the /docs/cis-javadoc-ucpm.zip file contains only the UCPM API information).

Acccessing the Active 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. Public interface IUCPMAPI is the locator for the Active, Fixed, and Common API objects. The Active API is avalable via getActiveAPI (), which returns a reference to the SCSActiveAPI object.

Note: The fully qualified method name is “CISApplication.getUCPMAPI ().getActiveAPI ()”.

The Active API comprises:

- **Active Search API** — the ISCSActiveSearchAPI is the command API implementation of the search commands.
- **Active File API** — the ISCSActiveFileAPI deals with the retrieval of files, and the dynamic conversions of files, from the content server.
- **Active Document APIs** — the Active Document APIs deals with active content in the content server, including the checking in and out of content, content information, and the deletion of content.
- **Active Workflow API** — the ISCSActiveWorkflowAPI deals with the workflow commands such as approval and rejection, viewing a user’s workflow queue, and interacting with the content server workflow engine.
- Various APIs for the implementation of the administrative commands, component commands, etc.

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, etc. Commands can either be retrieved by their full command name or by browsing all available commands.

Note: The fully qualified class name is "com.stellent.command.ICommandFacade".

Example using ISCSDocumentCheckinCommandAPI:

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

Understanding the Active API objects

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

Interface ISCSActiveObject

The ISCSActiveObject is the base interface for all objects in the Active API. It inherits from ISCSObject 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.

ISCSActiveObject objects have the concept of native property names. Specifically, properties that are available on ISCSActiveObject are available by two different names: the Java property name and the content server native name. For example, to get the title of a ISCSActiveContent 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. ISCSActiveObject objects can have more properties than those exposed via the getProperty () methods. The ISCSActiveObject 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 (); ) {
    ISCSActiveProperty property = (ISCSActiveProperty)it.next ();
    ISCSActivePropertyDescriptor descriptor = property.getActiveDescriptor ();
    if (descriptor.isBeanProperty ()) {
        System.out.println ("Property is available via get or set: " +
                             property.getName ());
    } else {
        System.out.println ("Property is a hidden or extended property: " +
                             property.getName ());
    }
    System.out.println ("Native property name: " + descriptor.getNativeName ());
}
```

The ISCSActiveObject property objects implement the ISCSActiveProperty interface, which adds the getActivePropertyDescriptor () method. The ISCSActivePropertyDescriptor adds the properties “beanProperty” and “nativeName” 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 Active API are handled as Java Date objects in Coordinated Universal Time (UTC time). All dates passed into the various properties of ISCSActiveObject must be in UTC time. All date objects returned from the Active 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...
```

Interface ISCSServerResponse

The result of any call to the Active API is usually an ISCSServerResponse object (or an InputStream for some specific document retrieval methods). 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.

Understanding the Active API servlets

The Active API requires that a number of servlets be available to the system while operating in a J2EE/Web environment and running in server mode. The following table lists servlet names and the appropriate configuration information needed in the web.xml file for a given web application.

SCSInitializeServlet, SCSFileDownloadServlet, and SCSDynamicConverterServlet servlets. SCSDynamicConverterServlet

Fully qualified name	Mapping	Description
SCSFileDownloadServlet com.stellent.web.servlets.SCSFileDownloadServlet	/getfile	Allows clients to retrieve files from the content server.
SCSCommandClientServlet com.stellent.web.servlets.SCSCommandClientServlet	/scscommandclient	Publishes the CIS server configuration information for CIS clients.
SCSFileTransferServlet com.stellent.web.servlets.SCSFileTransferServlet	/scsfiletransfer	Allows CIS 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.

Servlet parameters

This section provides a description of the parameters for the SCSInitializeServlet, SCSFileDownloadServlet, SCSDynamicConverterServlet, and SCSDynamicURLServlet servlets.

This section does not specify any of the available security parameters, which are detailed in "Servlet security" on page 35. All calls made to the content server use the identity as specified in the servlet security section.

SCSInitializeServlet

The SCSInitializeServlet is a convient 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 us to move the configuration of the how CIS initializes outside the scope of the EAR/WAR file.

Client property definitions

Propery	Description
cis.config.type	Must be set to "client" to initialize in client mode. Default is server.
cis.config.client.type	Can be either JNDI, RMI, or not specified to allow for the default behavior.
jndi.*	All properties beginning with jndi. will be treated as JNDI properties; the 'jndi.' prefix will be removed and the remaining property name/value will be passed to the InitialContext.
cis.config.client.jndi.lookup-strategy	The class name of a IJNDILookupStrategy; this class will be used to intercept all client JNDI lookups; this gives the caller a chance to modify the JNDI name for each query.
cis.config.client.jndi.lookup-strategy.args	A comma-separated list of arguments that will be passed, as Strings, to the constructor of the lookup strategy. This will allow for customization of the lookup strategy via the configuration file. Useful to pass information about the JNDI namespace as in the case of Websphere remote JNDI trees.
cis.config.client.rmi	The Remote Metod Invokation (RMI) URL to the host CIS instance; in the form rmi://hostname:port.

Note: Refer to "Initialization properties" in the CIS Installation guide (cis-installation-guide.pdf) for client configuration examples.

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 directoy of the unbundled stellent-cis-7.6.1.zip file (this is the directory containing the adapterconfig.xml file).

Property	Description
cis.config.type	Default is server (server mode). ▪ Set to "server" to initialize in server-mode ▪ Set to "client" to initialize in client mode.
cis.config.server.adapterconfigl	The URL pointing to the adapterconfig.xml file. In addition to standard URLs, this can also be in classpath form (i.e. classpath:/)
cis.config.server.type=sap	The type of server to initialize as; if not specified, the server attempts to evaluate the server type. Valid values are weblogic, websphere, tomcat, standalone, sap, sunone or custom.
cis.config.server.type.options	A comma separated list of the options for the custom server type. The options are either on or off, dependening on their presence in this property. Options are: ▪ ejb (for enabling ejb clients), ▪ rmi (for enabling rmi clients), ▪ jms (for enabling use of JMS servers) ▪ jca (for enabling use of the JCA adapter),
cis.config.server.type.options.ejb=true	Default is true (EJBs enabled).

Property	Description
cis.config.server.type.options.rmi=true	Default is true (MI enabled).
cis.config.server.type.options.jms=true	Default is true (JMS enabled).
cis.config.server.type.options.jca=false	Defaut is false (JCA is disabled).

Note: Refer to "Initialization properties" in the CIS Installation guide (cis-installation-guide.pdf) for server configuration examples.

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.

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 documentID (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 viewFormat of the conversion (i.e. Native or WebViewable).
useAlternate	false	If true, use the alternate file for conversion; default is false.

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: /scscis/scsdynamic/<adaptername>?...
fileUrl	true	The relative path to the content server file to retrieve.

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. 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". Default is "session".
- **principalLookupName** — the name of the scoped parameter that holds the user ID. Valid if principalLookupAllowed is TRUE. Default is "principal".
- **getUserPrincipalEnabled** — if set to FALSE, no call will be made tothe HttpServletRequest.getUserPrincipal () method to determine the user ID. Default is TRUE.
- **principal** — the default user ID if no user ID can be determined. 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.

Servlets and API interaction

The ISCSActiveFileAPI.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 Active 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 35.

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 35. 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 Stellent 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>
```

Using the Active APIs

The Active Search, Active File, Active Document, and Active 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).

Note: Additional samples can be found in the SDK/Samples/CodeSamples/ directory.

Active Search API

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

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

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

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

Active File API

The ISCSActiveFileAPI 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 Active File API
ISCSActiveFileAPI fileAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getFileAPI ();
SCSContentFileStream stream =
     fileAPI.getFile (m_context, content.getActiveDocumentID ());

//save the file to disk via a utility method
StreamUtil.copyStream (stream, new FileOutputStream (stream.getFileName ());

//close the stream
stream.close ();
```

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 = m_cisApplication._createFileInfo ();
fileInfo.setRendition ("Web");

//get the file
SCSContentFileStream stream =
     fileAPI.getFile (m_context, content.getActiveDocumentID (), fileInfo);
//do something with the stream...
```

The Active 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.

```
SCSContentFileStream stream =
     fileAPI.getDynamicConversion (m_context, content.getActiveDocumentID ());
//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._createConversionInfo ();
```

```
convertedInfo.setConversionLayout ("custom_layout");
convertedInfo.setRendition ("Alternate");

//execute the dynamic conversion
SCSContentStream contentStream =
     fileAPI.getDynamicConversion
     (m_context, content.getActiveDocumentID (), convertedInfo);
//do something with the stream...
```

Active Document APIs

The Active Document APIs deals with active content in the content server, including the checking in and out of content, content information, and the deletion of content. The ISCSActiveDocumentCheckinAPI and the ISCSActiveDocumentCheckoutAPI are discucussed in detail below.

ISCSActiveDocumentCheckinAPI

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
ISCSActiveDocumentCheckinAPI checkinAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckinAPI ();

//create an empty content object with the specified content ID
ISCSActiveContent content = checkinAPI._createContent ("my_test_file");
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");
SCSContentFileStream stream = new SCSContentFileStream (myFile);

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

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 ISCSObjects. 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");
```

ISCSActiveDocumentCheckoutAPI

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

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

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

Active Workflow API

The ISCSActiveWorkflowAPI 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
ISCSActiveWorkflowAPI workflowAPI =
     m_cisApplication.getUCPMAPI ().getWorkflowAPI ();
ISCSWorkflowActiveResponse workflowResponse =
     workflowAPI.getActiveWorkflows (m_context);

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

CHAPTER 4

Understanding the Fixed API

The UCPM API is modeled into a set of Services APIs, which are API calls that communicate with the target server, and into ISCSObject objects, which are the returned value objects from the server.

The Fixed API is a branch of the UCPM API that deals exclusively with Stellent Image Server. The term ”fixed” refers to the nature of the content residing within the image server which tends to be high volume but static in nature.

CIS communicates with Stellent Image Server through the use of Web services. Consequently, the protocol used is the Simple Object Access Protocol (SOAP), and the application-level protocol definition is Web Services Description Language (WSDL). See “CIS architecture” in the CIS Installation Guide for more information (cis-install-guide.pdf).

In this section:

- Accessing the Fixed API
- Understanding the Fixed API objects
- Using the Fixed APIs

Important: The current version of CIS uses the UCPM API which is not compatable with CIS 7.2. See “Migrating from Version 7.2” on page 73 for more information.

Note: See the JavaDocs for information on the Class/Interface, Field, and Method descriptions. The complete CIS JavaDoc is located in the /docs/cis-javadoc-all.zip file (the /docs/cis-javadoc-ucpm.zip file contains only the UCPM API information).

Accessing the Fixed 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. Public interface IUCPMAPI is the locator for the Active, Fixed, and Common API objects. The Fixed API is avalable via getFixedAPI (), which returns a reference to the SCSFixedAPI object.

Note: The fully qualified method name is “CISApplication.getUCPMAPI ().getFixedAPI ()”.

The Fixed API comprises:

- **ISCSFixedAdministrativeAPI** — allows you to query/create adapters programmatically.
- **ISCSFixedUserAPI** — used to log in to and out of an image server.
- **ISCSFixedSearchAPI** — implementation of the search commands used to perform searches against the image server.
- **ISCSFixedDocumentAPI** — used to contribute documents to, and retrieve documents from, the image server.
- **ISCSFixedProcessAPI** — used to query/effect information about current state of processes within the image server.

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, etc. Commands can either be retrieved by their full command name or by browsing all available commands.

Example using ISCSFixedSearchAPI:

```
ISCSFixedSearchAPI commandAPI =
     (ISCSFixedSearchAPI)m_commandFacade.getCommandAPI
     ("search", com.stellent.command.ICommandFacade.TYPE_UCM_FIXED_API);
```

Understanding the Fixed API objects

Once the CISApplication object has been initialized, you can retrieve the Fixed API objects:

```
CISApplication cisApplication = CISApplication.initializeServer (myHost, myPort);
cisApplication.getUCPMAPI ().getFixedAPI ()
```

Each object returned from a Fixed API call is a descendent of ISCSFixedObject. This object provides the ability to return the source SOAP object, allows the user to determine if an error occurred while building the object, and allows the user to retrieve the objects version.

Using the Fixed APIs

The Fixed User, Fixed Search, Fixed Document, and Fixed Process APIs are discussed and sample code provided. These APIs perform the task of logging in, searching, document export and contribution, and handling business processes.

Note: Additional samples can be found in the SDK/Samples/CodeSamples/ directory.

Fixed User API

The ISCSFixedUserAPI is used to log in to and out of an image server. If there is a trusted relationship between CIS and the image server, a login is not necessary. The trusted IP address allows CIS to log in behind the scenes when it uses the Fixed API, and only a valid user name on the context is necessary.

However, if this relationship does not exist, or is not desired, explicit logins are possible. Each login method will return a string value called ‘AIID’, which is an XML snippet that is used as a login token for the remainder of the application. Once you have successfully logged in, set this value on the context via the setAcordeID () method so that subsequent calls on the Fixed API will be authenticated.

In each of the following examples, the CISApplication object is initialized and ISCSFixedUserAPI is defined as userAPI:

```
ISCSFixedUserAPI userAPI =
    cisApplication.getUCPMAPI ().getFixedAPI ().getUserAPI ();
```

Basic login

This login requires only a username, password, and an adapter name that specifies the image server you wish to log into:

```
String aiid = userAPI.login ("sisadapter", "sysadmin","idc");

//Create a new context and populate it with the appropriate information.
ISISContext ctx = cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
ctx.setUser ("sysadmin");
ctx.setAcordeID (aiid);
ctx.setAdapterName ("sisadapter");

//This context is now authenticated and can be used in subsequent calls
//to the image server.
```

Log in to Stellent Process

If Stellent Process (electronic process monitoring tool) is installed on the image server, you can log in to both at the same time by specifying the database name:

```
String aiid =
  userAPI.login ("sisadapter", "sysadmin","idc", "ibpmprocessdemo", "FullSeat", "");

//Create a new context and populate it with the appropriate information.
ISISContext ctx = cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
ctx.setUser ("sysadmin");
ctx.setAcordeID (aiid);
ctx.setAdapterName ("sisadapter");

//This context is now authenticated against the image server and Stellent Process.
```

Logging out

You can invalidate a user token (i.e., 'AIID') by logging out explicitly. This may be useful to free up a license:

```
//Context here assumed to be authenticated
userAPI.logout (ctx);

//User token now invalidated and license released.
```

Fixed Search API

The ISCSFixedSearchAPI is the implementation of the search commands used to perform searches against the image server. Searching with the Fixed API can be accomplished in three different ways:

- Executing a saved search that has been defined on the server.

- Executing an adhoc search that lets the user specify the search parameters programmatically.
- Performing a lookup of a specific document by specifying two parameters that uniquely identify an image server document.

Each method of searching will return a ISCSFixedSearchResponse object. This object contains search status information, a list of column descriptors that describe the various user defined metadata fields present in the current set of search results, and a list of ISCSFixedContent objects that were found by the search.

Each content object provides an identifier, which uniquely identifies the piece of content through several system-level data fields, and a list of ISCSUserField objects, which represent the user-defined metadata for the content object.

Each type of search (excluding the lookup) allows you to specify the maximum number of results to return from the search. You may continue the search if there were more available by calling the continueSearch () method and passing in the searchCookie () value of the fixed search response.

Note: Stellent Image Server currently only supports moving forward through a result set.

In each of the following examples, the CISApplication object has been initialized and the ISCSFixedSearchAPI has been defined as searchAPI:

```
ISCSFixedSearchAPI searchAPI =
     cisApplication.getUCPMAPI ().getFixedAPI ().getSearchAPI ();
```

This example assumes that a context object has been created that refers to a SIS adapter:

```
ISISContext ctx =
     cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
ctx.setUser ("myUserID");
ctx.setAdapterName ("sisadapter");
```

Saved searches

The tasks of getting saved search names, getting saved search prompts, and continuing a search are discussed and a complete saved search example is provided.

Getting saved search names

If you do not know what saved searches are available, you can query the server for a list of these names:

```
// Returns a list of strings.
List names = searchAPI.getSavedSearchNames (ctx);

//Iterate through and print the names to the console.
Iterator iter = names.iterator ();
String searchName;
while (iter.hasNext ()) {
       searchName = (String)iter.next ();
       System.out.println (searchName);
}
```

Getting saved search prompts

Once you know the search name that you wish to execute, you can query the server for the list of user prompts available for that saved search:

```
//Retrieve the list of prompts for the search 'HR Search by Employee Last Name'
List prompts =
     searchAPI.getSavedSearchPrompts (ctx, "HR Search by Employee Last Name");

Iterator iter = prompts.iterator ();
ISCSFixedSearchPrompt prompt;
while (iter.hasNext ()) {
       prompt = (ISCSFixedSearchPrompt)iter.next ();
       //Print out Display text and operator value
       System.out.println (prompt.getDisplayText ());

       //This returns an operator string, the getOperator ()
       //returns an integer representation.
       System.out.println (prompt.getOperatorStr ());

       //If the prompt is the Employee Last Name, set the user input to 'Farley',
       //this will only return results that have the last name of 'Farley'.
       if (prompt.getDisplayText ().equals ("Employee Last Name")) {
           prompt.setUserInput ("Farley");
       }
}
```

Continuing a search

If you have specified a search that does not return all the results in one batch, you may continue the search to retrieve the remaining results:

```
//Get the initial five results.
ISCSFixedSearchResponse response =
     searchAPI.search (ctx, ("HR Search by Employee Last Name", 5);

//Use the search connection ID present in the first response to continue the search.
String connectionID = response.getSearchCookie ();

while (response.getStatus () != ISCSFixedSearchResponse.COMPLETE) {
       response =  searchAPI.continueSearch (ctx, connectionID);
       //Process next 5 results
}
```

A complete saved search example

Once you know the saved search name and the prompt values you wish to use, you can execute a complete saved search (this example assumes that the prompts List has been setup as above):

```
//Perform search with prompts set to last name of "Farley",
//returning a maximum of 20 results
ISCSFixedSearchResponse response =
     searchAPI.search (ctx, "HR Search by Employee Last Name", prompts, 20);

//Iterate through and print results.
List results = response.getResults ();
Iterator resultIter = results.iterator ();
Iterator fieldIter;
```

```
ISCSFixedContent content;
List userFields;
ISCSFixedUserField field;
while (resultIter.hasNext ()) {
       content = (ISCSFixedContent) resultIter.next ();
       userFields = content.getUserFields ();
       fieldIter = userFields.iterator ();

       while (fieldIter.hasNext ()) {
              field = (ISCSFixedUserField)fieldIter.next ();
              System.out.println (field.getName ());
              System.out.println (field.getValue ());
       }
}
```

Adhoc searches

The tasks of getting searchable applications and getting a specific application query is discussed and a complete adhoc query example is provided.

Getting searchable applications

To create a search programmatically, you first need the application name you wish to build the search for. The application is a document definition residing in the image server that defines the metadata present in each piece of content. You can query the server for searchable application types using this code:

```
//Get the response object.
ISCSFixedApplicationResponse resp = searchAPI.getSearchableApplications (ctx);

//Filter indicates what filter was used to get the application list.
System.out.println ("Filter: " + resp.getFilter ());

//Iterate through the list of returned applications and echo application name.
Iterator iter = resp.getApplications ().iterator ();
ISCSFixedApplication application;

while (iter.hasNext ()) {
       application = (ISCSFixedApplication)application;
       System.out.println (application.getName ());
}
```

Getting a specific application query

To create a programmatic query, you must first get the list of prompts and operators available for the application type you wish to query. This query has a list of ISCSFixedAdhocPrompts and a list of available logical operators (AND, OR, etc). Each adhoc prompt also has a list of comparison operators that can be used (EQUAL, LIKE, etc).

```
//You can retrieve the query by its string name
//or by passing in the associated ISCSFixedApplication object.
ISCSFixedAdhocQuery query =
     searchAPI.getApplicationQuery (ctx, "HR Application");

//Print out available logical operators.
Iterator logopIter = query.getLogicalOperators ().iterator ();
ISCSFixedLogicalOperator logop;
```

```
while (logopIter.hasNext ()) {
logop = (ISCSFixedApplication)logopIter.next ();
System.out.println (logop.getName ());
}

//Print out available prompts for this query.
Iterator promptIter = query.getPrompts ().iterator ();
ISCSFixedAdhocPrompt prompt;
while (iter.hasNext ()) {
       prompt = (ISCSFixedAdhocPrompt)iter.hasNext ();
       System.out.println (prompt.getName ());

//You can specify user input to this prompt with setUserInput ()
//and get a list of available comparison operators with getComparisonOperators ()
}
```

A complete adhoc query example

Once you know the application name and the prompt values you wish to use, you can execute a complete adhoc query:

```
ISCSFixedAdhocQuery query = searchAPI.getApplicationQuery (ctx, "HR Application");

//Get the Employee Name prompt and set the Employee Name to John.
ISCSFixedAdhocPrompt prompt = query.getPrompt ("EmployeeName");
prompt.setUserInput ("John");
prompt.setComparisonOperator (ISCSFixedAdhocPrompt.OPERATOR_BEGINSWITH);

//Get the DocumentType prompt and set to Dental Form,
//and set the comparison operator.
prompt = query.getPrompt ("DocumentType");
prompt.setUserInput ("Dental Form");
prompt.setComparisonOperator (ISCSFixedAdhocPrompt.OPERATOR_EQUAL);

//Set the query to AND the prompts together.
query.setLogicalOperator (ISCSFixedAdhocQuery.OPERATOR_AND);

//Perform the search of HR documents that have an Employee name of John
//and a document type of Dental Form. (The zero indicates that all results
//should be returned.)
ISCSFixedSearchResponse response = searchAPI.adhocSearch (ctx, query, 0);

//Process results.
```

Looking up a specific document

If you know the application name and row identifier of a specific document, you can perform a search designed to return a specific document. Also, if you have an identifier object retrieved from an ISCSFixedContent object, you also may look up a specific document with a single call.

```
//Fixed ID object assumed to be retrieved prior to this snippet.
//This call will either return the unique document specified by the identifier
//or will throw an exception if the document is not found.
ISCSFixedContent contentItem =
     searchAPI.lookup (ctx, identifier);

//String applicationName and String rowIdentifier assumed to be retrieved
//prior to this snippet.
```

```
ISCSFixedContent contentItem =
     searchAPI.lookup (ctx, applicationName, rowIdentifier);
```

Fixed Document API

The ISCSFixedDocumentAPI is used to contribute documents to, and retrieve documents from, the image server. The tasks of document export and document contribution are discussed and examples provided.

In the following examples, the CISApplication object has been initialized and the ISCSFixedDocumentAPI has been defined as documentAPI:

```
ISCSFixedDocumentAPI documentAPI =
     cisApplication.getUCPMAPI ().getFixedAPI ().getDocumentAPI ();
```

This example also assumes that a context object has been created that refers to a SIS adapter:

```
ISISContext ctx =
     cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
ctx.setUser ("myUserID");
ctx.setAdapterName ("sisadapter");
```

Document export

Each ISCSFixedContent object has a list of ISCSFixedContentPage objects associated with it. These pages can be exported in the following formats: GIF, BMP, TIF, JPG (default), PCX, OPTIKA, NATIVE, EMF, PDF, and SMARTMODE. There are constants available off the interface ISCSFixedContentPage. Once you have retrieved the list of content pages of the desired type, you may then make calls to the server to retrieve the stream for each page.

Default set of content pages

This example illustrates getting the default set of content pages associated with a piece of fixed content and saving the streams to disk:

```
//Identifier assumed to be prior to this snippet, it is retrieved
//via the getIdentifier () method on a valid piece of ISCSFixedContent.
List contentPages = documentAPI.getContentPages (ctx, identifier);

ISCSFixedContentPage contentPage;
Iterator iter = contentPages.iterator ();
FileOutputStream out;
InputStream in;
while (iter.hasNext ()) {
       contentPage = (ISCSFixedContentPage)iter.next ();
       //Relative location of content stream as seen by Image Server.
       System.out.println ("Location: " + contentPage.getLocation ());
       System.out.println ("PageNumber: " + contentPage.getPageNumber ());
       System.out.println ("SectionNumber: " + contentPage.getSectionNumber ());

       //Get the input stream associated with this content page.
       in = documentAPI.getContentStream (ctx, contentPage);
       //Create out
       out = new FileOutputStream (contentPage ().getPageNumber () + "_" +
       contentPage.getSectionNumber () + ".jpg");

       //Write the stream to disk
       byte[] bytes = new byte[64];
       int read;
       while ((read = in.read (bytes)) != -1) {
       out.write (bytes, 0, read);
       }

       in.close ();
       out.close ();
}
```

Native set of content pages

You can also get the native set of content pages. These represent the actual files that were contributed to the image server.

```
//Many defaults are used here, check JavaDocs for the meaning of each.
//The important thing to note is the type, which has been specified as Native.
List pages = m_fixedDocument.getContentPages (ctx, identifer, ISCSFixedContentPage.
DEFAULT_DPI, ISCSFixedContentPage.TYPE_NATIVE,
       ISCSFixedContentPage.DEFAULT_NAMESPACE,
       ISCSFixedContentPage.DEFAULT_SCALE,
       ISCSFixedContentPage.DEFAULT_HIGHEST_RESOLUTION);

//Process pages as desired.
```

Document contribution

You may contribute new documents to the image server through the Fixed API. This is accomplished by first uploading the desired file stream to the image server, retrieving a ticket, and then associating this ticket with a set of metadata that describes the document that is to be contributed.

Uploading a file to the image server

To upload a file to the image server, you need to create an SCSContentFileStream object and call the upload method.

```
File fileToUpload = new File ("c:\\TestDocument.tif");

//Create a new SCSContentStream
SCSContentFileStream fileStream =
     new SCSContentFileStream (new FileInputStream (fileToUpload),
     fileToUpload.getName (),
     "application/octet-stream", fileToUpload.length ());

//Upload stream and retrieve ticket
String ticket = documentAPI.uploadStream (ctx, fileStream);
```

Get creatable applications

To attach an associated set of metadata to the file that was uploaded, you must first decide which application type you wish to contribute the document as. You can retrieve a list of creatable application types with the following code.

```
ISCSFixedApplicationResponse response = documentAPI.getCreatableApplications (ctx);

//Iterate through and print out application names.
ISCSFixedApplication application;
Iterator iter = response.getApplications ().iterator ();
while (iter.hasNext ()) {
       application = (ISCSFixedApplication)iter.next ();
       System.out.println (application.getName ());
}
```

Get 'create document' request

Once the application type has been determined, you can get the object that defines the 'create document' request for that application. This request contains each metadata field that is associated with the chosen application type.

Note: There is no enforcing of required metadata fields; be careful that the data is valid before contributing.

```
//Application object here assumed to be retrieved prior to snippet.
ISCSFixedCreateDocumentRequest createRequest =
     documentAPI.getCreateDocumentRequest (ctx, application);

//Iterate through each associated field and print its name.
ISCSFixedUserField field;
Iterator iter = createRequest.getUserFields ().iterator ();
while (iter.hasNext ()) {
       field = (ISCSFixedUserField)iter.next ();
       System.out.println (field.getName ());
}
```

A complete 'create document' example

To finish the request, a reference ID must be set (this is a user-identified ID that is used to track the request), the request URI must be specified (this is the ticket value retrieved during the upload process), and finally, the metadata fields must be populated with the desired data.

```
File fileToUpload = new File ("c:\\TestDocument.tif");

//Create a new SCSContentStream
SCSContentFileStream fileStream =
     new SCSContentFileStream (new FileInputStream (fileToUpload),
     fileToUpload.getName (),"application/octet-stream", fileToUpload.length ());

//Upload stream and retrieve ticket
String ticket = documentAPI.uploadStream (ctx, fileStream);

ISCSFixedApplicationResponse response = documentAPI.getCreatableApplications (ctx);

//Just retrieve the first application in the list for this example.
ISCSFixedApplication application = response.getApplications ().get (0);
```

```
ISCSFixedCreateDocumentRequest createRequest =
     documentAPI.getCreateDocumentRequest (ctx, application);

//Iterate through each associated field and set its value to the exampleData
//for this example (a real application would set these to meaningful values).
ISCSFixedUserField field;
Iterator iter = createRequest.getUserFields ().iterator ();
String exampleData = "exampleData";
while (iter.hasNext ()) {
     field = (ISCSFixedUserField)iter.next ();
     field.setValue (exampleData);
}

//Set the reference ID and URI
createRequest.setReferenceID ("refOne");
createRequest.setURI (ticket);

//Make the create request.
ISCSFixedCreateDocumentResponse createResponse =
     documentAPI.createDocument (ctx, createRequest);

//You can get the identifier of the newly created object like this
ISCSFixedID identifier = createResponse.getIdentifier ();
```

Fixed Process API

The ISCSFixedProcessAPI is used to query/effect information about current state of processes within the image server. The image server has a large set of Business Process functionality available through the Fixed API. This ranges from querying the current state to creating and moving items through the defined processes.

In each of the following examples, the CISApplication object has been initialized and the ISCSFixedProcessAPI has been defined as processAPI:

```
ISCSFixedProcessAPI processAPI =
     cisApplication.getUCPMAPI ().getFixedAPI ().getProcessAPI ();
```

This example also assumes that a context object has been created that refers to a SIS adapter:

```
ISISContext ctx = cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();
ctx.setUser ("myUserID");
ctx.setAdapterName ("sisadapter");
```

Getting package templates

A package template is the structure used to define a package. You can query the defined templates with the following code:

```
Iterator iter = packageTemplates.iterator ();
ISCSFixedPackageTemplate template;
while (iter.hasNext ()) {
       template = (ISCSFixedPackageTemplate)iter.next ();
       System.out.println (template.getTemplateID ());
       System.out.println (template.getTemplateName ());
       System.out.println (template.getDescription ());
       System.out.println (template.isCreatable ());
}
```

Getting process profiles

You can also retrieve the available process profiles:

```
List processProfiles = processAPI.getProcessProfiles (ctx);

Iterator iter = processProfiles.iterator ();
ISCSFixedProcessProfile profile;

while (iter.hasNext ()) {
       profile = (ISCSFixedProcessProfile)iter.next ();
       System.out.println (profile.getProfileID ());
       System.out.println (profile.getProfileName ());
       System.out.println (profile.getProfileType ());
       System.out.println (profile.hasPrompts ());
       System.out.println (profile.isLastUsed ());
}
```

Getting prompted profiles

You can also retrieve the available prompted profiles:

```
//Assumed here that the profile object (ISCSFixedProcessProfile) was
//retrieved prior to snippet
List promptedProfiles = processAPI.getProfilePrompts (ctx, profile);

Iterator iter = promptedProfiles.iterator ();
ISCSFixedPromptedProfile promptedProfile;

while (iter.hasNext ()) {
       promptedProfile = (ISCSFixedPromptedProfile)iter.next ();
       System.out.println (promptedProfile.getProfileID ());
       System.out.println (promptedProfile.getProfileName ());
       System.out.println (promptedProfile.hasPrompts ());

       if (promptedProfile.hasPrompts) {
              List prompts = promptedProfile.getPrompts ();
              Iterator promptIter = prompts.iterator ();
              ISCSFixedProfilePrompt prompt;
              SCSFixedProcessOperator processOperator;
              while (promptIter.hasNext ()) {
                     prompt = (ISCSFixedProfilePrompt)promptIter.next ();
                     System.out.println ("Display Text: "
                                          + prompt.getDisplayText ());
                     System.out.println ("Field ID: "
                                          + prompt.getFieldID ());
                     System..out.println ("Field Type:"
                                          + prompt.getFieldType ());
                     System.out.println ("Pick List Name: "
                                          + prompt.getPickListName ());
                     System.out.println ("User Input: "
                                          + prompt.getUserInput ());
                     System.out.println ("Distinct List Name: "
                                          + prompt.getDistinctListName ());

                     processOperator = p.getOperator ();

                     System.out.println ("Description: "
                                  + processOperator.getDescription ());
                    System.out.println ("Display Text: "
                                  + processOperator.getDisplayText ());
                    System.out.println ("Type: "
                                  + processOperator.getType ());
                    }
          }
}
```

Getting packages by profile

This is an example of getting a list of package objects by the profile they are associated with:

```
//Assumed here that the profile object (ISCSFixedProcessProfile)
//was retrieved prior to snippet

//You can also get packages by prompted profile via the getPackagesByPrompt
//method or by explicit ID via the getPackageByID method.
List packages = processAPI.getPackagesByProfile (ctx, profile);

Iterator iter = packages.iterator ();
ISCSFixedPackage package;

while (iter.hasNext ()) {
       package = (ISCSFixedPackage)iter.next ();
       System.out.println ("Create Date: " + package.getCreateDate ());
       System.out.println ("Creator Name: " + package.getCreatorName ());
       System.out.println ("Due Date: " + package.getDueDate ());
       System.out.println ("In Queue Date: " + package.getInQueueDate ());
       System.out.println ("Locked By Name: " + package.getLockedByName ());
       System.out.println ("Package ID: " + package.getPackageID ());
       System.out.println ("Package Type: " + package.getPackageType ());
       System.out.println ("Priority: " + package.getPriority ());
       System.out.println ("Process Name: " + pkg.getProcessName ());
}
```

Creating a package

This is an example of creating a package:

```
//Method creates an empty package to populate. Specify the adapter you are
//going to create package in as a parameter.
ISCSFixedCreatePackage createPkg = processAPI._createPackage ("sisadapter");

//Set a request ID for reference later.
createPkg.setRequestID ("Req_01");

//Set the template name you wish this package to be associated with.
createPkg.setTemplate ("VoucherPackage");

//Create the package.
ISCSFixedCreatePackageResult result = processAPI.createPackage (ctx createPkg);

//Echo back the results of request.
System.out.println ("Decision Text: " + result.getAttachmentCount ());
System.out.println ("Package ID: " + result.getPackageID ());
System.out.println ("Request ID: " + result.getRequestID ());
System.out.println ("Is Created: " + result.isCreated ());
System.out.println ("Is Journal Entries Made: " + result.isJournalEntriesMade ());
System.out.println ("Is Placed In Flow: " + result.isPlacedInFlow ());
System.out.println ("Is Values Set: " + result.isValuesSet ());
System.out.println ("Is XML Metadata: " + result.isXMLMetaData ());
```

Setting package fields

This is an example of setting the metadata associated with a package:

```
//Package assumed to be retrieved prior to this snippet
List fields = processAPI.getPackageFields (ctx, package.getPackageID ());
Iterator iter = fields.iterator ();
ISCSFixedPackageField field;

while (iter.hasNext ()) {
       field = (ISCSFixedPackageField)iter.next ();

       //Set the SupplierName field to a new value.
       if (field.getFieldName ().equalsIgnoreCase ("SupplierName")) {
              field.setUserInput ("Software Center Inc.");
       }

       //Otherwise just print out current value.
       System.out.println ("Name : " + field.getFieldName ());
       System.out.println ("Value: " + field.getUserInput ());
}

//Set the new value.
results = processAPI.setPackageFields (ctx, package.getPackageID (), fields);

//The return object is simply a list of the fields that were set.
```

Placing a package in flow

You can place a package into a process with the following code:

```
//Create the request object.
ISCSFixedPlaceInFlowRequest req = processAPI._createPlaceInFlowRequest
("sisadapter");

//Retrieve a list of possible start events for the associated template
List startEvents = processAPI.getTemplateStartEvents (ctx, "VoucherPackage");

//Pick the appropriate start event.
ISCSFixedTemplateStartEvent se = (ISCSFixedTemplateStartEvent)startEvents.get (1);

//Package object assumed to be created prior to snippet.
//Populate request object
req.setPackageID (package.getPackageID ());
req.setComment ("Attempt to place in flow");
req.setProcessName ("Demo AP Process");
req.setStartEvent (se.getEventName ());

//Perform the request, this method is capable of processing several requests
//at once here we just do the one.
List results = processAPI.placePackageInFlow (ctx, Arrays.asList (new
Object[]{req}));

//We only performed one request so there should only be one result.
ISCSFixedPlaceInFlow pif = (ISCSFixedPlaceInFlow)results.get (0);

//Echo result object
System.out.println ("Package ID: " + pif.getPackageID ());
System.out.println ("Request Index: " + pif.getRequestIndex ());
System.out.println ("Was placed in flow: " + pif.wasPlacedInFlow ());
```

Locking and unlocking a package

This is an example of locking and unlocking a package:

```
//Package object assumed to be retrieved prior to example:
results = processAPI.lockPackage (ctx, package);

//The results object is a list of results, we only locked one package
//so there should be only one result
ISCSFixedPackageResult result = (ISCSFixedPacakgeResult)results.get (0);

//Echo Results
System.out.println ("Has succeeded: " + result.hasSucceeded ());
System.out.println ("Package ID: " + result.getPackageID ());
System.out.println ("Auto Attachment Type ID: " + result.getAutoAttachmentTypeID
());

//If you want to lock a package in queue, repeat the above code and replace the
//method with lockPackageInQueue and specify the queueName as an additional
//String parameter.

//Unlock the package in the same way
results = processAPI.unlockPackage (getSISContext (), package);
```

CHAPTER 5

Understanding the Common API

The UCPM API is modeled into a set of Services APIs, which are API calls that communicate with the target server, and into ISCSObject objects, which are the returned value objects from the server.

The Common API is a branch of the UCPM API that deals with both Stellent Content Server and Stellent Image Server, by providing a common set of APIs that are shared between both the Active API and the Fixed API. These include searching, contribution, and information querying.

In this section:

- Accessing the Common API
- Common metadata maps
- Developing with the Common API
- Using the Common API

Important: The current version of CIS uses the UCPM API which is not compatable with CIS 7.2. See “Migrating from Version 7.2” on page 73 for more information.

Note: See the JavaDocs for information on the Class/Interface, Field, and Method descriptions. The complete CIS JavaDoc is located in the /docs/cis-javadoc-all.zip file (the /docs/cis-javadoc-ucpm.zip file contains only the UCPM API information).

Accessing the Common 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. Public interface IUCPMAPI is the locator for the Active, Fixed, and Common API objects. The Common API is avalable via getCommonAPI (), which returns a reference to the SCSCommonAPI object.

Note: The fully qualified method name is “CISApplication.getUCPMAPI ().getCommonAPI ()”.

The Common API comprises:

- **ISCSCommonAdministrativeAPI** — used to create, load, and persist common metadata maps.
- **ISCSCommonAdministrativeAdapterAPI** — implementation of the administrative commands.
- **ISCSCommonDocumentAPI** — used to contribute content to a repository and migrate content from one repository to another.
- **ISCSCommonSearchAPI** — implementation of the search commands used to perform federated queries across multiple repositories.

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, etc. Commands can either be retrieved by their full command name or by browsing all available commands.

Note: The fully qualified class name is "com.stellent.command.ICommandFacade".

Example using ISCSCommonSearchAPI:

```
ISCSCommonSearchAPI commandAPI =
     (ISCSCommonSearchAPI)m_commandFacade.getCommandAPI
     ("search", com.stellent.command.ICommandFacade.TYPE_UCM_COMMON_API);
```

Common metadata maps

To create a mapping, you can simply create an XML file. This example illustrates a common metadata map:

```
<!--
    Default metadata map for the different adapter types. These values will be used
if a metadata map is not specified. The name attribute corresponds to the adapter
type (i.e., "scsadapter" is a Stellent Content Server Adapter and "sisadapter" is a
Stellent Image Server adapter).
-->
<metamap>
  <adapter name="scsadapter">
    <common>
      <map name="Title" native="dDocTitle" />
      <map name="Author" native="dDocAuthor" />
      <map name="Date" native="dInDate" />
      <map name="ID" native="dID" />
    </common>

    <contribution>
    <!-- native fields are used only for contribution -->
    <native name="comments" default="" override="true" />

    <!-- common fields refer back to common map -->
    <!-- can also be tagged with default and override attributes -->
      <common name="title" />
      <common name="author" />
    </contribution>
  </adapter>

  <adapter name="sisadapter">
    <common>
      <map name="Title" native="DocType" objectClass="SImageHRAppIndex" />
      <map name="Author" native="EmployeeName" objectClass="SImageHRAppIndex" />
      <map name="Date" native="DocDate" objectClass="SImageHRAppIndex" />
      <map name="ID" native="SSN" objectClass="SImageHRAppIndex" />
    </common>
  </adapter>
</metamap>
```

This metadata map specifies both a content server ('scsadapter') and an image server ('sisadapter'). The top-level node is <metamap> which can contian any number of <adapter> nodes. The <adapter> node takes a name attribute, which should correspond to the adapter that is configured in the adapter config XML for the repository you want to define. The above metadata map contains two adapters: one for a content server (‘scsadapter’), the other for an image server (‘sisadapter’).

Each <adapter> node can have two sub-nodes: <common> and <contribution>. The <common> node is used to create metadata field mappings. To create a field mapping, you need create only a sub-node <map> under the <common> node, and then provide the attributes ‘name’ and ‘native’.

- The ‘name’ attribute is the common name you will set for each adapter you configure and expect to make common calls on.
- The ‘native’ attribute is the name of the metadata field as seen by the specific repository. For example, in the above example, we have mapped the common name ‘title’ to the native name ‘dDocTitle’ on a content server instance.

To create a similar mapping for an image server, you need to specify an additional attribute. An image server repository allows different metadata models to be present in the same repository and separates them by Application type. The ‘objectClass’ attribute to the <map> node corresponds to this Application type.

In the above example, the same common name ‘title’ is mapped to the native ‘DocType’ of the application ‘SImageHRAppIndex’ in the image server repository defined by the adapter name ‘sisadapter’.

The <contribution> node is used for moving pieces of content between disparate repositories. The <common> sub-nodes refer back to common name mappings set up in the <common> node, and are used to provide a “common” metadata name mapping to a “native” metadata name.

Two additional attributes can be specified (if needed) for the <contribution> node:

- A default value that will be used during the contribution if no value is found.
- An override parameter that indicates that the default value should prevail even if a value is found.

If you want to use a default or override value for a native field for which there is no common mapping, you can specify this by adding a <native> node with the ‘default’ or ‘override’ attributes.

Developing with the Common API

Once the CISApplication object has been initialized, you can retrieve the Fixed API objects using this code:

```
CISApplication cisApplication =
     CISApplication.initializeServer (myHost, myPort);
     cisApplication.getUCPMAPI ().getCommonAPI ();
```

Many of the Common API calls require an ISCSCommonContext, which is a container for the specific context objects that refer to the various adapters that the call is meant to operate against.

For example, this code creates a list of repositories to query and sets up your common context for both a content server and an image server repository:

```
//Create list of repositories to query.
ISCSContext activeContext =
     cisApplication.getUCPMAPI ().getActiveAPI ()._createSCSContext ();
ISISContext fixedContext =
     cisApplication.getUCPMAPI ().getFixedAPI ()._createSISContext ();

ISCSCommonContext commonContext =
     cisApplication.getUCPMAPI ().getCommonAPI ()._createCommonContext ();

activeContext.setAdapterName ("scsadapter");
activeContext.setUser ("sysadmin");

fixedContext.setAdapterName ("sisadapter");
fixedContext.setUser ("Administrator");

commonContext.addContext (activeContext);
commonContext.addContext (fixedContext);

//Your common context is now setup to hit two repositories:
//a content server and an image server.
```

Using the Common API

The Common Search and Common Document APIs are discussed and sample code provided. These APIs perform the task of searching across multiple repositories, contributing content to a repository, and migrating content from one repository to another.

Note: Additional samples can be found in the SDK/Samples/CodeSamples/ directory.

Common Search API

The ISCSCommonSearchAPI is the implementation of the search commands used to perform federated queries across multiple repositories. A search will return an ISCSCommonSearchResponse object. This object contains search status information and a list of ISCSCommonContent objects which identify which adapters to query and what user information to use.

To perform a federated search

You can perform a search against as many repositories as you have configured in your adapter configuration. By populating the Common Context and creating a metadata map, you can begin to execute search queries that replicate the Verity syntax that the content server accepts.

However, when putting common names into the query, the '$' escape-character must be used. For example, searching for authors containing the string ‘Farley’ would result in this query:

```
"$author <substring> 'Farley'"
```

The Common API then substitutes the common name $author for the appropriate native name. For example, in the case of the content server, this will be replaced by ‘dDocAuthor’.

After the results are obtained, you can get an iterator over the results, and specify page size and sort criteria. The sort criteria can be “Ascending” or “Descending” over any commonly defined metadata field.

The following example assumes that the CISApplication, common context, and common metadata map have already been created and initialized:

```
//Perform search retrieving only 100 results per adapter
//configured within the common context.
ISCSCommonSearchResponse searchResponse =
     m_commonSearch.search
     (commonContext,"$author <substring> 'Farley'", metadataMap, 100);

//You can get the total results
System.out.println ("Total Count: " + resp.getTotalCount ());

//Or the count by adapter
System.out.println ("SIS Count: " + searchResponse.getCount ("sisadapter");
System.out.println ("SCS Count: " + searchResponse.getCount ("scsadapter");

//Now that we have some results we can create our sort criteria
ISCSSortCriteria sortCriteria = new SCSSortCriteria ();

//Sort by title in ascending order.
sortCriteria.setSortField ("Title");
sortCriteria.setSortOrder (ISCSSortCriteria.ASCENDING);

//Get result list based on this ordering.
resultList = searchResponse.getResults (sortCriteria);

//Get a special iterator object that has convenience methods around paging results.
//This takes in a page size.
ISCSResultIterator resultIterator = resultList.getResultIterator (20);

//Print the number of pages within iterator. (based on page size specified earlier)
System.out.println ("Total pages: " + resultIterator.getTotalPages ());

ISCSProperty property;
ISCSContent content;
for (int i = 0; i < iter.getTotalPages (); iter.pageForward ()) {
System.out.println ("Current Page: " + resultIterator.getCurrentPage ());
     System.out.println ("Can page forward: " + iter.isCanPageForward ());

     while (iter.hasNext ()) {
     //The iterator can get the next piece of content like this
     content = resultIterator.nextContent ();

     //The ISCSContent object can find a value with a common name and a metadata map
     property = content.getProperty ("author", metadataMap);
     System.out.println ("Author : + property.getValue ().getStringValue ());
     }
}
```

Common Document API

The ISCSCommonDocumentAPI is used to contribute content to a repository and migrate content from one repository to another. The tasks of common migration and common retrieval are discussed and sample code provided.

Common migration

In the following example, a piece of active content is moved from a content server to an image server. This example assumes that the CISApplication, common context, and an ISCSActiveContent object have already been retrieved:

```
//In this example we'll create our metadata map programmatically
ISCSCommonMetadataMap metadataMap =
     cisApplication.getUCPMAPI ().getCommonAPI ()
     .getAdministrativeAPI ()._createMetadataMap ();

//Setup the common mappings for type and description.
metadataMap.addMapping ("type", "dDocType", "scsadapter");
metadataMap.addMapping ("description", "dDocTitle", "scsadapter");

metadataMap.addMapping ("type", "DocType", "sisadapter");
metadataMap.addMapping ("description", "Description", "sisadapter");

//Create a contribution map for the sisadapter for the "UniversalMain" application
//and give the common contrib map a name of "Universal"
ISCSCommonContributionMap sisContribMap =
     metadataMap._createContributionMap
     ("sisadapter", "Universal", "UniversalMain");

//The defaults for the common fields are empty if not found,
//and set not to override already existing values.
sisContribMap.addCommonContributionField ("type", "", false);
sisContribMap.addCommonContributionField ("description", "", false);

//Map the ADACCT Content Server type to the common "Universal" name.
ISCSCommonContributionMap scsContribMap =
     metadataMap._createContributionMap ("scsadapter", "Universal", "ADACCT");

//Retrieve the stream for our already retrieved content object.
SCSContentFileStream activeStream =
     cisApplication.getUCPMAPI ().getActiveAPI ().getFileAPI ().getFile
     (scsContext, content.getActiveContentID ());

//Perform the contribution.
ISCSCommonCheckinResponse checkinResp =
     m_documentAPI.checkinStream
    (fixedContext, "Universal", metadataMap, content, stream);

//Get the identifier for the newly created content item.
ISCSID id = checkinResp.getIdentifier ()
```

Common retrieval

After you have retrieved a piece of ISCSContent or an ISCSID from a search, you may query what content pages are associated with this document.

- If the backing content is from a content server, the list of ISCSContentPage objects that you will receive will represent the streams for PrimaryFile, AlternateFile, and DynamicConversion.
- For the image server, the page objects represent the various pages of content that are returned by the repository.

The following example assumes that the CISApplication, common context, and an ISCSID object have already been been retrieved:

```
//This call gets you a list of available page objects (ISCSContentPage)
ISCSCommonDocumentAPI documentAPI =
     cisApplication.getUCPMAPI ().getCommonAPI ().getDocumentAPI ();
List pages = documentAPI.getContentPages (commonContext, identifier);

//Print out some properties about each page and save its stream to disk.
Iterator iter = pages.iterator ();
```

```
ISCSContentPage page;
InputStream in;
FileOutputStream out;

while (iter.hasNext ()) {
page = (ISCSContentPage)iter.next ();
System.out.println ("Name: " + page.getName ());
System.out.println ("Description: " + page.getDescription ());
System.out.println ("Source Adapter: " + page.getSourceAdapter ());

//Get the input stream associated with this page.
in = documentAPI.getContentStream (commonContext, page);
out = new FileOutputStream (page.getName ());

//Write the stream to disk.
byte[] bytes = new byte[64];
int read;
while ((read = in.read (bytes)) != -1) {
out.write (bytes, 0, read);
}
in.close ();
out.close ();
}

//Get the primary stream directly by passing in the identifier.
in = documentAPI.getContentStream (commonContext, identifier);
```

In the code above, note that you may also get the primary stream directly by passing in the identifier for the content page to getContentStream (), on the content server this will return the PrimaryFile, on the image server this will return the native file.

CHAPTER 6

Extending Commands

Stellent Content Integration Suite (CIS) contains a number of APIs, which are a collection of command objects of a given type grouped in common categories. The CIS APIs include the UCPM API, which, in turn, includes the Active API, Fixed API, and Common API.

The CIS layer can be extended to allow users to add in their own custom commands, and, in turn, their own custom APIs. Once a command is authored, the provided build environment will create the necessary files, automatically generate the wrapper API classes, and generate any necessary configuration files.

This section explains how to extend the UCPM API layer:

- About APIs
- ICommand definition
- Creating an ICommand object with the CommandBuilder SDK
- Adding a new command
- Building a command

Note: See the JavaDocs for information on the Class/Interface, Field, and Method descriptions. The complete CIS JavaDoc is located in the /docs/cis-javadoc-all.zip file (the /docs/cis-javadoc-ucpm.zip file contains only the UCPM API information).

About APIs

A command is the main class of an API. All API calls get routed to a particular command for execution. Specifically, a command is a single Java class that implements the ICommand interface. Commands of the same type are organized into APIs. Inside each API, commands are further organized into categories, which can have sub-categories and commands as children.

Each ICommand class is marked with metadata that describes the API to which it belongs, what category it lives in, and its API name. The build process uses this information to dynamically create API classes and implementations that route API calls to the command objects for execution. For example, in the internal commands there is a command object that is marked as an “active” command type belonging to the “file” category with the name “getFile”.

During the build process, the ISCSActiveFileAPI file is generated with the method “getFile”. (Notice the ISCS<api type><category>API format; <api type> is the type of API marked on the command, and <category> is the category defined in the command.) A call to ISCSActiveFileAPI.getFile() translates internally to a lookup of the command ‘file.getFile’ of type ‘active’. The command is located, the properties are set, and the command is routed through the command layer. Eventually, the command’s execute method is called. In this way, commands make up methods that are part of categories grouped together into common APIs.

All the added commands created by the SDK will be of type “custom” and will make up the Custom API. If we create three commands—two in the “document” category and one in the “information” category—we would have two main interfaces that make up the custom SDK (as built by the Command Builder SDK using the default naming values): IAcmeDocumentAPI and IAcmeInformationAPI.

The IAcmeDocumentAPI interface would have two methods, one for each command in the "document" category. The IAcmeInformationAPI interface would have a single method for the one command in the ”information” category.

ICommand definition

A command is responsible for executing the business logic. In the case of content server commands, it is responsible for formulating the query for the content server. A command is usually implemented as a single Java class, which is decorated with a number of XDoclet tags that describe the behavior for functionality of the command.

The build process reads these tags to generate a strongly-typed API class that encapsulates calls to all commands belonging to a particular category. This step generates the API interface. For example, the search command described above is generated into a Java interface called ISCSActiveSearchAPI, which has two methods, both called search, that make calls into the search command.

The ICommand Java objects are simply JavaBeans that encapsulate business logic. A typical ICommand object will have a number of getter and setter methods for each property the command wishes to expose. The ICommand also implements a method, executeCommand(), that is called by the server when the command is executed. This is the main method of the command and it should hold the business logic.

Server-specific commands

Commands written to communicate with the content server can extend the SCSCommand class, which is found in the com.stellent.command.scs.impl package. The SCSCommand class does much of the work in order to prepare the class to make a content server request. For such a request, the subclass only needs to override the populateBinder () method and set the appropriate binder properties.

Likewise, to add a command that talks to the image server, the SISCommand class should be extended, as it provides the necessary infrastructure to communicate with the image server.

Creating an ICommand object with the CommandBuilder SDK

To create an ICommand object, you must:

1. Create the Command Java class that implements ICommand (often extends Command or one of its subclasses).
2. Implement the executeCommand () method with the appropriate business logic.
3. Decorate the command with the Stellent XDoclet tags.

Command development kit

Included in the CIS distribution is a development kit for building new commands and extending existing commands. Located in the <install_dir>/sdk/CommandBuilder directory, the kit includes an Ant build.xml file that allows you to build the commands and package them appropriately. Also included are all the necessary library files and distribution files for creating a new distribution EAR file and client JAR files.

To run the build, from the command line, type:

```
cd $installdir/sdk/CommandBuilder
ant dist
```

Note: Ant must be installed for this process to work properly (see http://ant.apache.org/).

All new source files should be placed in the <install_dir>/sdk/CommandBuilder/src/java directory in their appropriate package structure. This ensures the Ant script will properly pick up the files when performing a build. Also, any new libraries required for the compilation need to be placed in an appropriate sub-directory of the <install_dir>/sdk/library directory. Here is a brief definition of each directory in the build SDK:

Directory	Description
.../library	The SDK shared library repository (contains the libraries needed for building).
src/java	The source files for the new commands.
target	Directory generated during the build; holds the classes and other build-related files (after the build is run; holds the built files).

There are a number of defined Ant targets in the build.xml file that can be executed, as described in the following table. All the targets can be run with the same syntax (replace <target> with a valid target name):

```
cd $installdir/sdk/CommandBuilder
ant <target>
```

Target	Description
compile	The default target; it is run if no target is specified. It will generate the necessary files and then compile the generated files along with the files in the src directory.
jar	Creates the custom_commands.jar file after compiling the source files. This file is not deployable by itself; it is used to create a deployment with the "dist" target.
dist	Creates a distribution. This will generate the deployable distribution file that can be re-deployed into the target application server or servlet engines; it will generate the JAR, EAR, and WAR files.
clean	Cleans up the build; removes any generated files created during previous builds.

The build.properties file, which is in the same directory as the build.xml file, allows you to override property values with values specific to your build. The following table describes some properties (see the build.xml file for more property information):

Property	Description
api.identifier	Should be your company name, starting with a capital letter. This name will be used when generating API names during the build.
api.package	The java package name for your generated API classes. Suggested name is com.<company>.integration.scsapi. (Note: it should not start with "com.stellent").

SCS Command components

An SCS Command (Stellent Content Server Command) is a single Java object, with bean properties, that implements the interface ISCSCommand. To write a command that communicates with the content server, one would create an ISCSCommand.

There are a number of files associated with the SCS Command, such as the ISCSCommandAPI, the commandbeans.xml, the API implementation class and the CommandDescriptor bean. Each of these files are generated, via XDoclet, during the build process.

JavaDoc tags

The logic for generating each of the associated files is embedded in the Command Bean via XDoclet entries that are read during the build. The tags tell the build system which IDC Service (content server service) is being executed, the command name and category, and what properties are available to the Command API.

scs.command

This tag is placed at the top of each SCS Command Bean class. It has four attributes:

Attribute	Description
name	The name of the command method.
category	The category the command resides in.
service	The name of the IDC Service (content server service) that will be executed.
returnType	The full class name of the Object that will be returned from this SCS Command. ■ If omitted, the default is idcbean.data.LWDataBinder.

For example, if we had a command that was going to call the ADD_USER service, we would add the following tags to our SCS Command Bean:

```
/**
 * @scs.command name="addUser"
 *              category="directory"
 *              service="ADD_USER"
 */
public class AddUserCommand extends SCSCommand {
//class definition follows...
}
```

scs.commandproperty

This tag is placed on the *getter* for each property that will be made available via the SCS Command API and has one property that may be set. The ‘optional’ property can be either “true” or “false” and tells the SCS Command layer that this is an optional parameter. For example:

```
scs.commandproperty optional="true"
```

scs.method

This tag describes the method that will be exposed via the SCS Command API that is generated from this command. There can be any number of scs.method tags. Each one represents a single method. The api property must be set to ‘custom’; the three other properties can be specified.

Property	**Description**
name	The name of the API method. ▪ If not set, it defaults to the name of the command.
parameters	A comma-separated list of parameters that represents the method signature. Only parameters identified by the scs.commandproperty may be specified; the parameters are specified using their JavaBean name (i.e., for the method getName(), the parameter is ‘name’).
returnType	You can optionally set returnType, which is the fully qualified name of the class to be returned or “void” if the method does not return an object. ▪ If no returnType is set, the default is "com.stellent.ucpm.active.ISCSServerResponse".
api	The name of the API to generate: this must be set to ‘custom’.

```
/**
 * @scs.command name="addUser"
 *             category="directory"
 *             service="ADD_USER"
 * @scs.method parameters="SCSContext, userName, authType, fullName"
 *            api="custom"
 */
public class AddUserCommand extends SCSCommand {
  protected String m_userName = null;
  protected String m_authType = null;
  protected String m_fullName = null;

  /**
   * @scs.commandproperty
   */
  public String getUserName () {
    return m_userName;
  }

  public void setUserName (String name) {
    m_userName = name;
  }

  /**
   * @scs.commandproperty
   */
  public String getAuthType () {
    return m_authType;
  }
  public void setAuthType (String authType) {
    m_authType = authType;
  }

  /**
   * @scs.commandproperty optional="true"
   */
  public String getFullName (String fullName) {
    m_fullName = fullName;
  }

  public void setFullName (String fullName) {
    m_fullName = fullName;
  }

  //other implementation follows...
}
```

Adding a new command

An individual SCS Command is made up of a single JavaBean that holds the logic for executing the command, which typically is a call to the content server. The command is responsible for providing the necessary properties that can be set through standard JavaBean conventions for the command to execute. The command is also responsible, optionally, for validating the command properties.

Creating a new command

The following example illustrates the creation of a new command based on a given IDC Service (content server service). In this example, we will create an addUser command in the new command category “directory”, which will allow us to add users to the content server via our command interface. This will be done by executing ADD_USER.

Step 1: Create the Java class that will become the command for implementing our custom logic.

In the /CommandBuilder/src/java directory, create a package structure for your command: com.acme.contentserver.commands.directory.

It is useful to organize the commands so their category names match their package names. Since our root command structure is at the package com.acme.contentserver.commands, we create a directory package to correspond with the “directory” category of our command and place the Java class in this package.

Step 2: Identify the service.

As mentioned, we will use the ADD_USER service (for a full list of available services and their usage, consult the Stellent Content Server documentation). We will then create the AddUserCommand Java class inside our created package space. The Java class should extend SCSCommand, as the SCSCommand class provides the infrastructure for executing IDC Services (content server services) easily. We also add the necessary XDoclet header labeling this class as a SCS Command Bean:

```
package com.acme.commands.contentserver.directory;
```

```
import com.stellent.common.bean.ParameterValidationException;
import com.stellent.command.scs.impl.SCSCommand;

/**
 * @scs.command category="directory"
 *              service="ADD_USER"
 *              name="addUser"
 */
public class AddUserCommand extends SCSCommand {
    //… insert implementation here
}
```

Step 3: Add our parameters.

The ADD_USER service has two required parameters: ‘dName’ and ‘dUserAuthType’ (for details, consult the Stellent Content Server documentation). So we add two parameters, ‘name’ and ‘authType’, to our command and label them with the appropriate JavaDoc comments that identify them as SCS Command Bean properties:

```
package com.acme.commands.contentserver.directory;
import com.stellent.common.bean.ParameterValidationException;
import com.stellent.command.scs.impl.SCSCommand;

/**
 * @scs.command category="directory"
 *              service="ADD_USER"
 *              name="addUser"
 * @scs.method parameters="SCSContext, name, authType"
 *             api="custom"
 */
public class AddUserCommand extends SCSCommand {
  protected String m_name = null;
  protected String m_authType = null;

  /**
   * @scs.commandproperty
   */
  public String getAuthType () {
    return m_authType;
  }

  public void setAuthType (String authType) {
    m_authType = authType;
  }

  /**
   * @scs.commandproperty
   */

  public String getName () {
    return m_name;
  }

  public void setName (String name) {
    m_name = name;
  }
}
```

Step 4: Add the code to execute the IDC Service.

This can be done by extending the populateBinder () method. This method is defined in the parent class, SCSCommand, and takes two parameters: ‘LWDataBinder’, which we will populate with the IDC Service (content server service) data; and ‘ICommandExecutionContext’, which we can use to look up services in the system. We will implement this method and set the appropriate parameters on the data binder:

```
package com.acme.commands.contentserver.directory;
import com.stellent.common.bean.ParameterValidationException;
import com.stellent.command.scs.impl.SCSCommand;

/**
 * @scs.command category="directory"
 *              service="ADD_USER"
 *              name="addUser"
 * @scs.method parameters="SCSContext, name, authType"
 *             api="custom"
 */
public class AddUserCommand extends SCSCommand {
  protected String m_name = null;
  protected String m_authType = null;

  protected LWDataBinder populateData
   (LWDataBinder binder, ICommandExecutionContext context) throws CommandException {

    //call the super class
    binder = super.populateBinder (binder, executionContext);

    //add in our properties
    binder.putLocal ("dName", getName ());
    binder.putLocal ("dUserAuthType", getAuthType ());

    //return our populated binder
    return binder;
  }

  //property definitions left out...
```

Optionally...

We can override the validate method to provide some parameter validation:

```
package com.acme.commands.contentserver.directory;
import com.stellent.common.bean.ParameterValidationException;
import com.stellent.command.scs.impl.SCSCommand;

public class AddUserCommand extends SCSCommand {
  public void validate () throws ParameterValidationException {
  //call the superclass first
  super.validate ();

  //the exception object
  ParameterValidationException exp = null;

  if (getUser () == null || getUser ().length () < 1) {
    (exp = getException (exp)).addError ("userName", "Username was not set");
  }

  if (getAuthType () == null || getAuthType ().length () < 1) {
    (exp = getException (exp)).addError ("authType", "Authtype was not set");
  }

  if (exp != null) {
    throw exp;
  }
}

//rest of the command follows...
```

Customizing the build

Before building the source code, some modifications need to be made to the build process to customize the output of the generated files. Open the CommandBuilder/ build.properties file in an editor and change the following properties values accordingly:

Property	Description
api.identifer	Name of your company, for example, Acme.
api.package	Package name of the generated APIs (which should not start with com.stellent), for example, com.acme.integration.stellentapi.

The build will generate a number of files that will create the API wrapper around your command objects. The files generated, and their specific use, are as follows (assuming the properties were not changed from their defaults), and are all generated in the /CommandBuilder/generated directory.

Generated file	Description
commandbeans-custom.xml	XML file containing the mapping of command names to command implementations.
com.acme.integration.stellent.api.AcmeCommandAPI	API accessor class, which is the class containing the methods to retrieve the individual custom APIs.
com.acme.integration.stellentapi.*	API interfaces and implementation classes; provides API wrapper to the command objects.

All the generated files are packaged, along with your custom classes, in the custom-commands.jar file, which is placed in the appropriate EAR or WAR file, depending on the appserver target of the build.

Building a command

After the new command is written, you must complete these steps in order to use the new SCS Command Bean:

1. Modify the build.properties file to reflect the target application server and the desired application package (e.g., com.acme.commands).
2. Build the command using the supplied Ant scripts 'dist' target.
3. Redeploy the updated EAR files (located in the target directory).
4. Reference the generated cis-custom-7.6.1.jar in any client application that needs to use the extended commands (located in the target directory).

Deploying the command

The build will have generated two EAR files called cis-server-7.6.1.ear and cis-admin-7.6.1.ear (for the server and administration application, respectively) in the sdk/CommandBuilder/target directory.

The new EAR files need to be deployed to the target application server. (Consult the installation and deployment instructions for your application server.)

The build will also generate a JAR file, called cis-custom-7.6.1.jar. This new JAR file needs to be deployed with all the client applications, such as portlets, before you can use the newly updated commands.

Using the new Command API

Once the command is built and deployed (on both the server and all clients), it can be consumed via the newly created custom API. The following code shows how to access the command accessor class, assuming the default naming scheme. We use the CISApplication class to retrieve our custom API accessor by name; from there, we can access all the generated API classes:

```
//Grab the Custom API Accessor class by its generated name.
AcmeCommandAPI acmeCommandAPI =
     (AcmeCommandAPI)m_cisApplication.getBean ("AcmeCommandAPI");

//Create our context.
ISCSContext context =
     m_cisApplication.getUCPMAPI ().getActiveAPI ()._createSCSContext ();
context.setUser ("sysadmin");

//Get our directory API object from the Command API.
IAcmeDirectoryAPI directoryAPI =
     acmeCommandAPI.getDirectoryAPI ();
ISCSServerResponse response = directory.addUser (context, "newuser", "local");
//do something with the returned server response object
```

Using the command from the Command Web Application

After deploying the new command and restarting the server, the new command will be available from the CIS Administration Application (http://localhost:port/scscis). By clicking “Browse APIs” and selecting the API type “Custom”, you will be able to browse your custom deployed APIs. From here, you can test the execution of the API.

APPENDIX A

Migrating from Version 7.2

If you are upgrading from Stellent Content Integration Suite version 7.2, you should be aware of a significant change in the product. CIS version 7.2 uses LWDBs (Light Weight Data Binders) as part of the integration architecture. In the current version, this layer has been replaced by the Universal Content and Process Management (UCPM) API.

To facilitate migration, you can continue to use the version 7.2 API and migrate code from it to the current release. However, compiling with the 7.2 API will result in returned "warning" messages. In version 8.0, the 7.2 APIs will be removed and attempting to compile will result in an error.

In this section:

- Initialization--START HERE
- API calls
- Property names and data access
- Property names and data access

Initialization--START HERE

CIS version 7.2 requires a call to the ICommandClient class to initialize the code. In the current release, initialization happens in the com.stellent.cis.CISApplication class, and the call is now more explicit: you specify the type of server and the location of the adapter configuration file (adapterconfig.xml) in the initialization call.

Note: See "Adapter configuration file" on page 24 for more information.

```
//7.2 code
ICommandClient commandClient = new CommandClient ();
commandClient.connect ();

//Current release code
CISApplication cisApplication = CISApplication.initializeServer
```

The code for the current release also has different methods for initializing a CIS client versus initializing a CIS server. In the current release, if you are connecting to another CIS server instance, you use the initializeClient () methods on the CISApplication class. If you are initializing a CISApplication as a server, you use one of the initializeServer () methods.

API calls

Calls to the Application Programming Interface (API) are now handled Active API portion of the UCPM API. The UCPM API differs from the 7.2 API in that all objects used in the UCPM API are part of the UCPM object model. The parameters to the APIs, and the return values, all implement ISCSObject. This allows for a much cleaner interface when calling the content server code.

All of the APIs available in the 7.2 API have been ported over, in one form or another, to the UCPM API. These APIs are now accessed using the CISApplication.getUCPMAPI ().getActiveAPI () method. The API names are similar to the 7.2 API names: the Active API objects are named ISCSActive<category>API.

APIs are no longer retrieved via the ICommandFaçade (although this method still works). Now, the appropriate method to access an API is to use the accessor methods off the CISApplication object. For example, to retrieve the Search API:

```
//7.2 code
ISCSSearchCommandAPI searchAPI =
     (ISCSSearchCommandAPI)m_commandFacade.getCommandAPI ("search");

//Current release code
ISCSActiveSearchAPI searchAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getSearchAPI ();
```

Notice the use of the CISApplication accessor methods. It is no longer required to look up methods using the command facade. In CIS 7.2, there was also a mechanism for looking up the Command Bean directly, using the ICommandFacade object. That method of making API calls is deprecated; all calls should go through the UCPM API.

Context

All APIs in the UCPM API still require an ISCSContext context object. The creation of the context object, however, has been moved to the SCSActiveAPI. This has been done because the other APIs of the UCPM API, including the Fixed API and the Common API, have their own versions of the context object. The code to create the context object should look like the following:

```
//Current release code
ISCSContext context =
     m_cisApplication.getUCPMAPI ().getActiveAPI ()._createSCSContext ();
context.setAdapterName ("myadapter");
context.setUser ("sysadmin");
```

API naming

All the 7.2 APIs have counterparts in the UCPM API. The naming convention used for each API is slightly different. In CIS 7.2, the API naming convention is ISCS<category>CommandAPI. In the current release, the naming convention is ISCSActive<category>API. Using this, you can determine how to find the new version of the API classes. For example, look at the following CIS 7.2 code that looks up the Document Checkin API and the Workflow API:

```
//7.2 code
ISCSDocumentCheckinCommandAPI checkinAPI =
     (ISCSDocumentCheckinCommandAPI)m_commandFacade.getCommandAPI
     ("document.checkin");
ISCSWorkflowCommandAPI workflowAPI =
     (ISCSWorkflowCommandAPI)m_commandFacade.getCommandAPI ("workflow");

//Current release code
ISCSActiveDocumentCheckinAPI checkinAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckinAPI ();
ISCSActiveWorkflowAPI workflowAPI =
     m_cisApplication.getUCPMAPI ().getWorkflowAPI ();
```

Method parameters

The 7.2 API and the UCPM API share similar names for method calls. For example, checkinFileStream () is available on both the ISCSDocumentCheckinCommandAPI and the ISCSActiveDocumentCheckinAPI interfaces. However, calling the method is different for most API calls. The parameters for the methods in the UCPM API often require UCPM objects as opposed to taking in simple types (strings, integers, etc.). For example, calling at the checkinFileStream () method from the 7.2 API would look like the following:

```
//7.2 code
ISCSDocumentCheckinCommandAPI checkinAPI =
     (ISCSDocumentCheckinCommandAPI)m_commandFacade.getCommandAPI
     ("document.checkin");

//create the context object
ISCSContext context = m_commandFacade.createSCSContext ();
context.setUser ("sysadmin");
context.setAdapterName ("myadapter");
Map metaMap = new HashMap ();
metaMap.put ("xComments", "Test Comment");

//Build the content stream
File file = new File ("test.txt");
SCSContentFileStream stream =
     new SCSContentFileStream (new FileInputStream (file),
     "test.txt", "text/plain", file.length ());

//Make the API call
LWDataBinder response =
     checkinAPI.checkinFileStream (context, "test_01", "ADACCT", "Public", null,
     "sysadmin", null, null, metaMap, stream);
```

With the current release, the checkinFileStream () method requires an ISCSActiveContent to be passed into the method. ISCSActiveContent, like all ISCSObject objects, can be created by using the '_create' method from the target API. In this instance, the ISCSActiveDocumentCheckinAPI has a method _createContent (), which creates an empty ISCSActiveContent object.

The object can then be set with the appropriate property values. Likewise, the method also needs an ISCSActiveContentID, which can be created by using the _createActiveContentID () method on the SCSActiveAPI object. Our new code would then look like:

```
//Current release code
ISCSActiveDocumentCheckinAPI checkinAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentCheckinAPI ();

//Create the context object
ISCSContext context =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().createSCSContext ();
context.setUser ("sysadmin");
context.setAdapterName ("myadapter");

//Create the ISCSActiveContent object
ISCSActiveContent content = checkinAPI._createContent ("myadapter");
content.setTitle ("My Title");
content.setAuthor ("sysadmin");
content.setSecurityGroup ("Public");
content.setComments ("My Comments");
```

```
//Create the content ID
ISCSActiveContentID contentID =
     m_cisApplication.getUCPMAPI ().getActiveAPI ()._createActiveContentID
     ("test_01", "myadapter");

//Build the content stream
File file = new File ("test.txt");
SCSContentFileStream stream = new SCSContentFileStream (file);

//Make the API call
ISCSDocumentCheckinResponse response =
     checkinAPI.checkinFileStream (context, content, contentID, stream);
```

Note the use of the '_create' methods to create new versions of the ISCSActiveContent and ISCSActiveContentID objects. Also note that the SCSContentFileStream now takes a java.io.File object in its constructor, allowing for easier creation of SCSContentFileStream objects.

Property names and data access

The UCPM objects are JavaBean objects. They follow the JavaBean standard for property names, including the naming convention (i.e., no lowercase 'd' or 'x' in front of the property name, as with content server property names).

Looking specifically at ISCSActiveContent, there are a number of properties available on the object: title, author, type, etc. To get the title from such an object, you would call:

```
activeContent.getTitle ();
```

Also, using the ISCSObject interface (which ISCSActiveContent implements), you can call the getProperty () method and pass in the property name to the object:

```
activeContent.getProperty ("title").getValue ();
```

Note that the above call returns an ISCSProperty object, which allows you to retrieve the value by using its accessor methods. Finally, because the content server supports any number of properties, you can retrieve the property by the native (content server) name by calling the getProperty () method:

```
activeContent.getProperty ("dDocTitle").getValue ();
```

These three calls are identical. They all return the same value internally and are all mapped to the same property. The same is true for the *setter* versions of the methods; calling setProperty ('dDocTitle', value) or setTitle (value) or setProperty ('title', value) all result in setting the title property to the given value.

LWDataBinder and ISCSServerResponse

The LWDataBinder was the primary object used to hold the returned server response from the content server. This object has been replaced in the UCPM Active API by the ISCSServerResponse interface. The server response interface inherits from ISCSObject and provides a more Java-friendly interface that supports the Java Collection classes. The following table provides a mapping between the LWDataBinder methods and the ISCSServerResponse methods:

LWDataBinder	**ISCSServerResponse**
getResultSet (String name)	getResultSet (String name)

LWDataBinder	ISCSServerResponse
getLocalData (String name)	getProperty (String name).getValue ()
getOptionList (String name)	getOptionList (String name)

Most API calls will extend ISCSServerResponse and provide a specific implementation for the call. For example, calling the getDocumentInformation () method on the ISCSActiveDocumentInformationAPI returns an ISCSDocumentInformationResponse. This object extends ISCSServerResponse and adds methods for getting the content object that was queried, the workflow associated with the document, and a list of the revision history. The available information from the returned service call is exposed in the returned interface and all information is wrapped into ISCSObjects.

The following code is a sample of a 7.2 API call for document information to retrieve a document title:

```
ISCSDocumentInfoCommandAPI infoAPI =
     (ISCSDocumentInfoCommandAPI)m_commandClient.getCommandFacade ().getCommandAPI
     ("document.information");
LWDataBinder dataBinder =
  infoAPI.getDocInfoByID (context, "12345");

//Get the document information result set
LWResultSet resultSet = dataBinder.getResultSet ("DOC_INFO");

//Get the title
dataBinder.first ();
String title = dataBinder.getStringValue ("dDocTitle");
```

Translating this to UCPM API calls results in the following:

```
ISCSActiveDocumentInformationAPI infoAPI =
     m_cisApplication.getUCPMAPI ().getActiveAPI ().getDocumentInformationAPI ();

ISCSActiveDocumentID documentID =
     m_cisApplication.getUCPMAPI ().getActiveAPI ()._createDocumentID ("12345");

ISCSDocumentInformationResponse response =
     infoAPI.getDocumentInformation (context, documentID);

String title = response.getContent ().getTitle ();
```

Glossary

The following terms are used in this Stellent product. They are listed here in alphabetical order.

Ant

Apache Ant. A Java-based build tool (http://ant.apache.org/).

API

Application Programming Interface. The specific method, prescribed by a computer operating system or an application, by which a programmer writing a program can make requests of the operating system or application.

applet

A J2EE component that typically executes in a web browser but can execute in a variety of other applications or devices that support the applet programming model.

applet container

A container that includes support for the applet programming model.

attribute (XML)

A qualifier on an XML tag that provides additional information.

authentication

The process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. The Java servlet specification requires three types of authentication—basic, form-based, and mutual—and supports digest authentication.

authorization

The process by which access to a method or resource is determined. Authorization depends on whether the principal associated with a request through authentication resides in a given security role. A security role is a logical grouping of users defined by the person who assembles the application. A deployer maps security roles to security identities, which may be principals or groups in the operational environment.

authorization constraint

An authorization rule that determines who is permitted to access a web resource collection.

basic authentication

An authentication mechanism in which a web server authenticates an entity via a user name and password obtained using the web application's built-in authentication mechanism.

binding (XML)

Generating the code needed to process a well-defined portion of XML data.

build file

The XML file that contains one or more asant targets. A target is a set of tasks you want to execute. When starting asant, you can select which targets you want to execute. When no target is given, the project's default target is executed.

business logic

Includes how we model objects in our application, how these objects are stored, how these objects interact with each other, and who can access and update these objects. It differs from presentation logic in that presentation logic is concerned how we display these objects to a user.

caller principal

The principal that identifies the invoker of the enterprise bean method.

certificate authority

A trusted organization that issues public key certificates and provides identification to the bearer.

Cascading Style Sheet

See CSS.

client-certificate authentication

An authentication mechanism that uses HTTP over SSL, in which the server and, optionally, the client authenticate each other with a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure.

Common Object Request Broker Architecture

See CORBA.

component

See J2EE component.

connection factory

See resource manager connection factory and JMS connection factory.

connector

A standard extension mechanism for containers that provides connectivity to enterprise information systems. A connector is specific to an enterprise information system and consists of a resource adapter and application development tools for enterprise information system connectivity. The resource adapter is plugged in to a container through its support for system-level contracts defined in the connector architecture.

connector architecture

An architecture for integrating J2EE products with enterprise information systems. It has two parts: a resource adapter provided by an enterprise information system vendor and the J2EE product that the resource adapter plugs into. The architecture defines a set of contracts that the resource adapter must support to plug into the J2EE product (example are transactions, security, and resource management).

container (EJB)

See EJB container.

container-managed sign-on

The mechanism whereby security information needed for signing on to a resource is supplied by the container.

container-managed transaction

A transaction whose boundaries are defined by an EJB container. An entity bean must use container-managed transactions.

content (XML)

In an XML document, the part that occurs after the prolog, including the root element and everything it contains.

context root

The name that gets mapped to the document root of a web application.

CORBA

Common Object Request Broker Architecture. A language-independent, distributed object model specified by the Object Management Group (OMG).

create method

A method defined in the home interface and invoked by a client to create an enterprise bean.

CSS

Cascading Style Sheet. A stylesheet used with HTML and XML documents to add a style to all elements marked with a particular tag to ensure that such elements display consistently in a web browser.

declarative security

The mechanisms used in an application that are expressed in a declarative syntax in a deployment descriptor.

deployment descriptor

An XML file provided with each module and J2EE application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

destination (JMS)

See JMS destination.

digest authentication

An authentication mechanism in which a web application authenticates itself to a web server by sending the server a message digest along with its HTTP request message. The digest is computed by applying a one-way hash algorithm to a concatenation of the HTTP request message and the client's password. The digest is typically much smaller than the HTTP request and doesn't contain the password.

distributed application

An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multi-tier (client-multiple, middleware-multiple servers).

Document Object Model

An API for accessing and manipulating XML documents as tree structures. DOM provides platform-neutral, language-neutral interfaces that enable programs and scripts to dynamically access and modify content and structure in XML documents.

DOM

See Document Object Model.

EAR

An Enterprise Archive file, that is, a JAR archive that contains a J2EE application.

EJB

See Enterprise JavaBeans.

EJB container

A container that implements the EJB component contract of the J2EE architecture. This contract specifies a run-time environment for enterprise beans that includes security, concurrency, life-cycle management, transactions, deployment, naming, and other services. An EJB container is provided by an EJB or J2EE server.

EJB home object

An object that provides the life-cycle operations (create, remove, find) for an enterprise bean. The class for the EJB home object is generated by the container's deployment tools. The EJB home object implements the enterprise bean's home interface. The client references an EJB home object to perform life-cycle operations on an EJB object. The client uses JNDI to locate an EJB home object.

EJB JAR file

A JAR archive that contains an EJB module.

EJB module

A deployable unit that consists of one or more enterprise beans and an EJB deployment descriptor.

EJB object

An object whose class implements the enterprise bean's remote interface. A client never references an enterprise bean instance directly; it always references an EJB object. The class of an EJB object is generated by a container's deployment tools.

EJB server

Software that provides services to an EJB container. For example, an EJB container typically relies on a transaction manager that is part of the EJB server to perform the two-phase commit across all the participating resource managers. The J2EE architecture assumes that an EJB container is hosted by an EJB server from the same vendor, so it does not specify the contract between these two entities. An EJB server can host one or more EJB containers.

EJB server provider

A vendor that supplies an EJB server.

element (XML)

A unit of XML data, de-limited by tags. An XML element can enclose other elements.

Enterprise JavaBeans

A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and secure.

Extensible Markup Language

See XML.

filter

An object that can transform the header or content (or both) of a request or response. Filters differ from web components in that they usually do not themselves create responses, but, rather, modify or adapt requests to, and responses from, a resource. A filter should not have any dependencies on a web resource for which it is acting as a filter so that it can be composable with more than one type of web resource.

filter chain

A concatenation of XSLT transformations in which the output of one transformation becomes the input of the next.

form-based authentication

An authentication mechanism in which a web container provides an application-specific form for logging in. This form of authentication uses Base64 encoding and can expose user names and passwords unless all connections are over SSL.

group

An authenticated set of users classified by common traits such as job title or customer profile. Groups are also associated with a set of roles. Every user that is a member of a group inherits all the roles assigned to that group.

handle

An object that identifies an enterprise bean. A client can serialize the handle and then later de-serialize it to obtain a reference to the enterprise bean.

home interface

One of two interfaces for an enterprise bean. The home interface defines zero or more methods for managing an enterprise bean. The home interface of a session bean defines **create** and **remove** methods, whereas the home interface of an entity bean defines **create**, **finder**, and **remove** methods. (See also remote interface.)

HTML

Hypertext Markup Language. A markup language for hypertext documents on the Internet. In addition to basic text formatting, HTML enables the embedding of images, sounds, video streams, form fields, and references to other objects (by means of URLs).

HTTP

Hypertext Transfer Protocol. The Internet protocol used to retrieve hypertext objects from remote hosts. HTTP messages consist of requests from client to server and responses from server to client.

HTTPS

HTTP layered over the SSL protocol (see SSL).

Hypertext Markup Language

See HTML.

Hypertext Transfer Protocol.

See HTTP.

initialization parameter

A parameter that initializes the context associated with a servlet.

J2EE

An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that enable the development, deployment, and management of multi-tier, server-based applications. This edition of the Java 2 Platform encompasses many specifications, including EJB, JMS, JSP and Servlets.

J2EE application

Any deployable unit of J2EE functionality. This can be a single J2EE module or a group of modules packaged into an EAR file along with a J2EE application deployment descriptor. J2EE applications are typically engineered to be distributed across multiple computing tiers.

J2EE component

A self-contained functional software unit supported by a container and configurable at deployment time. The J2EE specification defines several types of J2EE components: application clients and applets that run on the client; Java servlets and JSPs that run on the server; and EJBs that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and “standard” Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server or client container.

J2EE module

A software unit that consists of one or more J2EE components of the same container type and one deployment descriptor of that type. There are four types of modules: EJB, web, application client, and resource adapter. Modules can be deployed as stand-alone units or assembled into a J2EE application.

J2EE server

The runtime portion of a J2EE product. A J2EE server provides EJB or web containers, or both.

J2SE

Java 2 Platform, Standard Edition. The core Java technology platform.

JAR

Java Archive. A platform-independent file format that permits many files to be aggregated into one file.

Java 2 Platform, Enterprise Edition

See J2EE.

Java 2 Platform, Standard Edition

See J2SE.

Java Development Kit

See JDK.

Java Runtime Environment

See JRE.

Java Virtual Machine

See JVM.

Java Messaging Service

See JMS.

Java Naming and Directory Interface

See JNDI.

Java Secure Socket Extension

A set of packages that enable secure Internet communications.

JavaServer Pages

An extensible web technology that uses static data, JSP elements, and server-side Java objects to generate dynamic content for a client. Typically, the static data is HTML or XML elements, and in many cases the client is a web browser.

JavaServer Pages Standard Tag Library

A tag library that encapsulates core functionality common to many JSP applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization and locale-specific formatting tags, SQL tags, and functions.

JDK

Java Development Kit. A programming development environment for writing Java applets and applications. It consists of a runtime environment that “sits on top” of the operating system layer, as well as the tools and programming that developers need to compile, debug, and run applets and applications written in the Java language.

JMS

Java Messaging Service. An application program interface (API) that supports the formal communication known as messaging between computers in a network.

JMS administered object

A preconfigured JMS object (a connection factory or a destination) created by an administrator for the use of JMS clients and placed in a JNDI namespace.

JMS application

One or more JMS clients that exchange messages.

JMS client

A Java language program that sends or receives messages.

JMS connection factory

An object that enable JMS clients to create JMS connections.

JMS connection factory object

An object that encapsulates connection configuration information, and enables JMS applications to create a connection object.

JMS connection object

An object that represents an open communication channel between an application and the messaging system and is used to create a Session Object for producing and consuming messages. A connection creates server-side and client-side objects that manage the messaging activity between an application and JMS.

JMS destination

A JMS-administered object that encapsulates the identity of a JMS queue or topic.

JMS session object

An object that defines a serial order for the messages produced and consumed, and can create multiple message producers and message consumers. The same thread can be used for producing and consuming messages. If an application wants to have a separate thread for producing and consuming messages, the application should create a separate session for each function.

JMS provider

A messaging system that implements the Java Message Service as well as other administrative and control functionality needed in a full-featured messaging product.

JMS queue

A queue that defines a point-to-point destination type.

JMS session

A single-threaded context for sending and receiving JMS messages. A JMS session can be non-transacted or locally transacted, or participate in a distributed transaction.

JMS topic

A topic that identifies a publish/subscribe destination type.

JNDI

Java Naming and Directory Interface. An API that provides naming and directory functionality.

JRE

Java Runtime Environment. The JRE, also known as Java Runtime, is part of the Java Development Kit (JDK), a set of programming tools for developing Java applications. The JRE provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

JSP

See JavaServer Pages.

JSP action

A JSP element that can act on implicit objects and other server-side objects or can define new scripting variables. A JSP action follows the XML syntax for elements, with a start tag, a body, and an end tag; if the body is empty, it can also use the empty tag syntax. The tag must use a prefix. There are standard and custom actions.

JSP container

A container that provides the same services as a servlet container and an engine that interprets and processes JSP pages into a servlet.

JSP custom action

A user-defined action described in a portable manner by a tag library descriptor and imported into a JSP page by a taglib directive. Custom actions are used to encapsulate recurring tasks in writing JSP pages.

JSP custom tag

A tag that references a JSP custom action.

JSP page

See JavaServer Page.

JSP tag file

A source file containing a reusable fragment of JSP code that is translated into a tag handler when a JSP page is translated into a servlet.

JSP tag handler

A Java programming language object that implements the behavior of a custom tag.

JSP tag library

A collection of custom tags described via a tag library descriptor and Java classes.

JSPs

Several JavaServer Pages.

JSSE

See Java Secure Socket Extension.

JSTL

See JavaServer Pages Standard Tag Library.

JVM

Java Virtual Machine. An implementation of the Java Virtual Machine Specification that interprets compiled Java binary code to perform the instruction of a Java program.

keystore

A file containing the keys and certificates used for authentication.

message (JMS)

An asynchronous request, report, or event that is created, sent, and consumed by an enterprise application and not by a human. It contains vital information needed to coordinate enterprise applications in the form of precisely formatted data that describes specific business actions.

message consumer

An object created by a JMS session that is used for receiving messages sent to a destination.

message-driven bean

An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables can contain state across the handling of client messages, including an open database connection and an object reference to an EJB object. A client accesses a message-driven bean by sending messages to the destination for which the bean is a message listener.

message producer

An object created by a JMS session that is used for sending messages to a destination.

naming context

A set of associations between unique identifiers and objects.

naming environment

A mechanism that allows a component to be customized without the need to access or change the component's source code. A container implements the component's naming environment and provides it to the component as a JNDI naming context. Each component names and accesses its environment entries using the java:comp/env JNDI context. The environment entries are declaratively specified in the component's deployment descriptor.

Object Management Group

See OMG.

Object Request Broker

See ORB.

OMG

Object Management Group. A consortium that produces and maintains computer industry specifications for interoperable enterprise applications (http://www.omg.org/).

one-way messaging

A method of transmitting messages without having to block until a response is received.

ORB

Object Request Broker. A library that enables CORBA objects to locate and communicate with one another.

principal (authentication)

The identity assigned to a user as a result of authentication.

privilege

A security attribute that does not have the property of uniqueness and that can be shared by many principals.

public key certificate

Used in client-certificate authentication to enable the server, and optionally the client, to authenticate each other. The public key certificate is the digital equivalent of a passport. It is issued by a trusted organization, called a certificate authority, and provides identification for the bearer.

publish/subscribe messaging system

A messaging system in which clients address messages to a specific node in a content hierarchy, called a topic. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a node's multiple publishers to its multiple subscribers.

query string

A component of an HTTP request URL that contains a set of parameters and values that affect the handling of the request.

queue destination (JMS)

See JMS queue.

RAR

Resource Adapter Archive. A JAR archive that contains a resource adapter module.

realm

In the J2EE server authentication service, a complete database of roles, users, and groups that identify valid users of a web application or a set of web applications.

registry

An infrastructure that enables the building, deployment, and discovery of web services.

registry provider

An implementation of a business registry that conforms to a specification for XML registries (for example, ebXML or UDDI).

remote interface

One of two interfaces for an enterprise bean. (See also home interface.) The remote interface defines the business methods callable by a client.

Remote Method Invocation

See RMI.

resource adapter

A system-level software driver that is used by an EJB container or an application client to connect to an enterprise information system. A resource adapter typically is specific to an enterprise information system. It is available as a library and is used within the address space of the server or client using it.

A resource adapter plugs into a container. The application components deployed on the container then use the client API (exposed by the adapter) or tool-generated high-level abstractions to access the underlying enterprise information system. The resource adapter and EJB container collaborate to provide the underlying mechanisms—transactions, security, and connection pooling—for connectivity to the enterprise information system.

resource adapter module

A deployable unit that contains all Java interfaces, classes, and native libraries, implementing a resource adapter along with the resource adapter deployment descriptor.

resource manager

Provides access to a set of shared resources. A resource manager participates in transactions that are externally controlled and coordinated by a transaction manager. A resource manager typically is in a different address space or on a different machine from the clients that access it.

Note: An enterprise information system is referred to as a resource manager when it is mentioned in the context of resource and transaction management.

resource manager connection

An object that represents a session with a resource manager.

resource manager connection factory

An object used for creating a resource manager connection.

RMI

Remote Method Invocation. A technology that allows an object running in one Java virtual machine to invoke methods on an object running in a different Java virtual machine.

role (security)

An abstract logical grouping of users that is defined by the application assembler. When an application is deployed, the roles are mapped to security identities, such as principals or groups, in the operational environment.

In the J2EE server authentication service, a role is an abstract name for permission to access a particular set of resources. A role can be compared to a key that can open a lock. Many people might have a copy of the key; the lock does not care who you are, only that you have the right key.

role mapping

The process of associating the groups or principals (or both), recognized by the container with security roles specified in the deployment descriptor. Security roles must be mapped by the deployer before a component is installed in the server.

rollback

The point in a transaction when all updates to any resources involved in the transaction are reversed.

Secure Socket Layer

See SSL.

security constraint

A declarative way to annotate the intended protection of web content. A security constraint consists of a web resource collection, an authorization constraint, and a user data constraint.

security context

An object that encapsulates the shared state information regarding security between two entities.

server certificate

Used with the HTTPS protocol to authenticate web applications. The certificate can be self-signed or approved by a certificate authority (CA). The HTTPS service of the Sun Java System Application Server Platform Edition 8 will not run unless a server certificate has been installed.

server principal

The operating system principal that the server is executing as.

service element

A representation of the combination of one or more connector components that share a single engine component for processing incoming requests.

service endpoint interface

A Java interface that declares the methods that a client can invoke on a web service.

servlet

A Java program that extends the functionality of a web server, generating dynamic content and interacting with web applications by using a request-response paradigm.

servlet container

A container that provides the network services over which requests and responses are sent, and also decodes requests and formats responses. All servlet containers must support HTTP as a protocol for requests and responses, but can also support additional request-response protocols, such as HTTPS.

A "distributed servlet container" can run a web application that executes across multiple Java virtual machines running on the same host or on different hosts.

servlet context

An object that contains a servlet's view of the web application within which the servlet is running. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.

servlet mapping

Defines an association between a URL pattern and a servlet. The mapping is used to map requests to servlets.

session

An object used by a servlet to track a user's interaction with a web application across multiple HTTP requests.

session bean

An enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations, such as calculations or database access, for the client. Although a session bean can be transactional, it is not recoverable should a system crash occur. Session bean objects can either be stateless or maintain a conversational state across methods and transactions. If a session bean maintains state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

Simple Object Access Protocol

See SOAP.

SOAP

Simple Object Access Protocol. A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. Using XML technologies, it defines an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols.

SSL

Secure Socket Layer. A security protocol that provides privacy over the Internet. The protocol allows client-server applications to communicate in a way that cannot be eavesdropped upon or tampered with. Servers are always authenticated, and clients are optionally authenticated.

stateful session bean

A session bean with a conversational state.

stateless session bean

A session bean with no conversational state. All instances of a stateless session bean are identical.

topic destination (JMS)

See JMS topic.

transaction attribute

A value specified in an EJB’s deployment descriptor that is used by the EJB container to control the transaction scope when the enterprise bean’s methods are invoked. A transaction attribute can have the following values: Required, RequiresNew, Supports, NotSupported, Mandatory, or Never.

transaction isolation level

The degree to which the intermediate state of the data being modified by a transaction is visible to other concurrent transactions, and data being modified by other transactions is visible to it.

transaction manager

Provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

UDDI

Universal Description, Discovery and Integration. A platform-independent, XML-based registry for businesses to list themselves on the Internet.

Unicode

A standard defined by the Unicode Consortium that uses a 16-bit code page that maps digits to characters in languages around the world. Because 16 bits covers 32,768 codes, Unicode is large enough to include all the world’s languages, with the exception of ideographic languages, such as Chinese, that have a different character for every concept (http://www.unicode.org/).

Uniform Resource Locator

See URL.

Universal Description, Discovery and Integration

See UDDI.

Universal Standard Products and Services Classification

A schema that classifies and identifies commodities. It is used in sell-side and buy-side catalogs and as a standardized account code in analyzing expenditure.

UNSPSC

See Universal Standard Products and Services Classification.

URL

Uniform Resource Locator. A standard for writing a textual reference to data on the web. A URL looks like this: protocol://host/localinfo, where protocol specifies a protocol for fetching the object (such as http or ftp), host specifies the Internet name of the targeted host, and localinfo is a string (often a file name) passed to the protocol handler on the remote host.

user data constraint

Indicates how data between a client and a web container should be protected. This could be in the form of preventing either tampering with the data or eavesdropping on it.

user (security)

An individual (or application program) identity that has been authenticated. A user can have a set of roles associated with that identity, which entitles the user to access all resources protected by those roles.

valid XML

An XML document that is both well formed and conforms to all the constraints imposed by the specification. For example, it does not contain any tags that are not permitted, and the order of the tags conforms to the specification.

virtual host

Multiple hosts plus domain names mapped to a single IP address.

W3C

World Wide web Consortium. The international body that governs Internet standards (http://www.w3.org/).

WAR

Web Application Archive. A JAR archive that contains a web module.

web application

An application written for the Internet, including those built with Java technologies such as JavaServer Pages and servlets, as well as those built with non-Java technologies such as CGI and Perl.

A "distributed web application" uses J2EE technology written so that it can be deployed in a web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

web component

A component that provides services in response to requests; either a servlet or a JSP page.

web container

A container provided by a web or J2EE server that implements the web component contract of the J2EE architecture. This contract specifies a runtime environment for web components that includes security, concurrency, life-cycle management, transaction, deployment, and other services. A web container provides the same services as a JSP container as well as a federated view of the J2EE platform APIs.

A "distributed web container" can run a web application that executes across multiple Java virtual machines running on the same host or on different hosts.

web module

A deployable unit that consists of one or more web components, other resources, and a web application deployment descriptor contained in a hierarchy of directories and files in a standard web application format.

web server

Software that provides services to access the Internet, an intranet, or an extranet. A web server hosts web sites, provides support for HTTP and other protocols, and executes server-side programs (such as CGI scripts or servlets) that perform certain functions. In the J2EE architecture, a web server provides services to a web container. For example, a web container typically relies on a web server to provide HTTP message handling. The J2EE architecture assumes that a web container is hosted by a web server from the same vendor, so it does not specify the contract between these two entities. A web server can host one or more web containers.

web service

An application that exists in a distributed environment, such as the Internet. A web service accepts a request, performs its function based on the request, and returns a response. The request and the response can be part of the same operation, or they can occur separately, in which case the consumer does not need to wait for a response. Both the request and the response usually take the form of XML, a portable data-interchange format, and are delivered over a wire protocol, such as HTTP.

well-formed XML

An XML document that is syntactically correct. It does not have any angle brackets that are not part of tags, all tags have an ending tag or are themselves self-ending, and all tags are fully nested.

XML

Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the content, data, and text in XML documents.

Index