AutoVue Integration SDK 20.2.1

Technical Guide

ORACLE

September 2012

Table of Contents

1. PREFACE

The *AutoVue Integration SDK Technical Guide* describes the technical details of the AutoVue Integration SDK and how to implement your own integration based on the SDK Framework.

For the most up-to-date version of this document, go to the AutoVue Documentation Web site on the Oracle Technology Network (OTN) at http://www.oracle.com/technetwork/documentation/autovue-091442.html.

1.1 Audience

This document is intended for Oracle partners and third-party developers (such as integrators) who want to implement their own integration with AutoVue based on Web Service technology. If the target system has no Java™ interface (e.g. a .NET or PHP) then using Web Service is one the reliable ways to communicate with this SDK.

Note: If the target system has any Java API to access the documents, it is recommended to use the ISDK Skeleton and integrate it directly to the repository's Java API. The Sample Integration for FileSys package is an example of Java to Java integration of AutoVue SDK. For more information, refer toAppendix B – Sample integration for filesys.

1.2 Related Documents

For more information, see the following documents in the AutoVue Integration SDK library:

- *Overview*
- *Design Guide*
- *Installation and Configuration Guide*
- *User Guide*
- *Acknowledgments*
- *Javadocs*
- *Security Guide*

2. INTRODUCTION

Note: Prior to reading this document, it is strongly recommended that you first familiarize yourself with the AutoVue Integration SDK by reading through the *Overview*, *Design Guide*, *Installation and Configuration Guide*, *Security Guide,* and *User Guide*. These manuals are located in the **/docs** directory and can be accessed from **Quick Start.html** located in the root folder where you installed the AutoVue Integration SDK.

The AutoVue Integration SDK (ISDK) is an interface between Oracle AutoVue and Document Management Systems (DMS)[1]. It enables users to add powerful viewing and markup capabilities to the DMS by interfacing AutoVue with a particular DMS. This interface, or integration process, is composed of several activities: requirements specification, analysis, design, implementation, testing and maintenance. The ISDK provides a framework on top of which you can build your own integration with AutoVue.

The objectives of this document are to help you to understand and familiarize yourself with the ISDK framework, as well as to help you build your own integration of AutoVue. To assist you with the integration, an ISDK skeleton package, Web Services client package and two sample projects (Sample Integration for FileSys and Web Services Sample Server) are included in this ISDK.

3. SYSTEM REQUIREMENTS

For a complete list of system requirements specific to your platform, refer the *Installation and Configuration Guide.*

4. ARCHITECTURE

The following block diagram shows a typical integration between AutoVue and a DMS. The following sections describe how this configuration works.

[1] For the remainder of this document, a DMS/EDM/PDM system is referred to as DMS.

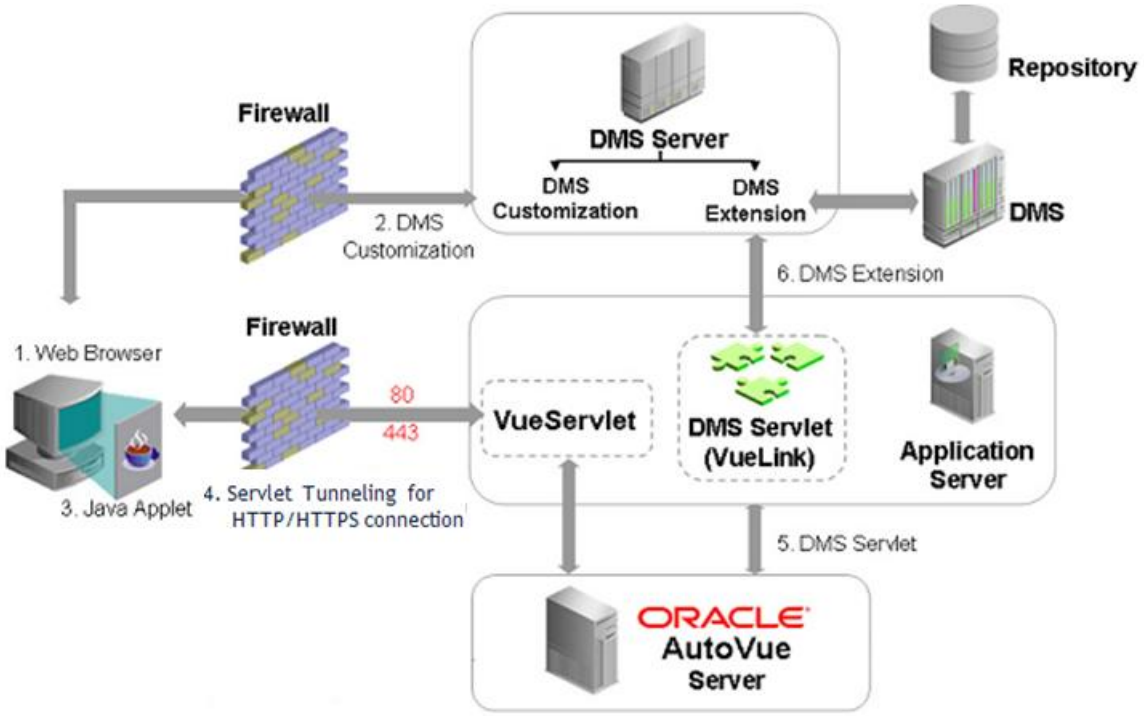


Figure 4-1: Typical configuration for AutoVue Integration with DMS server

4.1 How it Works

As seen in Figure 4-1, the DMS Servlet allows AutoVue server to communicate with a DMS using standard HTTP/HTTPS protocol.

The following is a description of how the DMS Servlet works. Note that the numbered steps refer to the numbers in Figure 4-1.

1. Log into the DMS through a Web browser.

2. With DMS customization in place, you are presented with a link labeled **View** next to each file stored inside DMS. This link allows you to view files in the AutoVue Applet viewer.

3. Click **View.**
   The AutoVue applet launches inside the Web browser window.

4. The AutoVue applet communicates with the AutoVue Server through servlet tunneling for HTTP/HTTPS connection (VueServlet).

5. The AutoVue server then communicates with the DMS servlet using a standard HTTP/HTTPS connection.

6. With the DMS extension installed on the server machine, the DMS Servlet is able to talk to the DMS Server to handle any request made by the AutoVue server, such as file fetching.

If you try to view a composite file (that is, a file having XRefs or font resource files), the DMS Servlet retrieves those files and makes them available to the AutoVue server.

Once the file and all its related XRefs and/or resources are fetched out of the **DMS**, they are processed by the AutoVue server, which renders the file(s) and streams the file to the AutoVue applet for display.

Once the file displays in the AutoVue applet, you can redline it, create new markups, save Markups into the DMS, and open Markups from the DMS.

4.2 Framework

The following block diagram shows the internal structure of a typical integration with a DMS. The framework included in the ISDK provides you with the foundation you need to build your own integration. This framework handles all the plumbing for parsing XML requests received from the AutoVue Server, as well as constructing XML responses sent back to the AutoVue server. This framework is provided so that you do not have to implement your integration from scratch.

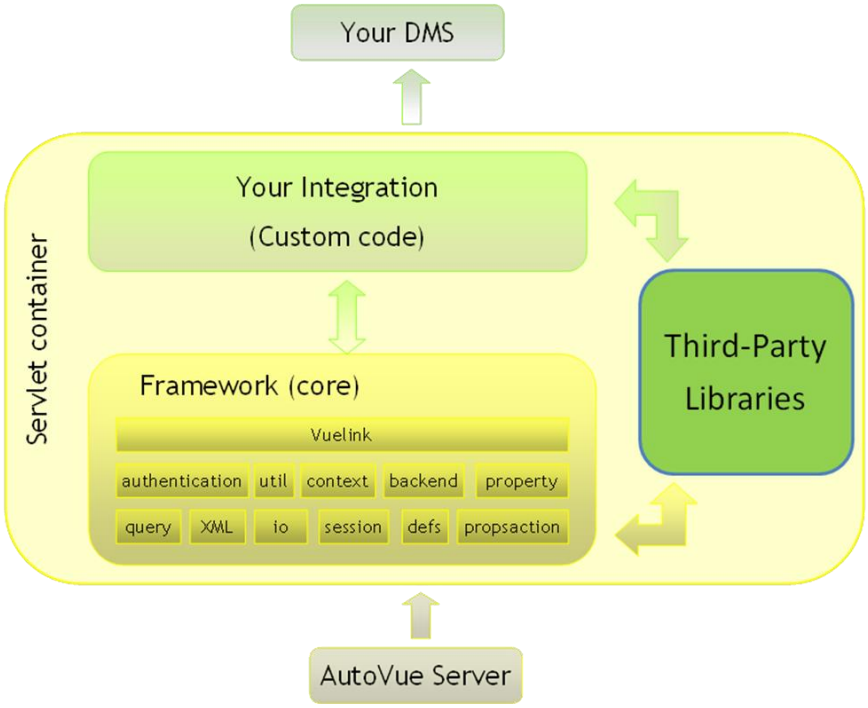


Figure 4-2. Internal Structure of the DMS Servlet

The AutoVue Integration SDK bundles some third-party Java libraries needed by the framework. These libraries are also available for you to call from your own code.

Your integration is responsible for interacting with your DMS. Depending on what type of SDK your DMS provides, such interaction can be as easy as calling your DMS Java libraries.

4.3 Sequence Flow

When a user selects a document to view, the AutoVue server makes several requests to the DMS servlet. The DMS servlet provides a response for each request. The scenario of the exchanges established between the AutoVue server and the DMS servlet are outlined in Figure 4-3 and can be summarized as follows:

- The AutoVue server asks for the PK. This request is handled by VueLink core.
- The AutoVue server asks for the user name (CSI_UserName).
- The AutoVue server asks for the document ID (DocID) of the selected document. This is done through the Open action, which obtains the DocID from the DMS.
- The AutoVue server asks for some properties of the document, such as document name, document size and date of the last modification. The reason is that the AutoVue server maintains a cache of the document and needs to know if it already has the exact save version of the document in its cache. In which case, AutoVue uses the cached copy rather than redownloading the document.
- AutoVue fetches the document through the Download action.

5. INTEGRATION DESIGN

Integration is generally composed of two components: the framework and your specific integration implementation.

The framework is a set of classes that can be used by your integration implementation. It provides you with all the needed functionalities to communicate with the AutoVue server and defines the key concepts to implement your new integration. Understanding these concepts is important for building accurate integrations. The following is a list of the most important classes and packages to consider for your integration design:

- **VueLink servlet**: Base class for your DMS servlet (this is your main class).
- **DMSAction interface**: Represents an execution thread that handles a particular action (such as open, delete, download, save, and so on).
- **DMSGetPropAction interface**: Represents an execution that handles the request for a specific property.
- **DocID interface**: Represents a DMS docID.

All these concepts are explained later in this section. For detailed information on these classes and packages, refer to *API Javadocs* located in the <AutoVue Installation Directory>/docs/javadocs folder.

The second component is your specific integration, which is the code you add on the top of the framework in order to have a working integration. This is the main subject of this documentation.

Your integration must create a DMS servlet that extends the VueLink class and implements some actions and property actions.

Figure 5-1 shows the minimum components you need to add to your integration.

- Your DMS Servlet class (extended from VueLink class)
- Your DocID class (implements DocID interface)
- Your ActionOpen class (implements DMSAction interface)
- Your ActionDownload class (implements DMSAction interface)
- Your ActionGetProperties class (implements DMSAction interface)

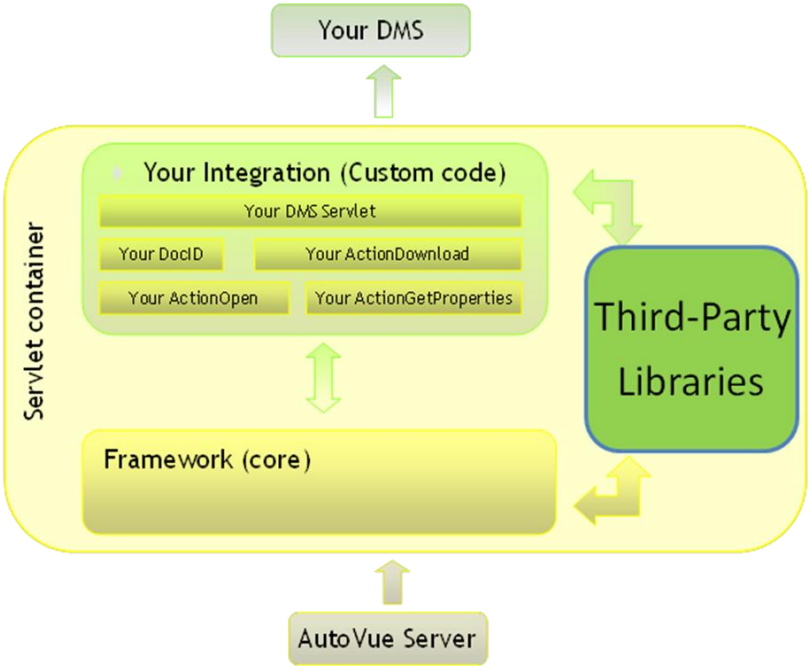


Figure 5-1. Your Integration

5.1 VueLink Class

The framework provides the `com.cimmetry.vuelink.Vuelink` class which is an `HttpServlet` and is configured through the servlet initialization file. The following lists important functionalities that establish the dialog between AutoVue and your integration. Note: Your DMS servlet must extend this class.

- It sets up the log manager for enabling logging at runtime without modifying the application binary (log4j API).
- It registers the DMS Context action and DMS actions classes provided by your integration. Refer to *Javadocs* for more on the **context** package and the **propsactions** package.
- It parses the HTTP request using the `HttpRequestPart` class.
- It uses the `DMSXmlRequest` class, to parse the XML document that contains the actual request. Refer to *Javadocs* for more on the **xml** package.
- It builds a query object (for example, *`DMSQuery`* object) containing all the document information and Properties that your integration needs. Refer to **Javadocs** for more on the **query** package.
- It also constructs some additional `DMSArguments` from an HTTP part or from some special data inside the XML document, such as the file content of a Save request for example. Refer to *Javadocs* for more on the **arguments** package.
- When `DMSQuery` is built, it calls the `execute()` method of the appropriate `DMSAction`, and gets the result back or catches a `VuelinkException` when an error occurs. Refer to *Javadocs* for more information on the `defs` package.
- Finally, it uses the `DMSXmlResponse` class to construct the XML part of the HTTP response before sending it back. Refer to *Javadocs* for more on the xml package**.**

5.2 DMSActions Interface

AutoVue sends requests to your integration and expects responses from it through the framework interface. The framework implements a mechanism that routes requests to your DMS servlet and constructs responses back to AutoVue. The framework provides the `com.cimmetry.vuelink.propsaction.`*`DMSAction`* interface, which represents an execution thread that handles a DMS query. Your integration must define one *`DMSAction`* for each of the following DMS action types:

- `Open`
- `Save`
- `Delete`
- `Download`
- `GetProperties`
- `SetProperties`

5.3 ActionGetProperties Interface

5.3.1 Single Class (Basic Monolithic)

This implementation handles `GetProperties` request using a single class called `ActionGetProperties` that has one monolithic `execute()` method to handle all the properties.

This class implements a DMSAction interface and is usually put in the actions package. You must register this class in the web.xml descriptor file.

This implementation has at least two limitations:

- **Understandability problem:** Too much code in one class, which makes it difficult to understand and to maintain.
- **Extendibility problem:** Since the class performs many functions, it is difficult to extend it with new behavior.

5.3.2 Multiple Classes (Recommended)

One of the main objectives of the AutoVue Integration SDK is that your integration must handle is `GetProperties`. This request covers a wide range of items. The implementation of this request as a single class with one monolithic `execute()` method that handles all the properties has at least two limitations:

- **Understandability problem:** Too much code in one class, which makes it difficult to understand and to maintain.
- **Extendibility problem:** Since the class performs many functions, it is difficult to extend it with new behavior.

One of the main objectives of the AutoVue Integration SDK is to make the framework open and easy to extend. Accordingly, instead of having a single class that takes care of the `GetProperties()` request, individual classes are provided that handle individual properties. Each individual class has its own `execute()` method. When a `GetProperties` request is received, the framework goes through the list of properties. For each property, the framework checks if there is an appropriate action to handle it. If such a class is found, its `execute()` method is called and its return property is saved. Any properties that do not have a specific handler class is passed to a default class.

The framework provides a class for retrieving the individual classes that handle the properties contained in the `GetProperties` request. This class is called

`com.cimmetry.vuelink.propsaction.ActionGetProperties` which implements the `DMSAction` interface. First, this class retrieves the class handler of the requested property, then it calls its `execute()` method, and finally it returns an array of properties containing the response.

Each individual class you provide to handle a specific property must realize the `DMSGetPropAction` interface, then implement the `execute()` method. The `execute()` method must make the request to the DMS, get the response, and then return it as an array of properties.

The `GetPropAction` retrieves each property action using the init-parameters mechanism. If the class is not registered, the framework looks for a property action defined with a default name `GetProp<prop name>` in the DMS servlet location. If no class is found, the `GetPropDefault` class is called. In this framework, the `GetPropDefault` class is treated as any other property action. If `GetPropDefault` is not found, an exception is thrown. Also, if the requested property is not handled in the `GetPropDefault` class, an exception must be thrown.

5.4 DocID Interface

The DocID in this framework always refers uniquely to a specific document or file in your DMS. You must be able to ask for the contents of the file by its DocID, and get a uniquely-identified result. In a typical DMS, this can be a combination of the object ID of the document that contains the file along with library name where this document is stored.

Chapter 6 describes the minimum set of steps you need to follow in order to implement the viewing functionality of files stored in your DMS using AutoVue.

6. IMPLEMENTING FILE VIEW FUNCTIONALITY IN YOUR DMS

This chapter describes the minimum steps required to add file viewing capabilities using AutoVue with your DMS. Once you have completed these steps, proceed to Chapter 7 for information on adding functionality such as searching the DMS, browsing the DMS, creating markups, performing conversions, and so on.

As mentioned in the *Overview* document, the AutoVue Integration SDK bundles a sample integration called *Sample Integration for Filesys DMS*. The purpose of this sample is to guide you in understanding the integration framework. This sample also acts as a good starting point for building your own integration between AutoVue and your DMS.

To learn more about the sample integration, refer to the appendix in this document.

The following sections describe the steps you need to follow in order to implement basic file viewing functionality using AutoVue and your DMS. Each step includes an excerpt of code to show how the *Sample Integration for Filesys DMS* is implemented. It helps you to understand the sample integration. But for your own implementation of the Integration SDK, it is highly recommended to follow the coding style in the Integration SDK Skeleton.

6.1 Step 1: Creating Your Main DMS Servlet by Extending the VueLink Class

As discussed in Chapter 5, Integration Design, the framework provides the `VueLink` base class which is a servlet implemented in the `com.cimmetry.vuelink` package of the SDK. The `VueLink` base class provides all the needed services to handle the requests and responses from the DMS and AutoVue Server. In most cases when implementing your DMS servlet, just deriving a new class from `VueLink` class is sufficient.

The following excerpt of code shows the implementation of the `FilesysVuelink` servlet in the `com.cimmetry.vuelink.filesys` package.

```
package com.cimmetry.vuelink.filesys;

import com.cimmetry.vuelink.*;
…

public class FilesysVuelink extends Vuelink {
…
}
```

For example, you can override the servlet's `init()` method to perform additional initialization or override the `doGet()` method to return your own HTML code.

6.2 Step 2: Defining Your Unique Document Identifier by Implementing DocID Interface

AutoVue and DMS exchange several types of files, such as the base document, XRefs, markups, renditions, and so on. To keep the correct mapping between the files and their original copies in the DMS backend system, an identification mechanism is needed. For this purpose, the framework provides us with the `DocID` interface. You must implement your own class based on the `DocID` interface and it should be convertible to a string.

Take note of the different concepts of the unique document identifier in DMS backend system and the unique document identifier (DocID) in the Integration SDK. Usually, DocID encapsulates the unique document identifier in DMS backend system and adds more attributes.

```
package com.cimmetry.vuelink.defs;

/**  */
public abstract class DocID implements
java.io.Serializable {
}
```

In the Sample Integration for Filesys DMS, the `FilesysDMSDocID` class is coded in the backend package (`com.cimmetry.vuelink.filesys.backend`). The `FilesysDMSDocID` class extends the *`DocID`* abstract class and builds a unique identifier for each file.

Note: It is helpful to think of the backend class as a wrapper around your DMS API. Implementing the `DMSBackend` interface is optional. To learn more about the backend package, refer to the Appendix B - Sample Integration for Filesys**.**
Inside the Filesys DMS backend system, the relative path for each file to the repository root folder is unique and can be used as a document identifier. When constructing a `FilesysDMSDocID` object, the `m_id` member is set to the relative path of a file, for example, /2D/AutoCAD.dwg/AutoCAD.dwg(1)/AutoCAD.dwg which is relative to the RootDir defined in web.xml.

```
package com.cimmetry.vuelink.filesys.backend;

/**  */
public class FilesysDMSDocID extends DocID implements DMSDefs{
  …
```

Note: It is recommended that the DocID size should be less than 2KB and should not contain a variable component.

6.3 Step 3: Creating a GetProperty action to return User Name

The AutoVue server sends a `GetProperties` request asking for `CSI_UserName`. The implementation of the class is responsible for returning it. It is similar to the implementation of `CSI_DocName` described in 6.6Step 4: Creating a Get Property Action to Return Document Name.

6.4 Step 4: Creating a class to implement DMSBackend interface

There is a `DMSBackend` interface provided by the VueLink core that has a `connect()` API that must be implemented. This implementation class is needed in order to avoid a deployment warning being thrown by the `GenericContext` class. At the beginning stage of your integration development, you can provide an empty implementation for the `connect()` method in your implementation class and register your `DMSBackend` implementation class in the web.xml file.

During the development phase, you can also include methods that handle communication with the backend DMS in your `DMSBackend` implementation class.

After your create your own context class as described in Creating Your Context, you must overwrite the `getBackendAPI()` method of the `GenericContext` class in order to retrieve your own `DMSBackend` implementation class. You must also overwrite the `getBackendSession()` method of the `GenericContext` class in order to use the `connect()` method of your `DMSBackend` implementation class. Doing overwriting this method allows AutoVue to re-use existing use sessions with your backend DMS system. For information on how to implement these classes, refer to the following ISDK Skeleton implementation classes:
`com.mycompany.autovueconnector.backend.DMSBackendImp`
`com.mycompany.autovueconnector.session.DMSBackendSessionImp`

6.5 Step 5: Creating an Open Action class that returns your DocID

When you select a document to view, the first request the AutoVue server sends is an open request asking for the DocID of this document. You must create the `ActionOpen` class in your integration by implementing the `DMSAction` interface to handle the open request. The framework automatically finds your class that handles this request and executes it. You must also implement the `execute()` method which returns the unique DocID for the document being viewed.

Usually the unique document identifier for the DMS backend system can be retrieved from the Original URL of the open request sent by the AutoVue server. However, your Integration SDK might also need to call DMS backend system's API to get the unique document identifier or other document attributes in order to construct your Integration SDK's DocID.

In the Sample Integration for Filesys DMS, as shown in the following excerpt of code, the `ActionOpen` class realizes the `DMSAction` interface and implements the `execute()` method. The `execute()` method returns the DocID obtained from `openFile()` method of the DMSBackend class that retrieves and constructs `FilesysDMSDocID` using relative file path and other attributes. Although implementing the `DMSBackend` interface is optional, the Sample Integration for Filesys implements this interface as an example to show how you can use it in your own integration.

```
package com.cimmetry.vuelink.filesys.actions;
…
public class ActionOpen implements DMSAction<FilesysContext>, DMSDefs{
   …
   public Object execute(final FilesysContext  context,
                  final DMSSession  session,
                  final DMSQuery   query,
                  final DMSArgument[] args
                  ) throws VuelinkException {
      …
      // open action returns the DocID
      DocID docID = context.getBackendAPI().openFile
}
```

If you do not place your DMSAction classes in the same package as your DMS Servlet, the framework retrieves the `ActionOpen` class from the web.xml descriptor file. In this case, each action class should be registered in this file as an init-parameter. The `ActionOpen` class has `dms.action.`**`Open`** as a parameter name and its value should be a fully qualified class name. In the case of the Sample Integration for Filesys DMS, this is `com.cimmetry.vuelink.filesys.actions.`**`ActionOpen`** as the parameter value. The `FilesysVulinkServlet` uses this init parameter to locate, register, and instantiate the `ActionOpen` class.

```
<init-param>
  <param-name>dms.action.Open</param-name>
  <param-
value>com.cimmetry.vuelink.filesys.actions.Action
```

For more information on the behavior of `ActionOpen` class, we advise you to (1) closely examine the source code and (2) run the Filesys project in IDE in debug mode, set breakpoint as shown in the following figure, and then follow the execution step by step. This will give you more insight into the behavior of this class.

```
53          // The action name in the query must be "open"
54          if (!"open".equalsIgnoreCase(query.getActionName())) {
55              throw new VuelinkException(DMS_ERROR_CODE_UNKNOWN_ERROR,
56                                         "Invalid action name within query:
57                                         query.getActionName());
```

In the Sample Integration for Filesys DMS, the `ActionOpen` class relies on the `openFile` method of the `FilesysDMSBackendImp` class to obtain the DocID of a file. This method has two parameters:

- The session information to connect to the backend.
- The information needed to open the file (for example, Filesys DMS backend system and name of the file)

```
public DocID openFile(DMSBackendSession session, Hashtable<String, String>
_params)throws VuelinkException {
```

This method returns the DocID of the file for Filesys. If it fails, it throws a Vuelink exception.

The `openFile` method parses the original URL available from the `open` request to get the unique document identifier (the relative file path), version and other parameters necessary to construct the DocID for Filesys DMS. Then it builds the `FilesysDMSDocID` to return back to the `ActionOpen` class. There is additional code in `openFile` method to construct data members that supports OEVF, versioning and rendition. The concept of OEVF, versioning, and rendition are discussed later.

Note: When the number of the version is not provided, the Filesys DMS system returns the latest version of this document.

```
package com.cimmetry.vuelink.filesys.backend;
…
public DocID openFile(DMSBackendSession session, Hashtable<String, String>
_params) throws VuelinkException { // get parameters
        Hashtable<String, String> params = _params;
        FilesysDMSDocID docID = null;

        String oevf = "oevf://";
        String origURL = params.get("origURL");
        String version = params.get("Version");
        …
        String relPath = null; // relative file path
        String aID = DMSUtil.getAssetID(origURL); // aID and wID are for OEVF
        String wID = DMSUtil.getWorkflowID(origURL);

       if( origURL.startsWith(oevf)) {
          …
       }else{
          relPath = origURL;
       }
```

For more information, examine the code and use the debugger to learn more about the actual behavior of this class.

6.6 Step 4: Creating a Get Property Action to Return Document Name

AutoVue sends several `GetProperties` requests to know if it already has the most recent copy of the document in its cache. The first request sent is for the name of the file identified by a DocID. This is done through the CSI_DocName property.

Note: The string value returned for CSI_DocName should include a file extension.

To handle get property requests, you have two options: you can either define a single class called `ActionGetProperties` that implements DMSAction or you can have separate classes that implement the `DMSGetPropAction` interface. The second approach is recommended because it reduces code complexity in a single class and improves readability, but each class needs to be registered in web.xml descriptor file if it is not named as "GetProp<prop name>" and located in the same package as your DMS servlet class.

Notice that we need to pass in a type parameter (any context that implements the `DMSContext` interface or extends the `GenericContext` class) when using `DMSAction` and `DMSGetPropAction` interface, before your Integration SDK implements your own Context class as described in 7.12 Creating Your Context, you can use `GenericContext` instead.

If you choose the first approach, use the following excerpt of code to define your own `ActionGetProperties` class. You can retrieve the list of properties from the query object passed as a parameter to the `execute()` method. You can then loop through the properties list and retrieve its value from your DMS. For more information refer to 5.3: ActionGetProperties Interface.

Usually, the `ActionGetProperties` class is put in the same `actions` package as other action classes. Note that you must register this class in the web.xml descriptor file as long as it is not located in the same package as your DMS servlet class.

```
package com.myisdk.actions;

/**  */

public class ActionGetProperties implements DMSAction<GenericContext>, DMSDefs{
   …
   public Object execute(final FilesysContext  context,
            final DMSSession  session,
            final DMSQuery    query,
            final DMSArgument[] args
            ) throws VuelinkException {
…       Property[] props = query.getProperties();
        String propName = props[i].getName();

        // GetProperty action returns attribute values
        If (propName.equals(DMSProperty.CSI_DocName) {
                … // return doc name
        } else if(propName.equals(DMSProperty.CSI_DocNameIsMultiContent) {
                … // return is multi content
        } else if(propName.equals(DMSProperty.CSI_DocDateLastModified) {
                … // return is date last modified
        } else if(propName.equals(DMSProperty.CSI_DocSize) {
                … // return is doc size
        }
        …
```

For the second approach, as demonstrated in the Sample Integration for Filesys DMS, separate classes are used to implement the `DMSGetPropAction` interface and they are located in `propactions` package. Additionally, a `GetPropDefault` class is implemented to process properties that are not handled by separate classes.

The following excerpt of code illustrates the implementation of the `GetPropCSI_DocName` class in the Sample Integration for Filesys DMS. It gets the document name from the `GetFilesysProperty` class, and then returns it to the AutoVue server.

```
package com.cimmetry.vuelink.filesys.propactions;

/**  */
public class GetPropCSI_DocName extends GetFilesysProperty
                        implements DMSGetPropAction<FilesysContext> {
        …
        public DMSProperty execute(FilesysContext context, DMSSession session,
                        DMSQuery query, DMSArgument[] args, Property property)
                        throws VuelinkException {
                final FilesysDMSDocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
                …
                DMSProperty attrs = getAttrs(context.getBackendAPI(),
                        context.getBackendSession(session, query),query, docID);

                DMSProperty retProp = new DMSProperty(Property.CSI_DocName,
                        attrs.getFirstChildWithName("DocName").getValue());

                m_logger.info("Got doc name: " + (String)attrs.getFirstChildValue("DocName"));
                return retProp;
        }
}
```

As explained in Chapter 5.3 ActionGetProperties Interface, each individual property class realizes the framework interface `DMSGetPropAction` by implementing the `execute`() method. Given a DocID, the `getAttrs` method returns a Hashtable of attributes of the corresponding document. One of these attributes is the document name, which is returned as a `DMSProperty` object. Refer to the Appendix B for information on implementing the `GetFilesysProperty` class.

To allow the framework to locate the register and instantiate the `GetPropCSI_DocName`, we must register class in the web.xml file. As illustrated in the following code, this class is registered with the parameter name `dms.getprops.CSI_DocName` and the parameter value `com.cimmetry.vuelink.filesys.propactions.GetPropCSI_DocName`.

```
<init-param>
 <param-name>dms.getprops.CSI_DocName</param-name>
 <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_DocName</param-
value>
</init-param>
```

Note:
For this property class, we have chosen a different name from the one suggested by the framework. The default name has the format `GetProp<property name>`. Note that in this case we decided to name the class `GetPropCSI_DocName`.

For more information, examine the code and use the debugger to learn more about the behavior of this class.

6.7 Step 5: Creating a GetProperty action to return Document Date Last Modified and Size

Note: This is an important step and should not be skipped.

The AutoVue server sends a second `GetProperties` request asking for the date of the last modification and the size of the document (for example, `CSI_DocDateLastModified` and `CSI_DocSize` properties). The implementation of the class responsible for returning these properties is very similar to the `CSI_DocName` presented in section 6.3. Create an Open Action class that returns your DocID.

Refer to section 6.4. Create a Get Property action to return Document Name for information on how to define your own ActionGetProperties class.

6.8 Step 6: Creating a Download action to return Document Content

The AutoVue server checks its cache to see whether it has a more recent copy of the document by comparing its time stamp against the properties retrieved in the previous steps. If the copy in the cache is older than the copy in the DMS, the AutoVue server tries to fetch the document from the DMS backend system by calling the Download Action.
You must create the `ActionDownload` class in your integration by implementing `DMSAction` interface. You must also implement the `execute()` method which returns `FileInputStream` object. The framework automatically streams the file content back to the AutoVue server.

The following excerpt of code from the Sample Integration for Filesys DMS presents the implementation of the `ActionDownload` class. Note that like any action class, this class realizes the `DMSAction` class and implements the `execute()` method. Using the DocID of the document, the `execute()` method calls the `checkout()` method, downloads the file as `FileInputStream` object, and then returns the stream. The rest is done by the Vuelink class before passing it back to the AutoVue Server. If the download operation fails, a `VuelinkException` is thrown.

```
package com.cimmetry.vuelink.filesys.actions;

/**  */
public class ActionDownload implements DMSAction<FilesysContext>, DMSDefs{
        …
        public Object execute(final FilesysContext  context,
              final DMSSession  session,
              final DMSQuery   query,
              final DMSArgument[] args
              ) throws VuelinkException {
        …
        final DocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
        // checkout the instance file of the document
        final FileInputStream doc =
…
}
```

The action download is registered in the web.xml file, as shown in the following excerpt of code.

```
 <init-param>
   <param-name>dms.action.Download</param-name>
   <param-
value>com.cimmetry.vuelink.filesys.actions.ActionDownload</
param-value>
```

For more information, examine the code and use the debugger to learn more about the behavior of this class.

This `checkout()` method gets a copy of a file from the DMS backend system by invoking the Filesys DMS `getFile()` method. It has two parameters:

- The session information to connect to the DMS
- The DocID of the file to be downloaded

```
package com.cimmetry.vuelink.filesys.backend;
…
public FileInputStream checkout(DMSBackendSession session, DocID docID) {
        DocInfo fsDocID = buildDocInfo(session,docID);
        FileInputStream fis = null;
        try {
                fis = new FileInputStream(m_filesysInfo.getFile(fsDocID));
        } catch (FileNotFoundException e) {
                System.out.println("File not found" + fsDocID.getName());
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e);
        }
        return fis;
```

6.9 Step 7: Implementing Remaining Actions and Registering in web.xml

Implement the DMSAction interface to create a skeleton for the following action classes in your integration:

- ActionDelete
- ActionSave
- ActionSetProperties

For each action, you must implement the `execute()` method. At this point, you can leave the `execute()` method empty as it does not serve a function at the moment. Implementing these actions is optional and is explained in more detail in the next chapter. For example, if you plan to add delete functionality to your integration, you can refer to section 7.11 Implementing File Delete Action.

Review the following code excerpt:

```
public class ActionDelete
        implements DMSAction<FilesysContext>, DMSDefs {

        public Object execute(final FilesysContext  context,
                final DMSSession  session,
                final DMSQuery   query,
                final DMSArgument[] args
                ) throws VuelinkException {
                  // TODO…

        }
}
```

As with `ActionOpen` and `ActionDelete`, if you place `ActionDelete/ActionSave/ ActionSetProperties` in the same package as your DMS Servlet, the framework automatically finds them. Otherwise, you need to register them in web.xml. In the case of the sample integration for Filesys, these actions are under the **actions** package and therefore has to be registered in web.xml.

```
 <init-param>
   <param-name>dms.action.Delete</param-name>
   <param-
 value>com.cimmetry.vuelink.filesys.actions.ActionDelete
 </param-value>
 </init-param>
```

7. IMPLEMENTING ADVANCED INTEGRATION FUNCTIONALITY IN YOUR DMS

This section describes optional functionality that you can choose to add to your integration. Each step includes an excerpt of code to show how the Sample Integration for Filesys DMS is implemented. It is helpful to understand the sample integration. However, for your own implementation of Integration SDK, it is highly recommended to follow the coding style in the Integration SDK Skeleton. For example, the Integration SDK Skeleton makes it a standard that all property retrieving methods in the DMSBackendImp class return DMSProperty object instead of the different object types returned by the FilesysDMSBackendImp class. So that, in most cases, the property action classes in Skeleton do not need to reprocess the returned objects from methods in DMSBackendImp class again.

Note: The following sections assume that you have already implemented file view functionality in your DMS as outlined in previous chapter.

7.1 Handling Document Attributes

One single `GetProperties` request from AutoVue server can ask for multiple properties of a document. As a result, it is recommended to get the whole set of attributes from DMS the first time they are needed, and then save it to be reused for getting other properties in that request.

In the Integration SDK Skeleton, this functionality is included inside the `GetPropDefault` class. The `listAllProperties()` method of the backend implement class is responsible for retrieving the properties for the first time and then saving it to the query object for one request.

```
package com.mycompany.autovueconnector.propactions;

public class GetPropDefault implements DMSGetPropAction<DMSContextImp>, DMSDefs {
   …
   public DMSProperty execute(…) throws VuelinkException{
        …
        DMSProperty attrs = (DMSProperty)query.getQueryData("attrs");
        ISDKDocID docID = new ISDKDocID().String2DocID(query.getDocID());

        if (attrs == null ){
             attrs = be.listAllProperties(beSession, docID); //retrieve for the first time
             if(attrs != null ){
                  query.setQueryData("attrs", attrs); //save to be reused
             }
        }
       …
   }
}
```

```
package com.mycompany.autovueconnector.backend;

public DMSProperty listAllProperties(
                DMSBackendSessionImp beSession,
                DocID docID
        ) throws Exception {

        Vector<DMSProperty> props =  new Vector<DMSProperty>();
        // TODO Retrieve all properties's name and value pair;
        // TODO Construct DMSProperty object for each property like
        //          new DMSProperty(name, value);
        //       For example,
        //          new DMSProperty(DMSProperty.CSI_DocName, docName);
        // TODO Add these DMSProperty objects to the vector "props"

        if(props == null || props.isEmpty())
                return null;

        // Need to pass an array (of DMSProperty) for the second parameter when
        // constructing the return DMSProperty
        DMSProperty [] aPL = new DMSProperty[1];
        return new DMSProperty(DMSProperty.CSI_ListAllProperties, props.toArray(aPL));
}
```

In the Sample Integration for Filesys, a separate `GetFilesysProperty` class is implemented to fulfill the same task.

```
package com.cimmetry.vuelink.filesys.propactions;

/**  */
public class GetFilesysProperty implements DMSDefs {
    …
   protected DMSProperty getAttrs(final FilesysDMSBackend be,DMSBackendSession beSession,
                    final DMSQuery query, DocID docID) throws VuelinkException {
        …
        DMSProperty attrs = (DMSProperty) query.getQueryData("attrs");
        if (attrs == null) {
                attrs = be.getAttributes(beSession, docID);
                m_logger.info("got document attributes " + attrs);
                query.setQueryData("attrs", attrs);
        }
        return attrs;
    }
}
```

Note that this class is not a property class and does not realize the `DMSProperty` interface or implement the `execute()` method. As a result, we do not need to register it in the web.xml file. This class supports all the property classes that use the document attributes. This class gets the attributes from the DMS backend system by means of the `getAttributes()` method of the Filesys DMS backend class (for example, `FilesysDMS` class). One `GetProperties` request from AutoVue server can ask for multiple properties, thus `GetFilesysProperty` class saves the retrieved attributes from the DMS to be reused for getting multiple properties in one request.

The `getAttributes()` method of the Filesys DMS backend class first asks the Filesys DMS system to give it a Hashtable<String, String> that stores the name and value pairs of a list of attributes. As shown in the following code, this is done by calling the *m_filesysInfo*`.getAttributes()` method by passing the DocID of the document.

The list of attributes retrieved by *m_filesysInfo*.getAttributes() method includes:

- **DocName:** The name of the file. The value is a String.
- **DateLastModified:** The date the file was last modified. The value is as a java.util.Date object.
- **DocSize:** The size of the file.
- **DocFormat:** Document format (for example, "document" or "folder") . The value is an Integer.
- **Version:** The version number of a document. The value is a String.
- **VersionsNumber:** The number of versions of a document. The value is a String.
- **path:** The absolute path for the file in Filesys DMS. The value is a String.

Then it builds a DMSProperty class for each attribute and puts them into a Vector<DMSProperty> object.

Finally, it converts the vector to an array and wrap it as a DMSProperty object to return.

```
package com.cimmetry.vuelink.filesys.backend;
…
public DMSProperty getAttributes(DMSBackendSession session, DocID docID) {
        DocInfo fsDocID = buildDocInfo(session,docID);
        Vector<DMSProperty> result = new Vector<DMSProperty>();
        try{
                Hashtable<String,String> attrs =
                    m_filesysInfo.getAttributes(fsDocID);
                Enumeration<String> keys = attrs.keys();
                while (keys.hasMoreElements()) {
                    String key = keys.nextElement();
                    String value = attrs.get(key);
                    if (value != null && value.split(";").length > 1) {
                        // multi value
                        result.add(new DMSProperty(key,value.split(";")));
                    }else {
                        result.add(new DMSProperty(key,value));   //single value
                    }
                }

        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e);
        }
        DMSProperty[] answer = new DMSProperty[0];
        answer = result.toArray(answer);
                return new DMSProperty(DMSProperty.CSI_ListAllProperties,answer);
    }
```

7.2 Returning External References (XRefs)

Chapter 6 discussed the case of viewing a simple document composed of a single file. Documents, however, are often compound and may have many associated files or External Reference files (XRefs). In this case, the AutoVue server asks for XRefs by passing CSI_XREFS within the GetProperties request. The response to this request is provided by GetCSI_XREFS, the XRefs property class.

In the Sample Integration for Filesys DMS, since `GetCSI_XREFS` is a property class it realizes the `DMSGetPropAction` and implements the `execute()` method. The following code shows all the imported classes from the AutoVue Integration SDK framework. All these classes are referenced in the `execute()` method parameters. Refer to the Appendix B for more information on these parameters.

```
package com.cimmetry.vuelink.filesys.propactions;
…
import com.cimmetry.vuelink.defs.DocID;
import com.cimmetry.vuelink.defs.VuelinkException;
import com.cimmetry.vuelink.filesys.FilesysContext;
import com.cimmetry.vuelink.filesys.backend.FilesysDMSBackend;
import com.cimmetry.vuelink.filesys.backend.FilesysDMSDocID;
import com.cimmetry.vuelink.property.Property;
import com.cimmetry.vuelink.propsaction.DMSGetPropAction;
import com.cimmetry.vuelink.propsaction.DMSProperty;
import com.cimmetry.vuelink.propsaction.arguments.DMSArgument;
import com.cimmetry.vuelink.query.DMSQuery;
import com.cimmetry.vuelink.session.DMSBackendSession;
import com.cimmetry.vuelink.session.DMSSession;

public class GetPropCSI_XREFS implements DMSGetPropAction {
```

The following excerpt of code shows how the `execute()` method builds a `CSI_XREFS` DMSProperty from the list of XRef files returned by calling the `dmsListXRefs` method of the FilesysDMS backend class. The `CSI_XREFS` DMSProperty is returned to the VueLink servlet which provides the response to the AutoVue server.

```
public DMSProperty execute(FilesysContext context, DMSSession session,
        DMSQuery query, DMSArgument[] args, Property property)
        throws VuelinkException {

    final DocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
    DMSProperty retProp = new DMSProperty(Property.CSI_XREFS,
          buildXREFSProperty(((FilesysDMSBackend)context.getBackendAPI()),
    context.getBackendSession(session, query), docID));
    m_logger.debug("got the xrefs property: " + retProp);
    return retProp;
}
```

The `dmsListXRefs()` method of the Filesys DMS backend class talks to the Filesys DMS backend system and gets the list of the XRef file as vector. For each element of the vector, it builds a `DMSProperty` as specified in the CORE API specification.

The difference between the Integration SDK skeleton and the Sample Integration for Filesys DMS is that he `dmsListXRefs()` method of the Skeleton DMS backend class returns the final DMSProperty object directly instead of returning a list of DocID and construct in the `GetCSI_XREFS` class.

```
private Property[] buildXREFSProperty(FilesysDMSBackend be, DMSBackendSession beSession,
                DocID docID) {

        // Gets list of xrefs from DMS
        Vector<DocID> xrefsDocIds = be.dmsListXRefs(beSession, docID);
        DMSProperty[] xrefs = new DMSProperty[xrefsDocIds.size()];

        for (int i = 0; i < xrefsDocIds.size(); i++) {
             DMSProperty xrefProp[] = new DMSProperty[2];
             xrefProp[0] = new DMSProperty(Property.CSI_DocID,
                        ((FilesysDMSDocID)xrefsDocIds.get(i)).DocID2String());

             xrefProp[1] = new DMSProperty(DMSProperty.PROP_NAME,
                        ((FilesysDMSDocID)(xrefsDocIds.get(i))).getName());
             xrefs[i] = new DMSProperty(Property.PROP_XREF,xrefProp);
        }
        m_logger.debug("got the list of xrefs : " + xrefs);
        return xrefs;
}
```

The `GetPropCSI_XREFS` is registered in the web.xml file as shown in the following code excerpt.

```
 <init-param>
   <param-name>dms.getprops.CSI_XREFS</param-name>
   <param-
 value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_XREFS
 </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this method.

This method asks the Filesys DMS for the list of XRefs associated with a given document by providing its `DocID`. After it receives the vector of XRef files, it builds a DocID for each XRef. Finally, it returns the list of DocIDs as a vector.

For more information, examine the code and use the debugger to learn more about the behavior of this method.

```
public Vector<DocID> dmsListXRefs(DMSBackendSession session, DocID docID)
{
        FilesysDMSDocID fsDocID = (FilesysDMSDocID) docID;
        …
        xrefsInfos = m_filesysInfo.listXRefs(fsDocID);

        xrefs = new Vector<DocID>();
        …
        for (int i = 0 ; i < xrefsInfos.size() ; ++i) {
                xrefs.add(new
FilesysDMSDocID((DocInfo)xrefsInfos.get(i)));
        }
        return xrefs;
}
```

7.3 Handling Markups

When users view a markup, the AutoVue server asks the DMS for the list of markups associated with the document. The server does so by sending a `GetProperties` request for the `CSI_Markups` property. The `GetPropCSI_Markups` class handles the response for the request. The response consists of two parts: a GUI response and a Markup response.

7.3.1 GUI Response

When you develop an integration based on ISDK you can control some aspects of the AutoVue UI; specifically, the Markup Open and Save dialogs. AutoVue constructs UI elements in these dialogs based on your response to the Markup GUI.
The GUI part is composed of three sections: Display Options, Edit, and Display.

The Display Options specifies whether or not users are allowed to perform particular operations on markups. In the Sample Integration for Filesys, the following excerpt of code builds several properties and sets their value to `true` or `false`. Each of these properties is dedicated to a particular operation. For instance, in the property `AllowDelete` (which allows users to delete markups), Markups is set to `true`. The last line of the code shows how all the properties are grouped in a single property labeled `DisplayOptions`.

```
package com.cimmetry.vuelink.filesys.propactions;

public class GetPropCSI_Markups extends GetFilesysProperty implements
              DMSGetPropAction<FilesysContext> {
    …
    private DMSProperty[] buildMarkupGui(FilesysDMSBackend be,
              DMSBackendSession beSession, DocID docID) {

        DMSProperty guiProps[] = new DMSProperty[3];
        DMSProperty DispOptArr[] = new DMSProperty[7];
        DispOptArr[0] = new DMSProperty("AllowDelete","true");
        DispOptArr[1] = new DMSProperty("ShowPreviousVersions","true");
        DispOptArr[2] = new DMSProperty("AllowNew","true");
        DispOptArr[3] = new DMSProperty("AllowImport","false");
        DispOptArr[4] = new DMSProperty("AllowExport","false");
        DispOptArr[5] = new DMSProperty("AllowNewLayers","false");
        DispOptArr[6] = new DMSProperty("AllowModifyLayers","false");

        guiProps[0] = new DMSProperty("DisplayOptions", DispOptArr);
```

The Edit section specifies the GUI elements we want to use to populate the Save Markup dialog. The Save Markup dialog contains two GUI elements: an edit box and a drop-down list.

For example, if you want AutoVue to display the Save Markup File As dialog as shown in Figure 7-1, you must define the input box and list UI elements.

The label of the edit box is **Name** and its control ID is `CSI_DocName`.
The label of the drop-down list is **Markup Type** and its control ID is `CSI_MarkupType`.
The drop-down list contains three selections: **normal, master** and **consolidated**, with the

default value set to **normal**. AutoVue sets the default value to the one specified in `GUIElementCombo` class.

The label of the second drop-down list is **Read-Only** and its control ID is `CSI_Doc_ReadOnly`. The drop-down list contains two options: **false** (default value) and **true**.

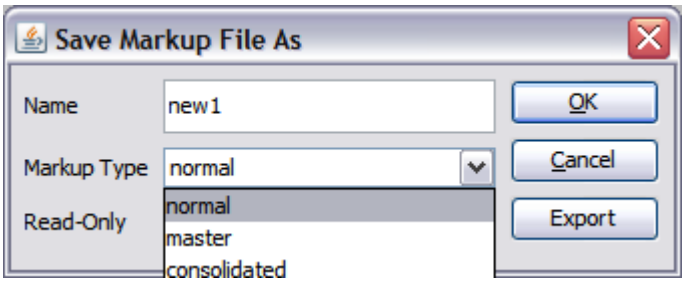


Figure 7-1. Save Markup dialog

In the Sample Integration for Filesys, the following excerpt of code builds the `GUIElementCombo` property, which specifies a drop-down list that contains three selections: **normal**, **master** and **consolidated**. The default selection is set to **normal**. This is done by passing **normal** as the third parameter when constructing `GUIElementCombo` class. Note that the last line of code attaches the `GUIElementCombo` property in a `DMSProperty` labeled `DMSProperty.PROP_GUI_EDIT`.

The code for building the `GUIElementCombo` property for Ready-Only is similar and is described in detail in Implementing Read-Only Markups.

```
String comboVals[] = new String[3];
comboVals[0] = DMSProperty.CSI_MarkupType_Normal;
comboVals[1] = DMSProperty.CSI_MarkupType_Master;
comboVals[2] = DMSProperty.CSI_MarkupType_Consolidated;
EditArr[1] = new GUIElementCombo(DMSProperty.CSI_MarkupType, "Markup Type",
DMSProperty.CSI_MarkupType_Normal, comboVals, false);
…
guiProps[2] = new DMSProperty(DMSProperty.PROP_GUI_EDIT,EditArr);
```

The Display section specifies properties to be displayed in tabular format inside the Markup Files dialog when the Open Markups action is selected from the AutoVue GUI.

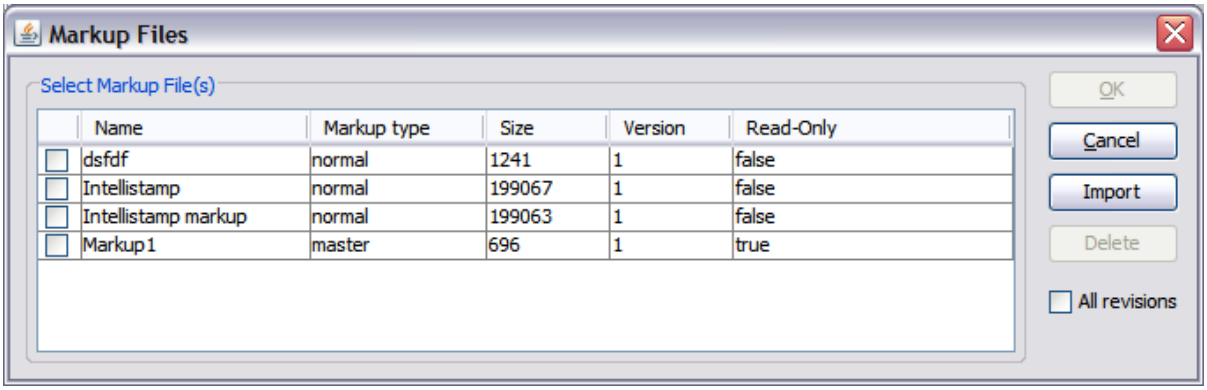


Figure 7-2. Markup files dialog

In the Sample Integration for Filesys, the following code defines five GUI elements that compose the Markup Files dialog: document name, markup type, document size, the version of the document, and whether the markup is read-only or can be modified. Each of these elements is encapsulated as a `DMSProperty` labeled, `CSI_DocName`, `CSI_MarkupType`, `CSI_DocSize`, `CSI_Version`, and `Read-Only`. Finally all these properties are attached to a `DMSProperty.PROP_GUI_DISPLAY` object.

```
DMSProperty DispArr[] = new DMSProperty[5];
DispArr[0] = new DMSProperty(Property.CSI_DocName,"20");
DispArr[1] = new DMSProperty(Property.CSI_MarkupType,"15");
DispArr[2] = new DMSProperty(Property.CSI_DocSize,"10");
DispArr[3] = new DMSProperty(Property.CSI_Version,"10");
DispArr[4] = new DMSProperty("Read-Only","6");

guiProps[1] = new DMSProperty(DMSProperty.PROP_GUI_DISPLAY, DispArr);
```

Note: All GUI properties (for example, *DisplayOptions*, *Display and Edit*) must be attached to a `DMSProperty` object with *PROP_GUI* identification.

7.3.2 Markup Response

The Markup response specifies the list of markups associated with the current document. Each element of the list must be encapsulated in a Markup `DMSProperty`. For more information, refer to GetPropCSI_Markups.java class found inside Filesys package for the actual format of the Markup response. The list of markups is returned by the `dmsListMarkups` method of the `FilesysDMSBackendImp` class.

In the Sample Integration for Filesys, the following code excerpt of the Markup response shows all the required information for each markup. This information includes the DocID, the name, the type and the size of the markup, the version of its base document and it is read-only or not. Each piece of information is built into its own DMSProperty object, respectively labeled `CSI_DocID`, `CSI_DocName`, `CSI_MarkupType`, `CSI_DocSize`, `CSI_Version` and `CSI_DocReadOnly`. An additional `DMSProperty` object is needed for Read-Only

attribute. Note that a single `DMSProperty` property labeled `PROP_MARKUP` is stored for each markup.

```
private Property[] buildMarkupProperty(FilesysDMSBackend be, DMSBackendSession beSession,
        DMSQuery query) throws VuelinkException{
       final DocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
       DMSProperty guiProps[] = buildMarkupGui(be, beSession, docID);
       //Gets the list of markups from the DMS
       Vector mrkDocIds = be.dmsListMarkups(beSession, docID);
       DMSProperty markup[] = new DMSProperty[mrkDocIds.size()+1];
       markup[0] = new DMSProperty(Property.PROP_GUI,guiProps);

       for (int i = 0; i < mrkDocIds.size(); i++)
       {
              DMSProperty mrkProp[] = new DMSProperty[7];
              DMSProperty mrkProp[] = new DMSProperty[7];
              mrkProp[0] = new DMSProperty("CSI_DocID",
                            be.buildDocID(beSession,mrkDocIds.get(i)).DocID2String());
              mrkProp[1] = new DMSProperty("CSI_DocName", mrkDocIds.get(i).getName());
              …
              mrkProp[2] = new DMSProperty(Property.CSI_MarkupType, mrkType);
              mrkProp[3] = new DMSProperty(Property.CSI_DocSize,
                     mrkDocIds.get(i).getFile().length()+"");

              DMSProperty attrs = getAttrs(be, beSession,query, docID);
              mrkProp[4] = new DMSProperty(Property.CSI_Version,
                     attrs.getFirstChildValue("Version"));

              mrkProp[5] = new DMSProperty(Property.CSI_DocReadOnly,
                new Boolean(bReadOnly).toString());  // This is needed for AutoVue Server
              mrkProp[6] = new DMSProperty("Read-Only",
                new Boolean(bReadOnly).toString());

              markup[i+1] = new DMSProperty(DMSProperty.PROP_MARKUP,mrkProp);
       }
       …
       return markup;
}
```

7.3.2.1 Bundling PROP_GUI and PROP_MARKUP

Finally, the `execute()` method bundles the `PROP_GUI` and `PROP_MARKUP` properties in a `CSI_Markups` property and returns it to the VueLink servlet.

The registration of the `GetPropCSI_Markups` class is done as indicated below.

```
<init-param>
  <param-name>dms.getprops.CSI_Markups</param-name>
  <param-value>
       com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Markups
  </param-value>
```

For more information, examine the code and use a debugger to learn more about the behavior of this method.

Note: For saving and deleting Markups, refer to sections Action Save and Action Delete, respectively.

7.3.2.2 dmsListMarkup method

The `dmsListMarkups()` method in FilesysDMSBackendImp class asks the Filesys DMS backend system for the list of the Markups associated with a given document by providing its DocID.

```
package com.cimmetry.vuelink.filesys.backend;
…
public Vector<DocInfo> dmsListMarkups(DMSBackendSession session, DocID docID) {
        try{
                DocInfo fsDocID = buildDocInfo(session, docID);
                return m_filesysInfo.listMarkups(session,fsDocID);
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e );
                return null;
        }
}
```

7.4 Handling Renditions

The AutoVue server allows you to view hundreds of file formats. The viewed files are often large and time-consuming. To enhance performance, AutoVue generates files in a lightweight format called *streaming files*. Streaming files contain display information for the native file and are quickly accessed by AutoVue. AutoVue can also generate renditions such as TIFF, PDF and BMP format.

When a user wants to view a file, the AutoVue server sends several requests to the DMS system through the integration interface. One of these requests is related to streaming files. The AutoVue server sends a `GetProperties` request with the `CSI_Renditions` property in it. This request asks the DMS if it already has a streaming file associated with the base document. The response to this question is provided by the `GetPropCSI_Renditions`. A description of how this response is built is provided later in this section. If the response is yes, the AutoVue server sends requests to download the original file and the streaming file. Next, it verifies if the streaming files is a true replica, in which case AutoVue displays the streaming file instead the original one.

If the DMS does not have a streaming file, or the streaming file it has out of date, the client (for example, the applet) makes a request to the AutoVue server to generate a streaming file of the original file. When the user decides to close the viewed file, AutoVue sends a request to the DMS to save the generated streaming file. Refer to the Action Save section for information on how to build the response for this case.

In the Sample Integration for Filesys, the following excerpt of code shows how the `GetPropCSI_Renditions` class how the class encapsulates the DocID returned by the `getMetaRendition()` method of the Filesys DMS backend class in the `CSI_DocID` DMSProperty object.

```
package com.cimmetry.vuelink.filesys.propactions;

/**  */

public class GetPropCSI_Renditions implements DMSGetPropAction {

private DMSProperty[] buildRenditionProperty(FilesysDMSBackend be,
                    DMSBackendSession beSession, DocID docID) throws VuelinkException{

      FilesysDMSDocID rendDocIds = (FilesysDMSDocID)be.getMetaRendition(beSession, docID);

      if (rendDocIds == null) return null;

      DMSProperty[] metaRend = new DMSProperty[1];
      metaRend[0] = new DMSProperty(DMSProperty.CSI_DocID, rendDocIds);
      m_logger.debug("got the docID: " + metaRend);
      return metaRend;
}
```

As illustrated in the following code, the `execute()` method builds a `CSI_Renditions` DMSProperty and attaches to it an array `DMSProperties` with the first element to be a property labeled `CSI_DocID` for the streaming file rendition. The method then returns DMSProperty to the VueLink servlet which provides the AutoVue server with the response. The method also retrieves a list of supported rendition formats by the DMS backend system which is defined in web.xml. Note that this list of rendition formats is a subset of the rendition formats supported by the AutoVue server.

```
public DMSProperty execute(FilesysC … {

      final FilesysDMSDocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
      String sValidateMeta = context.getInitParameter("ValidateStreamingFile");

      String sRendition = context.getInitParameter("RenditionFormats");
      String[] aRenditionList = sRendition.split(";");

      DMSProperty[] rendition = null;

      if (sValidateMeta != null && sValidateMeta.equalsIgnoreCase("false")) {
            //no streaming file validation
            m_logger.debug("No StreamingFile Validation: ValidateStreamingFile option is
set to false in vuelink properties");
      } else {
            rendition = buildRenditionProperty(context.getBackendAPI(),
                        context.getBackendSession(session, query), docID);
      }

      return new DMSProperty(DMSProperty.CSI_Renditions, aRenditionList , rendition);
}
```

The `GetPropCSI_Renditions` is registered in the web.xml file as indicated in the following code.

```
 <init-param>
   <param-name>dms.getprops.CSI_Renditions</param-name>
   <param-value>
      com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Renditions
   </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this class.

The `getMetaRendition` method asks the FilesysDMS backend system for the streaming file associated with the base document identified by its DocID. After it receives the streaming file, it builds and returns the DocID.

```
public DocID getMetaRendition(DMSBackendSession session, DocID docID) {
        DocInfo fsDocID = buildDocInfo(session,docID);
        DocInfo metafile = null;
        try{
                metafile = m_filesysInfo.getMetaInstance(fsDocID);
                return buildDocID(session,metafile);
        }
        catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e );
                return null;
         }
}
```

7.5 Returning the List of All Properties of the DMS Document

When users select **Properties** from the **File** menu and then click the DMS tab (see Figure 7-3), the AutoVue server asks for some attributes of the current document by passing the `CSI_ListAllProperties` property within the `GetProperties` request. The response to this request is done through a property class called `GetPropCSI_AllProperties`.

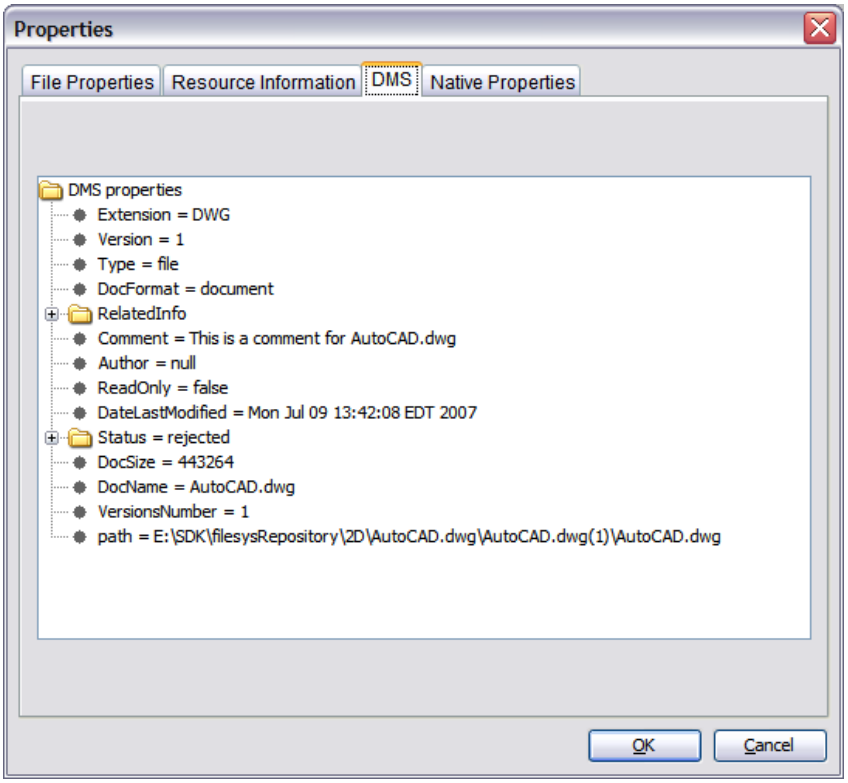


Figure 7-3. Properties dialog

In the Integration SDK Skeleton, `GetPropCSI_ListAllProperties` class calls the `listAllProperties()` method in the `DMSBackendImp` class to retrieve all the requested attributes and wraps them as a DMSProperty object to return.

In the case of the Filesys DMS, `GetPropCSI_AllProperties` class is derived from the `GetFilesysProperty` class and calls the `getAttrs()` method of the latter class which in turn calls the `getAttributes()` method of the FilesysDMSBackendImp class to retrieve the document attributes and build DMSProperty object to return. This is shown in the following excerpt of code. After getting the attributes, the `getAttributes()` method of the `FilesysDMSBackendImp` class builds a `DMSProperty` object for each attribute. For instance, it builds a DMSProperty named `CSI_Version` for the number of document versions. Finally, from this set of properties, a DMSProperty is built with the value set to `CSI_ListAllProperties` and is returned.

```
package com.cimmetry.vuelink.filesys.propactions;
….

public class GetPropCSI_ListAllProperties extends GetFilesysProperty
                                          implements DMSGetPropAction {
    …
   private DMSProperty buildListProperties(…) throws VuelinkException {

        DMSProperty attrs = getAttrs(context.getBackendAPI(), beSession, query,
                        docID);
        …
        return attrs;
   }
```

```
package com.cimmetry.vuelink.filesys.backend;
….

public DMSProperty getAttributes(DMSBackendSession session, DocID docID) {
        DocInfo fsDocID = buildDocInfo(session,docID);
        Vector<DMSProperty> result = new Vector<DMSProperty>();
        try{
                Hashtable<String,String> attrs=m_filesysInfo.getAttributes(fsDocID);
                Enumeration<String> keys = attrs.keys();
                while (keys.hasMoreElements()) {
                        String key = keys.nextElement();
                        String value = attrs.get(key);
                        if (value != null && value.split(";").length > 1) {
                                result.add(new DMSProperty(key,value.split(";")));
                                //multi value
                        }else {
                                result.add(new DMSProperty(key,value));//single value
                        }
                }
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e);
        }
        DMSProperty[] answer = new DMSProperty[0];
        answer = result.toArray(answer);

        return new DMSProperty(DMSProperty.CSI_ListAllProperties,answer);
}
```

This `GetPropCSI_AllProperties` class is registered in the web.xml file, as indicated in the following code excerpt.

```
 <init-param>
   <param-name>dms.getprops.CSI_ListAllProperties</param-name>
   <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ListAllProperties
 </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this class.

7.6 Implementing File Browse

Users may want to browse the DMS backend system to select documents for viewing or comparison. In this case, the AutoVue Server sends two `GetProperties` requests. The first request is for the GUIs that will support the definition of the browse operation. The second request is for the result of the browse action performed by the user.

7.6.1 GUI Request

In the first request, AutoVue asks for the Browse dialog structure by passing the GUI property with a value set to **Browse** within the request. The response to this first request is done through a property class called `GetPropGUI`. The GUI section defines the columns displayed in the Browse dialog.

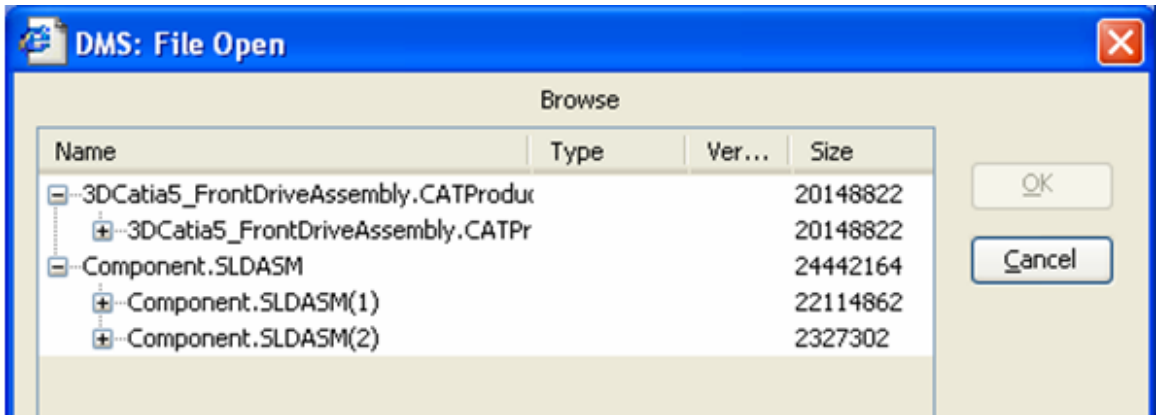


Figure 1. DMS Browse dialog

The following excerpt of code shows how to construct the Browse dialog shown in Figure 7-4. Note that this is the same dialog used in the Sample Integration for Filesys. Each column is identified with a unique ID and constructed as `DMSProperty` object.

- Document name `CSI_DocName`
- Document type folder or file `SP_TYPE`
- Document version `SP_FileVersion`
- Document size `CSI_DocSize`

You can specify the size for each column. Note that the **Name** column is a GUI tree where row values can be either a file or a folder; the folders are nodes that can be expanded by the user. All these properties are returned as single property labeled `PROP_GUI_DISPLAY`.

```
package com.cimmetry.vuelink.filesys.propactions;
…
private DMSProperty[] buildBrowseGUI() {

        final String SP_Type        = "Type";
        final String SP_FileVersion = "Version";

        DMSProperty[] guiValue = new DMSProperty[4];
        guiValue[0] = new DMSProperty(DMSProperty.CSI_DocName,"35");
        guiValue[1] = new DMSProperty(SP_Type,"10");
        guiValue[2] = new DMSProperty(SP_FileVersion,"6");
        guiValue[3] = new DMSProperty(DMSProperty.CSI_DocSize,"14");

        DMSProperty[] gui = {new
        DMSProperty(DMSProperty.PROP_GUI_DISPLAY,guiValue)};
        m_logger.info("building GUI for browsing: " + guiValue);
        return gui;
}
```

7.6.2 Request for Browse Results

The second request sent by the AutoVue server is for the list of browse results. These results appear as children nodes in the **Name** tree in figure 7-4. This request is done by calling `GetProperties` and passing the `CSI_Browse` property as a parameter. The response to this request is handled by `GetProp_ListItems` class. This class returns the data that populates the Browse dialog.

In the Sample Integration for Filesys, all this information is obtained by calling the `dmsListItemsForBrowse` method of the FilesysDMS backend class. This method returns a vector of DocIDs of the expanded document's direct children nodes.

The following excerpt shows how `GetProp_Browse` builds properties for returning a list of documents in the Sample Integration for Filesys. For each document we build a DMSProperty for each of the following information and wrap them together in a single DMSProperty labeled `CSI_DocID`.

- Type of document folder or file `CSI_ItemType`
- Document name `CSI_DocName`
- Date of last modification `CSI_DocDateLastModified`
- Document size `CSI_DocSize`
- Version of the document `Version`

Finally, the `execute()` method gathers the built properties for all listed documents in a single property labeled *`CSI_ListItem`* and returns it to the VueLink servlet.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class GetPropCSI_ListItems implements DMSGetPropAction {
   …
   private DMSProperty[] buildListItems(FilesysDMSBackend be, DMSBackendSession beSession,
               DocID _rootID){

       DocID rootID = _rootID;
        // Gets the of items from the DMS
       Vector<DocID> listItemsInfos = be.dmsListItemsForBrowse(beSession, rootID);

       if (listItemsInfos != null) {
              DMSProperty listItems[] = new DMSProperty[listItemsInfos.size()];
              …
              for (int i = 0 ; i < listItemsInfos.size() ; ++i) {
                     DocID instId = listItemsInfos.get(i);
                     DMSProperty docAttrs = be.getAttributes(beSession,instId);
                     DMSProperty props[] = new DMSProperty[5];

                     props[1] = new DMSProperty(DMSProperty.CSI_DocName,
                              docAttrs.getFirstChildValue("DocName"));

                     if (!docAttrs.getFirstChildValue("DocFormat").equals("folder")) {
                            // a file
                            props[0] = new DMSProperty(DMSProperty.CSI_ItemType,
                                   DMSProperty.CSI_Document);
                            props[2] = new DMSProperty("Type",
                                   docAttrs.getFirstChildValue("Extension"));
                            props[3] = new DMSProperty("Version",
                                   docAttrs.getFirstChildValue("Version"));
                            props[4] = new DMSProperty(DMSProperty.CSI_DocSize,
                                   docAttrs.getFirstChildValue("DocSize"));
                     }else{ // a folder
                            props[0] = new DMSProperty(DMSProperty.CSI_ItemType,
                                          DMSProperty.CSI_Folder);
                     }
                     listItems[i]= new DMSProperty(DMSProperty.CSI_DocID,
                                          instId.DocID2String(),props);
              }
              …
              return listItems;
       }else{
              return null;
       }
   }
```

```
public DMSProperty execute(…) throws VuelinkException {

       final DocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());
       DMSProperty retProp = new DMSProperty(DMSProperty.CSI_ListItems,
              buildListItems(((FilesysDMSBackend)context.getBackendAPI()),
                            context.getBackendSession(session, query), docID));
       return retProp;
}
```

The classes `GetPropGUI` and `GetPropCSI_ListItems` are registered in the web.xml as indicated below.

```
 <init-param>
   <param-name>dms.getprops.CSI_GUI</param-name>
   <param-
 value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_GUI</param
 -value>
 </init-param>
```

```
<init-param>
  <param-name>dms.getprops.CSI_ListItems</param-name>
  <param-
value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ListItems<
/param-value>
```

For more information, examine the code and use the debugger to learn more about the real behavior of these classes.

The `dmsListItemsForBrowse` method asks the Filesys DMS backend system for the list of direct children of a node given by its DocID. After it receives the vector of the direct children of the document, it builds a DocID for each child. Finally, it returns the list of the DocIDs as a vector.

```
package com.cimmetry.vuelink.filesys.backend;
…
public Vector<DocID> dmsListItemsForBrowse(DMSBackendSession session, DocID docID) {

        DocInfo fsDocID = buildDocInfo(session,docID);

        Vector<DocInfo> browseItemsIDs = null;
        try{
                browseItemsIDs = m_filesysInfo.listItemsForBrowse(fsDocID);
        }catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR, e );
        }
        if (browseItemsIDs == null) {
                return null;
        }

        Vector<DocID> docIDs = new Vector<DocID>();
        for (int i = 0 ; i < browseItemsIDs.size() ; ++i) {
                docIDs.add(buildDocID(session,browseItemsIDs.get(i)));
        }
        return docIDs;
}
```

7.7 Implementing File Search

You may want to search for documents in the DMS backend system for viewing or comparison. In this case, the AutoVue server sends two `GetProperties` requests: one is for the GUI components that support the definition of the search operation and the other is for the result of the search operation that displays on the GUI.

7.7.1 First Request

There are two dialogs to define. In the first one we define the search criteria elements. In the second dialog we define the structure where the returned information elements are displayed. In the first request AutoVue asks for the structures of the two dialogs by passing the GUI property with a value of **Search** within the request. The response to this first request is

handled by a property class called `GetPropGUI` (this class is presented in the Request for Browse Results).

The response is specified by two parts: EDIT and DISPLAY. The EDIT response specifies the GUI elements of the search dialog to use when entering the search criteria. This dialog includes two GUI Elements: Criteria drop-down list and Value field. The control ID for the Criteria list is `CSI_Criteria` and it contains two selections: Name and Type. The default value is Name. The Value field's control ID is `CSI_Entry`.

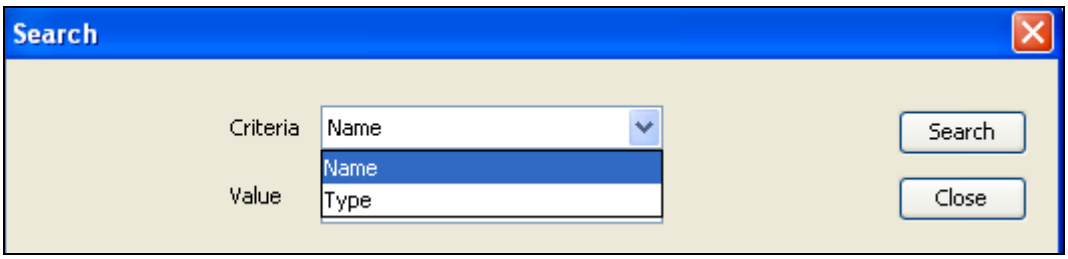


Figure 7-5. Search dialog

The following excerpt of code prepares information for building the first part of the response in the Sample Integration for Filesys. It builds a `GUIElementCombo` property for specifying the drop down list and a `GUIElementEdit` property for specifying the edit box. The two properties are returned in a single property labeled `PROP_GUI_EDIT`.

```
package com.cimmetry.vuelink.filesys.propactions;
…
private DMSProperty addEditForSearch() throws VuelinkException{
        DMSProperty props = null;

        String [] values = {"By name","By type"};
        GUIElementCombo comboForType  =  new GUIElementCombo("CSI_Criteria",
                                "Search criteria", null, values, true);

        GUIElementEdit editForName = new GUIElementEdit("CSI_Entry",
                                "Search for", null, false);

        Property [] p = new Property[2];
        p[0] = comboForType;
        p[1] = editForName;

        props = new DMSProperty(Property.PROP_GUI_EDIT, p );
        return props;
}
```

In the second part of the response, DISPLAY specifies columns to be displayed inside the Search dialog as shown in Figure 7-6.

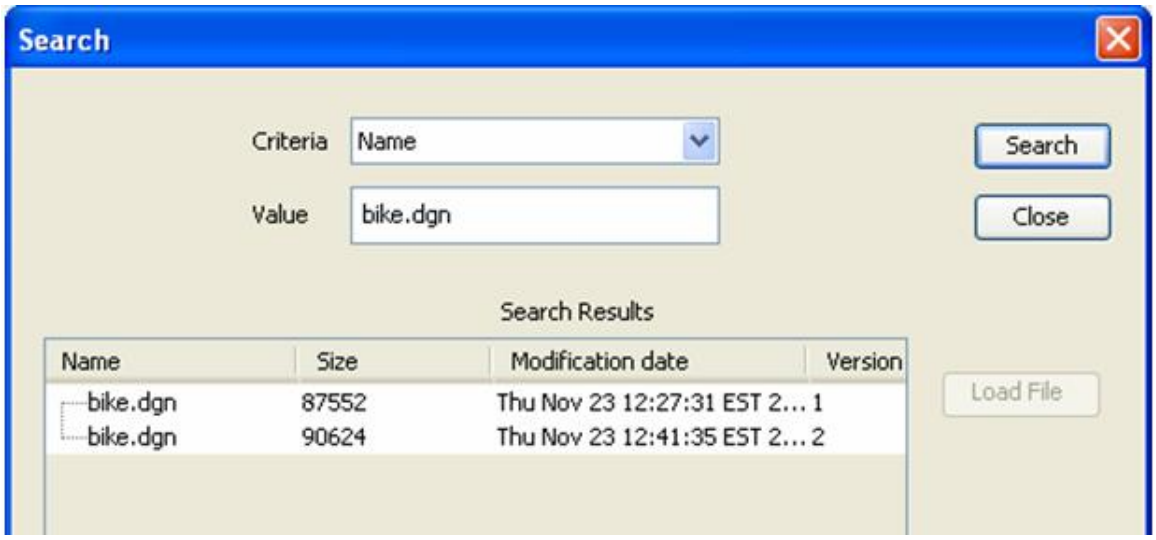


Figure 7-6. Search results

The following excerpt of the code shows how this box is defined in the Sample Integration for Filesys. It builds properties for the following information:

- Document name `CSI_DocName`
- Document size `CSI_DocSize`
- Date of last modification `CSI_DocDateLastModified`
- Version of the document `CSI_Version`

All these properties are returned in a single property labeled `PROP_GUI_DISPLAY`.

```
private DMSProperty addDisplayForSearch() throws VuelinkException{

        DMSProperty[] props = new DMSProperty[4];
        props[0] = new DMSProperty(DMSProperty.CSI_DocName, "18");
        props[1] = new DMSProperty(DMSProperty.CSI_DocSize, "18");
        props[2] = new DMSProperty(DMSProperty.CSI_DocDateLastModified, "18");
        props[3] = new DMSProperty(DMSProperty.CSI_Version, "4");

        return new DMSProperty(Property.PROP_GUI_DISPLAY, props );
}
```

The two parts are then combined and returned as a single property labeled as `Prop_GUI`.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public DMSProperty buildSearchGUI(…) throws VuelinkException{
        m_logger.debug("***inside getSearchGuiProperty() ");

        // get the GUI property value
        DMSProperty[] props = new DMSProperty[2];
        props[0] = addEditForSearch();
        props[1] = addDisplayForSearch();
        return new DMSProperty(DMSProperty.PROP_GUI, "Search", props);
}
```

7.7.2 Request for Search Results

The second request sent by the AutoVue server is for the list of items that match the search criteria. This is done through a `GetProperties` request containing the `CSI_Search` property. The response to this request is handled by the `GetProp_Search` class. This class must return the data that populates the Search dialog.

In the Sample Integration for Filesys, the search results are obtained from the `dmsListItemsForSearch` method of the FilesysDMS backend class. The following excerpt of code shows how the `GetProp_Search` class builds properties for the returned document. For each document, we build a DMSProperty for each of the following information and wrap them together in a single DMSProperty labeled `CSI_DocID`.

- Type of document folder or a file `CSI_ItemType`
- Document name `CSI_DocName`
- Date of last modification `CSI_DocDateLastModified`
- Document size `CSI_DocSize`
- Version of the document `CSI_Version`

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class GetPropCSI_Search extends GetFilesysProperty implements
              DMSGetPropAction<FilesysContext> {

   private Property[] listItems(…) throws VuelinkException{
        …
       Vector items = be.dmsListItemsForSearch(docID, rootDir, criteria, type);
       DMSProperty[] sItems = new DMSProperty[items.size()];

       for (int i = 0; i < items.size(); i++) {
              DMSProperty sProp[] = new DMSProperty[5];
              docID = items.get(i);

              DMSProperty attrs = (DMSProperty) query.getQueryData( "attrs");
              …
              if (attrs.getFirstChildValue("DocFormat").equals("folder")) {
                  sProp[0] = new DMSProperty(DMSProperty.CSI_ItemType, \
                                DMSProperty.CSI_Folder);
              }else{
                  sProp[0] = new DMSProperty(DMSProperty.CSI_ItemType,
                      DMSProperty.CSI_Document);
              }
              sProp[1] = new DMSProperty(Property.CSI_DocName,
                       attrs.getFirstChildValue("DocName"));
              sProp[2] = new DMSProperty(Property.CSI_DocSize,
                      attrs.getFirstChildValue("DocSize"));
              sProp[3] = new DMSProperty(Property.CSI_DocDateLastModified,
                      attrs.getFirstChildValue("DateLastModified"));
              sProp[4] = new DMSProperty("CSI_Version",
                      attrs.getFirstChildValue("Version"));
              sItems[i] = new DMSProperty(Property.CSI_DocID, docID.DocID2String(), sProp);
       }
       m_logger.info("Get the list of items that match the search creteria :" + sItems);
       return sItems;
   }
```

Finally, the `execute()` method gathers all the built properties in a single property labeled `CSI_Search` and returns it to the VueLink servlet.

The `GetPropCSI_Search` class is registered in the web.xml file as indicated in the following code excerpt.

```
<init-param>
  <param-name>dms.getprops.CSI_Search</param-name>
  <param-value>
       com.cimmetry.vuelink.filesys.propactions.GetPropCSI_ Search
  </param-value>
</init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this class.

As shown in the following, the `dmsListItemsForSearch` method asks the Filesys DMS backend system for the list of documents that match the search criteria. To perform the search, the backend system provides the criteria type, criteria value, and the backend system root.

```
public Vector<DocID> dmsListItemsForSearch(DocID docID, String root,
                                           String creteria, String type) {
      DocInfo fsDocID = buildDocInfo(session,docID);
      // comments
      Vector searchItemsIDs = null;
      …
      searchItemsIDs = m_filesysInfo.listItemsForSearch(fsDocID,root,
                                                        criteria, type );
      …
      Vector<DocID> docIDs = new Vector<DocID>();
      for (int i = 0 ; i < searchItemsIDs.size() ; ++i) {
            docIDs.add(new FilesysDMSDocID((DocInfo)searchItemsIDs.get(i)));
      }
      return docIDs;
}
```

When the `dmsListItemsForSearch` method receives the vector of the files from the DMS backend system, it builds a DocID for each file and then returns the list of the DocIDs as a vector.

7.8 Handling Versions

To compare the viewed document with another version of the document, from AutoVue, you must select **Analysis** and then **Compare** to launch File Open dialog. At this moment, the AutoVue server sends a `GetProperties` request asking the DMS backend system for all versions of the current document by passing `CSI_Versions` property within it. The response to the request is handled through a property class called `GetPropCSI_Versions`.

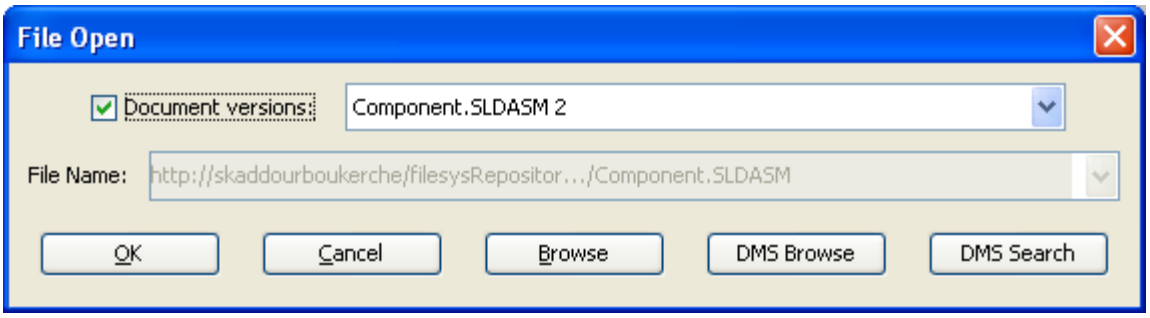


Figure 7-7. File Open dialog with Document versions for compare

In the Sample Integration for Filesys, to build the response, the `GetPropCSI_Versions` class first receives from `dmsListVersions()` method a vector of DocIDs of all the version of the document. It then loops through each version and builds `CSI_Version` property.

The following excerpt of code shows how we build the content of the `CSI_Versions` property. For each version of a document we create a `PROP_VERSION` property and we attach to it the DocID `CSI_DocID`, the name `CSI_DocName` and the version number `CSI_Version` properties. Finally, the list of `PROP_VERSION` properties are attached to `CSI_Versions` property and returned to the VueLink servlet.

```
private DMSProperty[] buildListProperties(FilesysDMSBackend be, DMSBackendSession
beSession, DocID docID){

   Vector<DocID> versionsDocIDs = be.dmsListVersions(beSession, docID);

   DMSProperty[] versions = new DMSProperty[versionsDocIDs.size()];

   for (int i = 0; i < versionsDocIDs.size(); i++) {
        DMSProperty[] aVersion = new DMSProperty[3];
        FilesysDMSDocID doc = (FilesysDMSDocID)(versionsDocIDs.get(i)) ;

        aVersion[0] = new DMSProperty(DMSProperty.CSI_DocID, doc.DocID2String());
        aVersion[1] = new DMSProperty(DMSProperty.CSI_DocName,
                ((FilesysDMSDocID)doc).getName());
        aVersion[2] = new DMSProperty(DMSProperty.CSI_Version, doc.getVersion());
                    versions[i] = new DMSProperty(DMSProperty.PROP_VERSION, aVersion);
   }

   m_logger.info("Get the list of versions of a document :" + versions);
   return versions;
}
```

The `GetPropCSI_Versions` class is registered in the web.xml file as indicated above.

```
 <init-param>
   <param-name>dms.getprops.CSI_Versions</param-name>
   <param-value>
         com.cimmetry.vuelink.filesys.propactions.GetPropCSI_Versions
   </param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

The `dmsListVersion` method asks the Filesys DMS backend system for the list of document versions by providing the DocID of the current document.

```
public Vector<DocID> getVersions(DMSBackendSession session, DocID docID) {
        Vector<DocID> versions = null;
        Vector<DocInfo> versionsInfos =  null;
        try{
                DocInfo fsDocID = buildDocInfo(session,docID);
                versionsInfos = m_filesysInfo.listVersions(fsDocID);
        } catch(Exception e){
                m_logger.error(DMSDefs.DMS_ERROR_CODE_ERROR , e);
        }
        versions = new Vector<DocID>();
        for (int i = 0; i < versionsInfos.size() ; ++i) {
                versions.add(buildDocID(session,versionsInfos.get(i)));
        }
        return versions;
}
```

After it receives the vector of the document versions, it builds a DocID for each element. Finally, it returns the list of the DocIDs as a vector.

7.9 Implementing handler for Default Property

When the AutoVue server sends a `GetProperties` request with a property that does not have a class for handling it, the framework runs the `GetPropDefault` class. The `GetPropDefault` class is not dedicated to a particular property and there is no property called `Default`, so when you register the web.xml file you must use `dms.getprops.Default` as the parameter name. Of course, you can give the class a different name from the default one. However, if you choose not to register the class, then you must name it `GetPropDefault`.

```
<init-param>
 <param-name>dms.getprops.Default</param-name>
  <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropDefault</param-value>
</init-param>
```

Later we will discuss when to use individual classes for handling properties and when to use `GetPropDefault` class. Also we will discuss how you can avoid implementing the `GetPropDefault` by implementing a class for each request property.

For more information, refer to the source code of this class and run this class in debug mode for more information on its behavior.

The following figure shows code of the `execute()` method of the `GetPropDefault` class of the Integration SDK Skeleton.

```
package com.mycompany.autovueconnector.propactions;
…
public class GetPropDefault implements DMSGetPropAction<DMSContextImp>, DMSDefs {
        …
        public DMSProperty execute(
                        DMSContextImp context,
                        DMSSession session,
                        DMSQuery query,
                        DMSArgument[] args,
                        Property property
                        ) throws VuelinkException {

        final String propName = property.getName();

        if ("VueLinkID".equals(property.getName())) {
          return new DMSProperty("VueLinkID", "");
        }

        if ("CSI_MIMETypes".equals(propName)) {
                return new DMSProperty("CSI_MIMETypes", MIME_TYPES);
        }

        DMSProperty prop = null;
        try {
                DMSBackendImp be = (DMSBackendImp)context.getBackendAPI();
                DMSBackendSessionImp beSession =
                        (DMSBackendSessionImp)context.getBackendSession(session,query);

               if (DMSProperty.CSI_AllowBrowse.equals(propName)) {
                        return new DMSProperty(DMSProperty.CSI_AllowBrowse,
                        be.isAllowBrowse(beSession));
                }

               if (DMSProperty.CSI_AllowSearch.equals(propName)) {
                        return new DMSProperty(DMSProperty.CSI_AllowBrowse,
                        be.isAllowSearch(beSession));
                }
               …

                DMSProperty  attrs = (DMSProperty)query.getQueryData("attrs");
                ISDKDocID docID = new ISDKDocID().String2DocID(query.getDocID());

                if(attrs == null ){
                        attrs = be.listAllProperties(beSession, docID);
                        if(attrs != null ){
                                query.setQueryData("attrs", attrs);
                        }
                 }

                 prop = (DMSProperty)attrs.getFirstChildWithName(propName);

        } catch (Exception e) { … }

        if (prop == null) {
                m_logger.error("Unsupported property: " + propName);
                throw new VuelinkException(DMSDefs.ERROR_CODE_DMS_GETPROPERTIES,
                                "Unsupported property: " + propName);
        }

        return prop;
        }
}
```

7.10 Implementing File Save Action

You can create and modify markups and convert documents to other formats as TIFF and PDF. When these documents are saved in the DMS backend system by selecting the **Save** or

Save As actions from AutoVue's **File** menu, the AutoVue server sends an Action Save request. The response of this request is done through the `ActionSave` class.

In saving Markups, there are two cases to handle. The first case is when trying to save a new Markup file. In this case, and as shown in the following excerpt of code, the Save action must return a valid DocID of the newly created Markup. The second case is when trying to save an existing Markup file. In this case the Markup file keeps its old DocID. For saving Markups, the `ActionSave` class relies on the service of the `saveMarkup()` method of the Filesys DMS backend class.

When performing conversion of a document by selecting the **Convert** action from the **File** menu, AutoVue exhibits the same behavior as for saving Markups. But this time the `ActionSave` invokes the `saveRendition()` method of the Filesys DMS backend class.

When an AutoVue Real-Time Collaboration is closed, the chat transcript during the collaboration session might need to be saved. In this case, `ActionSave` invokes the `saveChat()` method of the Filesys DMS backend class to save the chat content.

In the Sample Integration for Filesys, the `getDMSArgsProperties` API is very useful. This API provides properties about the DocID of the base document, the DocID of the Rendition or the Markup document if it exists, the Markup and Rendition types, and the Markups and Renditions files name. This information lets the Filesys DMS backend system locate where the documents are saved, and is therefore very important.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class ActionSave implements DMSAction<FilesysContext>, DMSDefs{
        …
        public Object execute(final FilesysContext  context,
              final DMSSession  session,
              final DMSQuery   query,
              final DMSArgument[] args
              ) throws VuelinkException {
        …
        final Property[] props = query.getDMSArgsProperties();
        …
        // Get file name
        final DMSArgument fileArg = args[0];
        String type = fileArg.getType(); …
        String sUploadFile = fileArg.getName(); …

        boolean bSaveChat = false;    // "True" if saving chat content for a meeting
        boolean bReadOnly = false;    /* true if it is a read-only markup */
        String  rendType  = null;
        DocID   baseID = null;        /* if non-null, we're doing a Save-As */
        DocID   saveID = null;        /* if non-null, we're doing a Save */
        String  docName = null;       /* the value of 'name' for Save, or 'CSI_DocName'
                                         for Save-As */
        String  markType = null ;     /* not null if the mark type is specified */

        if (props != null) {
              … // assign values for the above variables
        }
        …
        /** Upload the file */
        DocID newDocID = null;
        try {
               InputStream fIn = null;
               … // put uploading content in fIn
               if (bSaveChat) { // save collaboration chat transcript
                      …
                      return be.saveChat(beSession, docName, fIn);
               }
               else
               if(rendType != null) {  // save rendtion (new or existing)
                      …
                      newDocID = be.saveRendition(beSession,baseID, saveID, docName,
                             rendType, fIn);
               } else {// save markup (new or existing)
                      newDocID = be.saveMarkup(beSession,baseID, saveID, docName,
                             markType, bReadOnly, fIn);
                      }
        }catch (…) {…}

        return newDocID;
}
```

In the Sample Integration for Filesys, the `saveMarkup` method uploads the Markup file as an `InputStream` object and invokes the `saveMarkup` method from the FilesysDMS backend to save the file in the backend system. The parameters are: DocID of the base document (docID), DocID of the markup (mrkID) which is null for new markup, the markup file name, the markup type, the markup read-only attribute and the markup file content as `InputStream`.

```
package com.cimmetry.vuelink.filesys.backend;
…
public DocID saveMarkup(DMSBackendSession session, DocID docID, DocID mrkID,
        String filename, String markupType, boolean bReadOnly, InputStream fIn
        throws FileNotFoundException, IOException, VuelinkException {

        DocInfo fsDocID = null;
        if (mrkID == null) { // save new Markup
                …
                fsDocID = buildDocInfo(session,docID);
        } else { // save existing Markup
                …
                fsDocID = buildDocInfo(session,mrkID);
        }

        return buildDocID(session,
           m_filesysInfo.saveMarkup(fsDocID, markupType, bReadOnly, filename, fIn));
}
```

It returns the DocID of the saved Markups if it fails it throws an exception.

In the Sample Integration for Filesys, the `saveRendition` methods upload the Rendition file as an `InputStream` object and invokes the `saveRendition` method from the FilesysDMS backend to save the file in the backend system. The parameters are: DocID of the base document, DocID of the rendition (renID) which is null for new rendition, the rendition file name, the rendition type, and the rendition file content as `InputStream`.

```
package com.cimmetry.vuelink.filesys.backend;
…
public DocID saveRendition(DMSBackendSession session, DocID docID, DocID renID,
        String filename, String rendType, InputStream fIn)
        throws FileNotFoundException, IOException, VuelinkException {

        DocInfo fsDocID = null;
        if (renID == null) { //new rendition
                fsDocID = buildDocInfo(session,docID);
        } else {  // existing rendtion
                fsDocID = buildDocInfo(session,renID);
        }
        return buildDocID(session,
                m_filesysInfo.saveRendition(fsDocID, rendType, filename, fIn));
        }
}
```

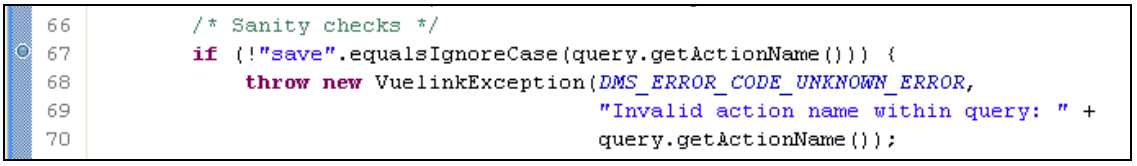
It returns the DocID of the saved rendition. If it fails, it throws an exception.

For saving Collaboration chat transcript, refer to section Implementing RTC and Meeting Management for a detailed description.

The `SaveAction` class is registered in the web.xml file as indicated in the following code excerpt.

```
 <init-param>
   <param-name>dms.action.Save</param-name>
   <param-value>com.cimmetry.vuelink.filesys.actions.ActionSave</param-value>
 </init-param>
```

Refer to the source code of this class for more information. You can also run this class in debug, as shown in the following figure to help you learn more about the dynamic behavior of this class.



```
66          /* Sanity checks */
67          if (!"save".equalsIgnoreCase(query.getActionName())) {
68              throw new VuelinkException(DMS_ERROR_CODE_UNKNOWN_ERROR,
69                                         "Invalid action name within query: " +
70                                         query.getActionName());
```

7.11 Implementing File Delete Action

You have the option of deleting Markups from within the AutoVue applet. When deleting existing markups, the AutoVue server sends a Delete Action request. The response to this request is handled by the `ActionDelete` class. The document to be deleted is indicated by the DocID parameter.

In the Sample Integration for Filesys, to be deleted effectively from DMS backend system this class sends its request through the `deleteMarkup()` method of FilesysDMS backend class.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class ActionDelete implements DMSAction<FilesysContext>, DMSDefs{
        …
        public Object execute(final FilesysContext  context,
                final DMSSession  session,
                final DMSQuery   query,
                final DMSArgument[] args
                ) throws VuelinkException {
        …
        final DocID docID = new FilesysDMSDocID().String2DocID(query.getDocID());

        // delete markup document
        if (! context.getBackendAPI().deleteMarkup(
                  context.getBackendSession(session,query), docID)) {
                …
                throw new VuelinkException(DMS_ERROR_CODE_ERROR,
                                           DMS_ERROR_MSG_DELETE);
        }
        return null;
}
```

In the Sample Integration for Filesys, the `deleteMarkup` method sends a request to the DMS backend system to delete the markups identified by a DocID passed in the parameter. If the document is deleted, it returns TRUE. Otherwise it returns FALSE.

```
package com.cimmetry.vuelink.filesys.backend;
…
public boolean deleteMarkup(DMSBackendSession session, DocID docID)
        throws VuelinkException{

        DocInfo fsDocID = buildDocInfo(session,docID);
        …
        boolean deletedDoc = false;
        try{
                deletedDoc = m_filesysInfo.deleteDocument(fsDocID);
        }catch(Exception e){ … }

        return deletedDoc;
}
```

The `ActionDelete` class is registered in the web.xml file as shown in the following excerpt of code.

```
 <init-param>
   <param-name>dms.action.Delete</param-name>
   <param-value>com.cimmetry.vuelink.filesys.actions.ActionDelete</param-value>
 </init-param>
```

For more information, examine the code and use the debugger to learn more about the behavior of this method.

7.12 Creating Your Context

Each VueLink has a context that holds various environment settings that remain constant throughout the VueLink servlet lifetime. This context is initialized during the VueLink servlet initialization and is passed to actions every time the VueLink handles a request.

The framework publishes the `com.cimmetry.vuelink.context.`*`DMSContext`* interface which describes a set behavior that a context handler must exhibit, which includes:

- Initializing this `DMSContext` by fetching the appropriate information within the DMS servlet initialization parameters.
- Finding, registering, and locating the appropriate backend API class for the current DMS servlet.
- Finding the backend session object corresponding to the `DMSSession`.
- Creating a new backend session if an existing session cannot be found.

The framework provides the `com.cimmetry.vuelink.context.GenericContext` class which is a default implementation of the *`DMSContext`* interface. You must provide your own implementation of the *`DMSContext`* interface only if the `GenericContext` does not satisfy your needs. It is recommended that you extend your context from `GenericContext` class.

For each DMS servlet, the context action is registered during the initialization of the DMS servlet and loaded by the framework in the following sequence:

1. It fetches the initialization parameters looking for whether a custom action context (with param-name as “dms.context”) is provided.
2. It looks for `ActionContext` class in the same location as the Integration SDK’s DMS VueLink servlet.
3. It looks for `ActionContext` class in the same location as the framework DMS VueLink servlet.
4. It looks for `GenericContext` in the same location as the framework DMS VueLink servlet.
5. It throws an exception if it does not succeed in finding a class to handle the context.

In the Filesys DMS application, a new class `FilesysContext` is extended from the `GenericContext` class to provide a custom context.

```
package com.cimmetry.vuelink.filesys;
…
public class FilesysContext extends GenericContext {
    …
    public FilesysDMSBackendImp getBackendAPI() throws VuelinkException {
        if (m_backend == null) {
                throw new VuelinkException(DMS_ERROR_CODE_ERROR,
                   "Backend API not registered");
        }
        return (FilesysDMSBackendImp)m_backend;
    }

    public DMSBackendSession getBackendSession(DMSSession session, DMSQuery query)
         throws AuthorizationException {
        …
        // Get BackendSession from DMSSession if it has been put there before
        if (session.getAttribute("backendSession") != null) {
                …
        }

        // No backend session exists yet. Establish new connection to DMS and
        // create new backend session.
        …
        Hashtable<String,Object> connectInfo = new Hashtable<String,Object>();
        …
        FilesysBackendSession backendSession =
                (FilesysBackendSession)m_backend.connect(connectInfo);
        …
        return backendSession;
    }
}
```

The `FilesysContext` class is registered in web.xml. If your context is not in the same location as your DMS VueLink servlet, you have to register it.

```
<init-param>
 <param-name>dms.actions.Context</param-name>
  <param-value>com.cimmetry.vuelink.filesys.FilesysContext</param-value>
</init-param>
```

7.13 Overriding GetProp<CSI Property> classes

You may want to extend the response provided by a property class. For instance, you may add the number of all existing versions of a document to the `ListAllProperties` response. There are several ways to implement a mechanism that lets you extend the behavior of property classes. One mechanism you may consider is inheritance. A second one may be similar to the mechanism already implemented in the framework, such as simply implementing a new class that implements the `DMSGetPropAction` interface and registers it in the web.xml file.

The inheritance has two limitations. The first limitation is that the new behaviors are added statically (for example, at compilation time). The second is that for each new behavior, we must derive a new class and we know that the multiplication of the number of classes can be a maintenance nightmare. The second mechanism consists of replacing the old class by a new one which implements the new behaviors. A better solution is to add new behaviors to existing ones since it is not necessary to rewrite existing code that has been tested and proven to be bug-free.

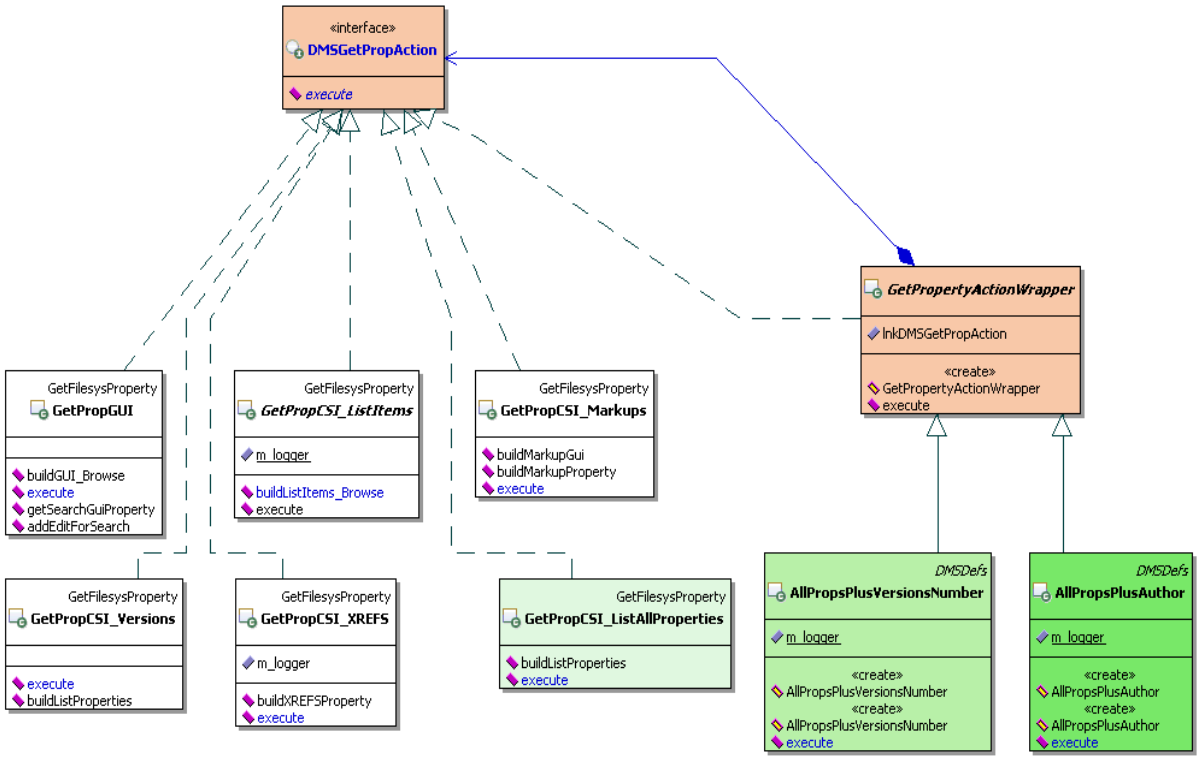


Figure 7-8. Structure of the DMSGetPropAction interface

An advanced integration mechanism has been designed that allows integrators and professional services to extend the handling of specific CORE API messages without recompiling or rebuilding the entire integrations by just adding the overriding code. As illustrated in the previous figure, and in the excerpt of the following code, a class called

`com.cimmetry.vuelink.propsaction.GerPropertyActionWrapper` has been designed. The class implements the `com.cimmetry.vuelink.propsaction.`*`DMSGetPropAction`* interface and has a variable that references any object that implements this interface. Note that the wrapper class implements the same interface as the classes it is going to wrap.

```
package com.cimmetry.vuelink.propsaction;

/**  */

public abstract class GetPropertyActionWrapper implements DMSGetPropAction<DMSContext>
{
        /**
         * {@link com.cimmetry.vuelink.core.DMSGetPropAction} object instance
         */
        protected DMSGetPropAction propertyAction ;

        /**
         * Constructs a decorator from the object to extend
         *
         * @param propAction object to extend
         */
        public GetPropertyActionWrapper(DMSGetPropAction propAction){
                this.propertyAction = propAction;
        }
}
```

To add a new behavior you just have to add a new class derived from the wrapper class. This mechanism allows third-party integrators to easily upgrade their solutions.

For example, in the Sample Integration for Filesys, to add the number of versions of a document to the ListAllProperties class we can create a new `AllPropsPlusVersionsNumber` class that wraps the `GetPropCSI_ListAllProperties` and adds to it the number of versions of a document.

```
public class AllPropsPlusVersionsNumber extends GetPropertyActionWrapper
                                                implements DMSDefs {
        /**
         * Wrap the existing object
         */
        public AllPropsPlusVersionsNumber(){
                super(new GetPropCSI_ListAllProperties());
        }
        public DMSProperty execute(DMSContext context, DMSSession session,
                        DMSQuery query, DMSArgument[] args, Property property)
                                                        throws VuelinkException {
                // add the new behavior
                …
```

Finally, you must register this class as indicated in the following excerpt of code. Note that the wrapper is still using the services of the object it wraps.

```
 <init-param>
  <param-name>dms.getprops.CSI_ListAllProperties</param-name>
  <param-value>
    com.cimmetry.vuelink.filesys.propactions.GetPropCSI_AllPropsPlusVersionsNumber
  </param-value>
 </init-param>
```

The major advantage of this mechanism is its capability to dynamically compose wrapper classes. For example, you may add a new behavior to the same class by adding the document author property you just have to follow the same steps above. But in this case, wrap a wrapper class as shown in the following excerpt of code.

```
public class AllPropsPlusAuthor extends GetPropertyActionWrapper implements
            DMSDefs {

      public AllPropsPlusAuthor(){ super(new AllPropsPlusVersionsNumber(new
                                                  _ListAllProperties()));
                                                  }
```

You can also decide that a document has two authors. In this case, you need to compose the new behavior as indicated in the following line of code without adding any line of code.

```
new AllPropsPlusAuthor(new AllPropsPlusAuthor(new AllPropsPlusVersionsNumber(new
CSI__GetListAllProperties())))
```

7.14 Implementing Read-Only Markups

In combination with the AutoVue markup type (normal, master and consolidate), a markup can be created as read-only that cannot be updated after being created. To support read-only markups, the integration interface should enhance *CSI_Markups* and *Save* requests sent by the AutoVue server described in section Handling Markups.

The response to the CSI_Markups request is to specify the markup GUI for the Open and Save requests. The GUI is enhanced to allow users to choose to save a markup as read-only and when listing existing markups to display markups with the read-only attribute.

The response to the Save request sets the markup file as read-only if requested by the user. It can be either a physical read-only file as with the sample integration for FileSys, or the ready-only attribute is set in the meta-data.

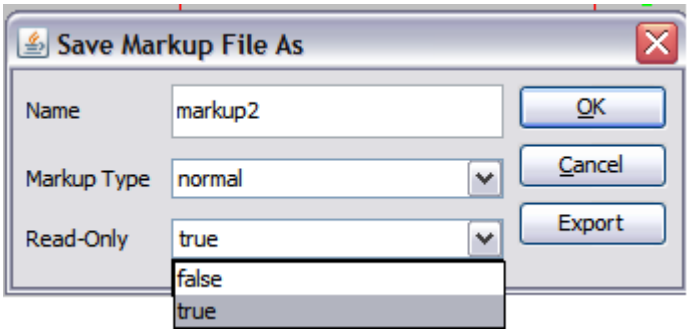


Figure 7-2 Save Markup File as with Read-Only selection

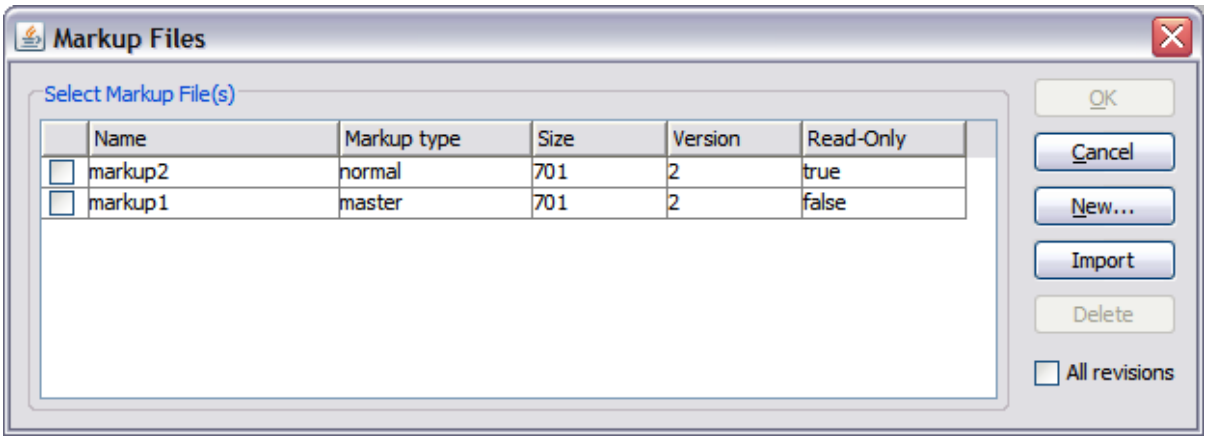


Figure 7-3 Markup Files dialog with Read-Only attribute

In the Sample Integration for FileSys, the Open and Save requests are handled by the `GetPropCSI_Markups` class and `ActionSave` class. In the `GetPropCSI_Markups` class, `buildMarkupGui()` method generates the heading for the Markup Files dialog and displays the GUI for the Save Markup File As dialog. The `buildMarkupProperty()` method loops through each markup and includes the Read-Only attribute.
The following is a code excerpts are of the `buildMarkupGui()` and `buildMarkupProperty()` methods, respectively:

```
private DMSProperty[] buildMarkupGui(FilesysDMSBackend be, …) {
        //For "Markup Files" dialog
        DMSProperty DispArr[] = new DMSProperty[5];
        …
        DispArr[4] = new DMSProperty("Read-Only","6");
        guiProps[1] = new DMSProperty(DMSProperty.PROP_GUI_DISPLAY, DispArr);
        …
        // For "Save Markup File As" dialog
        Property EditArr[] = new Property[3];
        …
        String [] opts = {"false", "true"};
        EditArr[2] = new GUIElementCombo(DMSProperty.CSI_DocReadOnly, "Read-Only", "false",
opts, true);

        guiProps[2] = new DMSProperty(DMSProperty.PROP_GUI_EDIT, EditArr);
         …
}
```

```
private Property[] buildMarkupProperty(FilesysDMSBackend be, …) throws VuelinkException{
        …
        DMSProperty guiProps[] = buildMarkupGui(be, beSession, docID);
        Vector<DocInfo> mrkDocIds = be.dmsListMarkups(beSession, docID);
        DMSProperty markup[] = new DMSProperty[mrkDocIds.size()+1];
        markup[0] = new DMSProperty(DMSProperty.PROP_GUI ,guiProps);
        for (int i = 0; i < mrkDocIds.size(); i++) {
                DMSProperty mrkProp[] = new DMSProperty[7];
                …
                boolean bReadOnly = false;
                …
                // Special treatment for OEVF markups
                if (!editable) { // default asset markup in non-editable mode
                        bReadOnly = true;
                }
                else { // non-oevf markup
                        File file = mrkDocIds.get(i).getFile();
                        if (file.canWrite() == false) {
                                m_logger.info(file.getAbsolutePath() + " is not writable.");
                                bReadOnly = true;
                        }
                }
                …
                mrkProp[5] = new DMSProperty(Property.CSI_DocReadOnly, new
Boolean(bReadOnly).toString());  // This is needed for AutoVue Server
                mrkProp[6] = new DMSProperty("Read-Only", new Boolean(bReadOnly).toString());

                markup[i+1] = new DMSProperty(DMSProperty.PROP_MARKUP,mrkProp);
        }
        m_logger.debug("got the list of markups: " + markup);
        return markup;
}
```

In the `ActionSave` class, the `execute()` method retrieves the Read-Only attribute and passes it to the backend `saveMarkup()` method, as show in the follow code excerpt.

```
public Object execute(final FilesysContext context, …) throws VuelinkException {
        …
        boolean bReadOnly = false; /* true if it is a read-only markup */
        …
        if ( Property.PROP_DOC_READONLY.equals(name) ) {
                try {
                        bReadOnly = prop.getValue().equalsIgnoreCase("true");
                } catch (Exception ex) {
                        bReadOnly = false;
                }
        }
        …
        // saving markup (new or existing)
        newDocID = be.saveMarkup(beSession,baseID, saveID, docName, markType, bReadOnly,
fIn);
        …
}
```

7.15 Implementing Stamps

The Stamp markup entity allows you to create a stamp that includes document and user information (metadata) pulled directly from the DMS backend system.

Stamps are created with the *Design Stamp* tool that is included with the AutoVue installation. Refer to the *Oracle AutoVue User's Manual* for information on how to create an Stamp.

An includes a Stamp definition file (`dmstamps.ini`) and one or more background image files. The Stamp definition file contains information about its background images. The default location for `dmstamps.ini` is located under `<AutoVue Installation Directory>\bin` folder.

After a Stamp is created, the Stamp definition file (`dmstamps.ini`), along with the background images, are stored in a location accessible by your integration application. They may be accessed through files that have absolute path or relative path to your integration application or from documents that have been checked into your backend DMS system. In either case, your integration application should know how to find the Stamp definition file and its background images. You should define the location of dmstamps.ini in web.xml file using `CSI_IntellistampDefLocation` parameter name as in Oracle AutoVue's demo application. If the locations of underlay images are different from those at the designing phase, make sure to modify the paths inside the Stamp definition file (dmstamps.ini).

When adding a Stamp markup entity, the AutoVue server sends a `GetProperties` request by passing the `CSI_IntelliStamp` property in it. The response data that your integration sends back includes the following:

1. The definition file for a Stamp
   - This is basically the content of dmstamps.ini file which is generated by stampdlg.exe tool shipped with AutoVue.

2. The background images for the Stamp
   - This is basically the name and DocID of each of the background images for each Stamp.

The AutoVue server downloads each of the underlying images by invoking the normal file download request and passing the DocID of the stamp image.

The AutoVue server also sends a `GetProperties` request to retrieve DMS attributes defined inside Stamps. These attributes may have values that can be selected from a Pick List. As illustrated in the following image, the Status attribute can be selected from a Pick List that has several values. There are four attributes in the Pick List: **Single Valued and Constrained**, **Single Valued and Non-Constrained**, **Multi Valued and Constrained**, and **Multi Valued and Non-Constrained**. *Constrained* means that the valid value is restricted to the Pick List and *Multi* valued means multiple values can be assigned to an attribute.

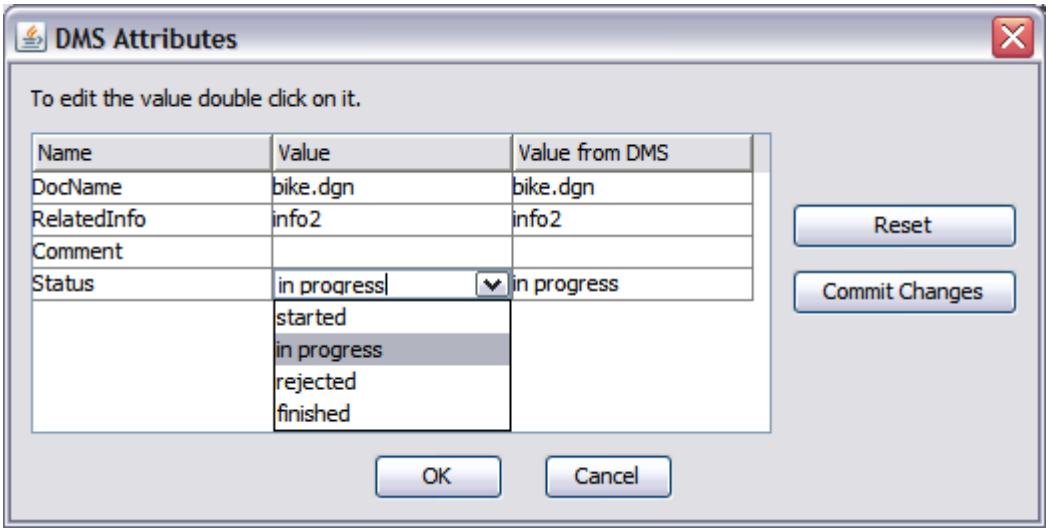


Figure 7-11 Stamp DMS Attributes dialog

After modifying the values of Stamp, the AutoVue server sends a `SetProperties` request to synchronize metadata in the DMS system through the integration interface.

In the sample Integration for FileSys, `GetPropCSI_IntelliStamp` class handles the `GetProperties` request for `CSI_IntelliStamp`, `GetPropDefault` class handles the request for attributes inside Stamps, and `FilesysDMSBackendImp` class has methods to be called from `GetPropDefault` class for the list property.

As illustrated in the following code, the `getImagesDoc()` method of `GetPropCSI_IntelliStamp` class attaches `CSI_DocName` and `CSI_DocID` DMSProperty to `CSI_IntelliStampImage` DMSProperty for each background image file.

```
private DMSProperty[] getImagesDoc(Vector<String> imageFiles){
        if(imageFiles.isEmpty()) return null;

        int numOfImage = imageFiles.size();
        DMSProperty [] images = new DMSProperty[numOfImage];
        for(int i = 0; i < numOfImage; i++){
                …
                DMSProperty [] imagePro= new DMSProperty[2];

                imagePro[0] = new DMSProperty(Property.CSI_DocName,
                        (String)imageFiles.elementAt(i));

                imagePro[1] = new DMSProperty(Property.CSI_DocID,
                        new FilesysDMSDocID (stampImage.getAbsolutePath()).DocID2String());
                images[i] = new DMSProperty(DMSProperty.CSI_IntelliStampImage, imagePro);
                …
        }
        return images;
}
```

Lastly, the `execute()` method of `GetPropCSI_IntelliStamp` class attaches `CSI_IntelliStampDefinition` and `CSI_Intellistamp_Images` DMSProperty in a `CSI_IntelliStamp` DMSProperty and returns it to the VueLink servlet.

```
package com.cimmetry.vuelink.filesys.propactions;
….

public class GetPropCSI_IntelliStamp extends GetFilesysProperty
                                        implements DMSGetPropAction<FilesysContext> {
…
        // Retrieve the content of the Intellistamp Definition file
        String content = getIntelliStampDefinition();

        if(content != null && content.length()!= 0){
                // Construct response CSI_Intellstamp_definition
                DMSProperty ini = new
                        DMSProperty(DMSProperty.CSI_IntelliStampDefinition,content);

                // Retrieve the list of underlying images for the Intellistamp
                Vector<String> imageFiles = getImageFiles(content);

                DMSProperty[]  imagesInfo = getImagesDoc(imageFiles);
                DMSProperty image = null;
                if(imagesInfo != null && imagesInfo.length != 0) {
                        //Construct response CSI_Intellistamp_Images
                        image = new DMSProperty(DMSProperty.CSI_IntelliStampImages,
                                        imagesInfo);
                }
                if(image != null){
                        // Intellistamp Definition file and the underlying images
                        // are all available
                        DMSProperty[] pro = new DMSProperty[2];
                        pro[0] = ini;   // response CSI_Intellstamp_definition
                        pro[1] = image; // response CSI_Intellistamp_Images
                        retProps= new DMSProperty(DMSProperty.CSI_IntelliStamp, pro);
                }else{
                        // Image files are not available
                        retProps= new DMSProperty(DMSProperty.CSI_IntelliStamp, ini);
                }
        }else{
                // Intellistamp Definition file does not exist or is empty
                retProps= new DMSProperty(DMSProperty.CSI_IntelliStamp, "");
        }

        return retProps;
```

The `GetPropCSI_IntelliStamp` class is registered in web.xml as indicated in the following code:

```
<init-param>
    <param-name>dms.getprops.CSI_IntelliStamp</param-name>
    <param-value>com.cimmetry.vuelink.filesys.propactions.GetPropCSI_IntelliStamp</param-
value>
</init-param>
```

When the `GetProperties` request handled by `GetPropDefault` class gets DMS attributes for Stamps, at the last stage, it is handled by calling the `replaceWithPickListIfApplies()` method of the `FilesysDMSBackendImp` class.

```
package com.cimmetry.vuelink.filesys.propactions;
..
public class GetPropDefault implements DMSGetPropAction<FilesysContext>, DMSDefs {
        …
        public DMSProperty execute(…) throws VuelinkException {
                …
                DMSProperty retProp=null;
                …
                retProp =  context.getBackendAPI().replaceWithPickListIfApplies(
                        context.getBackendSession(session, query),retProp);
                return retProp;
        }
}
```

The `replaceWithPickListIfApplies()` method checks to see if the DMS attribute is a Pick List. If it is, the `makePickList()` method is called first and then a different DMSProperty labeled with the property name is returned based on whether the Pick List is single-valued or multi-valued.

```
package com.cimmetry.vuelink.filesys.backend;
…
public DMSProperty replaceWithPickListIfApplies(..){
      try {
            … // Check whether is a PickList
            if (list != null ) { // is Pick List
                …
                if (prop.isSingleValue()) {    //Single valued
                        return new DMSProperty(prop.getName(), prop.getValue(),
                                makePickList(options, constrained, multi));
                } else {    // Multi valued
                        …
                        return new DMSProperty(prop.getName(), prop.getObjectValues(),null,
                                makePickList(options, constrained, multi),multiValue);
                }
            } else {  // no pick list
                return prop;
            }
        } catch (Exception e) {
                …
                return prop;
        }
}
```

The `makePickList()` method builds an array of DMSProperty labeled *`PickValue`* for each available option value in the Pick List and attaches this array and additional info about whether the Pick List is multi-valued and constrained to a DMSProperty labeled *`PickList`* to return.

```
/**
 * create a pick list DMSProperty
 * @param options a Vector<String> of available option list
 * @param constrained if true means options are restricted to the options list
 * @param multiValue if true means multiple items from options list can be selected
 * @return
 */
public DMSProperty[] makePickList(Vector<String> options, boolean constrained, boolean
multiValue) {
        if (options == null || options.size() == 0) {
                return null;
        }
        DMSProperty[] pickValue = new DMSProperty[options.size()];
        for(int k = 0; k < pickValue.length; k++){
                pickValue[k] = new DMSProperty("PickValue", options.get(k));
        }
        Hashtable<String, Boolean> attrs = new Hashtable<String, Boolean>();
        attrs.put(DMSProperty.ATTRIB_CONSTRAINED, constrained);
        attrs.put(DMSProperty.ATTRIB_MULTI_VALUE, multiValue);
        DMSProperty pickList = new DMSProperty("PickList",null, null,pickValue, attrs);
        DMSProperty[] aPL = new DMSProperty [1];
        aPL[0] = pickList;
        return aPL;
}
```

For more information, examine the code and use the debugger to learn more about the real behavior of this class.

7.16 Implementing Markup Policy

Markup policy defines a set of rules to determine certain restrictions and privileges for users of the using AutoVue offline. If no markup policy is defined, a default set of values are used by the AutoVue server.

Usually the definition of markup policy is defined in an xml file (for example, `MarkupPolicy.xml`). It might be stored as a physical file that has absolute path or relative path to the integration application, or be checked as a document into your backend DMS/ERP/PLM/UCM system. In whichever case, your integration should be able to retrieve the content of the defined markup policy. You can define the location of Markup Policy file in your web.xml using `CSI_MarkupPolicyDefLocation` parameter name as in Oracle AutoVue's demo application.

When user uses AutoVue offline, the AutoVue server sends a `GetProperties` request asking for the definition of markup policy by passing `CSI_MarkupPolicy` property within the request. In the sample integration for FileSys, the response to this request is done through `PropCSI_MarkupPolicy` class. The `getMarkupPolicy()` method reads the content of the `MarkupPolicy.xml` file into a string, then the `execute()` method builds a `CSI_MarkupPolicy` DMSProperty with the content of `MarkupPolicy.xml` file. The

way of getting the content of Markup Policy depends on your integration implementation. The following is a code snippet of construct *CSI_MarkupPolicy* DMSProperty:

```
DMSProperty policyProp = null;

String policy = getMarkupPolicy();
if (policy != null) {
        policyProp = new DMSProperty(DMSProperty.CSI_MarkupPolicy ,
                (String[])null, new Object[]{new CData(policy)}, null);
        }
        …
        return policyProp;
}
```

The following is a sample Markup Policy used by the sample integration for FileSys. Refer to the *Oracle AutoVue User's Manual* for information on customizing your own markup policy.

```
<?xml version="1.0" encoding="utf-8"?>
<MarkupPolicy>

<Action name="SaveNewMarkup" default="true">
</Action>

<!-- Only allow owner to modify master markup  -->
<Action name="SaveExistingMarkup" default="true">
  <ExConditions>
      <AndOperator>
        <MarkupFileCondition name="CSI_MarkupType" value="master"/>
        <NotOperator>
           <MarkupFileCondition name="CSI_DocAuthor" value="$CURRENT_USER"/>
        </NotOperator>
      </AndOperator>
  </ExConditions>
</Action>

<Action name="EditMarkup" default="true">
</Action>

<!-- Only allow owner to delete master markups -->
<Action name="DeleteMarkup" default="true">
  <ExConditions>
      <AndOperator>
        <MarkupFileCondition name="CSI_MarkupType" value="master"/>
        <NotOperator>
           <MarkupFileCondition name="CSI_DocAuthor" value="$CURRENT_USER"/>
        </NotOperator>
      </AndOperator>
  </ExConditions>
</Action>

<!-- Only open master markups automatically -->
<Action name="AutoOpenMarkup" default="false">
  <ExConditions>
      <OrOperator>
        <MarkupFileCondition name="CSI_MarkupType" value="master"/>
      </OrOperator>
  </ExConditions>
</Action>

</MarkupPolicy>
```

7.17 Online/Offline Support

AutoVue provides the ability to view and add markups to files in a disconnected environment. Whether you are travelling or need to share files with an external partner, you can still view files and markups, and add new markups. Additionally, when connected, you can update your backend DMS/PLM/ERP/UCM system with edits you make offline.

7.18 Implementing Redirection

In a distributed environment where several remote content servers are used for storing files, ISDK-based integration deployed at a master location (Primary) may redirect the download / upload requests to another ISDK-based integration deployed at remote location (Secondary) where files actually reside. This greatly improves performance since the AutoVue server is installed in the same location as the remote content server.

To deal with this use case, ISDK-based integration adds redirection support when handling `Download` and `Save` requests sent by the AutoVue server.

Handling Redirection for Download

When users view a file, the AutoVue server sends a `Download` request to the primary integration. The primary integration checks whether the file should be picked up from a remote location (that is, a redirection is needed). The way to check this is based on the specific implementation of the backend system that it is integrated for. If redirection is needed, the primary integration sends back a redirection response with a ticket authorizing the AutoVue server to download the file directly from the remote location specified in the response.

In the Demo Integration of Filesys, the `Download` request is handled by `ActionDownload` class. In the method `execute()`, it checks whether a redirection is needed based on whether a `redirectURL` is present in the web.xml and whether a ticket is available in the Authorization block of the `Download` request. If redirection is needed, it constructs the ticket that includes username and password for remote login and calls the `constructRedirectURL()` method in the `DMSUtil` class to generate the redirect response. Usually the ticket is generated by the backend system mechanism as shown in the following code snippet.

```
public Object execute(final FilesysContext context,… ) throws VuelinkException {
        …
        //REDIRECT SUPPORT start based on whether web.xml defines Redirect_VL_URL or not
        String ticket = query.getAuthorization().getTicket();
        if (ticket == null ) { //try to get from session
                ticket = (String)session.getAttribute("Ticket");
        }

        if (ticket == null ) {
            try {
                String redirectURL = FilesysContext.getStaticParameter(
                        FilesysContext.PARAM_CSI_REMOTE_VUELINK);

                //Redirect download if URL is provided
                if (!DMSUtil.isNullOrBlank(redirectURL))  {
                        String username = (String)session.getAttribute("username");
                        String password = (String)session.getAttribute("password");

                        if (username != null && username.length() > 0 && password != null ) {
                                ticket = username.trim()+ "&" + password.trim();
                                m_logger.debug("Ticket: " + ticket);
                        }
                        return DMSUtil.constructRedirectURL(query, redirectURL, ticket);
                }
            }catch (Exception e) {
                m_logger.error("redirecting download faild " + e.toString());
            }
        }
        //REDIRECT SUPPORT finish
        …
}
```

As shown in the following , in the `constructRedirectURL()` method of the `DMSUtil` class, the redirect response encapsulates five properties in a single *`Redirect`* property: HTTP URL as redirection type, ticket authorizing download from remote file cache server, URL to DMS server component located at remote file cache server, the original FILENAME and the Document ID.

```
   public static DMSProperty constructRedirectURL(final DMSQuery  query, String redirectURL,
String ticket){

  DMSProperty [] redirect = new DMSProperty[5];

  redirect[0] = new DMSProperty(DMSProperty.TYPE, DMSProperty.URL);
  redirect[1] = new DMSProperty(DMSProperty.TICKET, ticket);
  redirect[2] = new DMSProperty(DMSProperty.SERVER, redirectURL);
  redirect[3] = new DMSProperty(DMSProperty.ORIGINALURL, query.getOriginalURL());
  String docID = query.getDocID();

  /* if  no docID is returned to AV Server, AV server won't send
   * redirect request to VL at the remote content server.
   * so return one faked docID to AV Server when saving a new markup.
   * */
  if(docID == null || docID.length() == 0){
   docID = "docID";
  }
  redirect[4] = new DMSProperty(DMSProperty.CSI_DocID, docID);

  return new DMSProperty(DMSProperty.REDIRECT, redirect);
 }
```

After the AutoVue server receives this redirection response, it issues another `Download` request with the ticket information directly to the secondary integration deployed at the remote location. The secondary integration uses the ticket to log-in to the remote backend system and download the file as usual.

Handling Redirection for Save

When users want to save a file, prior to the `Save` request, the AutoVue server sends a `GetProperties` request with `CSI_Redirected` property to the primary integration asking whether redirection is supported. If supported, the primary integration responds `TRUE` for this `CSI_Redirected` property.

If `TRUE` is returned, the AutoVue server sends a `Save` request to the primary integration without file content. If the primary integration checks that redirection is needed, it sends a *`REDIRECT`* DMSProperty response similar to that for the `Download` request above with a ticket authorizing the AutoVue server to upload the file directly to another location specified in the redirection response.

Upon receiving the ticket, the AutoVue server then sends a second `Save` request to the secondary integration located at the remote location by adding the ticket to the Authorization block of the request and attaches the file to be saved.

Once the uploaded file is checked in successfully, the secondary integration returns a confirmation in the form of a receipt in place of the returned DocID.
The AutoVue server then issues a third `Save` request forwarding this receipt to the primary integration again. Primary integration then returns the DocID of the uploaded file to finalize the `Save` process.

The following are code snippets from the Demo Integration of Filesys.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class GetPropDefault …{
        …
        public DMSProperty execute(final FilesysContext context,…) throws VuelinkException {
                …
                final String propName = property.getName();
                …
                if (DMSProperty.CSI_Redirected.equals(propName)) {
                        String redirectURL = FilesysContext.getStaticParameter(
                                FilesysContext.PARAM_CSI_REMOTE_VUELINK);
                        boolean redirected = false;
                        if(redirectURL != null && redirectURL.length()  > 0){
                                redirected = true;
                        }

                        return  new DMSProperty(DMSProperty.CSI_Redirected,
                                        new Boolean(redirected));
                }
                …
        }
        …
}
```

Note that the `GetPropDefault` class handles responses to `GetProperties` requests for `CSI_Redirected` DMSProperty. Filesys decides that redirection is supported if `PARAM_CSI_REMOTE_VUELINK` is defined in the web.xml file. In your integration, it should be decided based on communication with the backend system.

`ActionSave` class handles the response to `Save` request from the AutoVue server. The `execute()` method checks whether redirection is involved.

- If redirection is involved, it checks whether it has Receipt property in the request.
    - If not, it means that this is the first `Save` request and it generates a ticket for remote login and responses back with a Redirect property similar to that for handling `Download` request.
    - If yes, it means that this is the third `Save` request and it responds with a *`CSI_DocID`* DMSProperty.

- If no redirection is involved, this can be the second `Save` request at the secondary site or a normal `Save` request at the primary site. In either case, it should check in the file. If the check-in happens at the secondary location, a receipt for the saved file is returned; if the check-in happens at the primary location, a valid DocID for the saved file is returned.

```
package com.cimmetry.vuelink.filesys.actions;
…
public class ActionSave …     {
        …
    public Object execute(final FilesysContext context,…) throws VuelinkException {
        …
        // REDIRECT SUPPORT start based on whether web.xml defines Redirect_VL_URL or not
        String ticket = (String)session.getAttribute("Ticket");
        if (ticket == null && args == null) {
                try {
                        String redirectURL = FilesysContext.getStaticParameter(
                                FilesysContext.PARAM_CSI_REMOTE_VUELINK);

                        if (!DMSUtil.isNullOrBlank(redirectURL)) {
                                // Redirect is involved
                                String receipt = getReceipt(props);
                                if (receipt != null && receipt.length()!= 0) {
                                        // There is Receipt in the Save request
                                        return new DMSProperty(Property.CSI_DocID, receipt);
                                }
                                … //Generate ticket using username and password
                                m_logger.debug("Ticket: " + ticket);
                                return DMSUtil.constructRedirectURL(query, redirectURL,
                                        ticket);
                        }
                }catch (Exception e) {
                        m_logger.error("redirecting save failed " + e.toString());
                }
        }
        …

        // The following is psuedo code
        if checkin file at the redirected site {
          String receipt = secondaryCheckIn(); // Save file
          return receipt;
        }
        else {
          String docID = primaryCheckIn();     // Save file
          return docID;
        }
    }
    …
}
```

7.19 Implementing Real-Time Collaboration and Meeting Management

Oracle AutoVue provides real-time collaboration functionality that enables multiple users to review files interactively and simultaneously. ISDK-based integration can integrate AutoVue Real-Time Collaboration (RTC) functionality with third-party meeting management systems.

The steps for RTC and meeting management integrations include:

- Customizing UI to provide links for launching AutoVue in RTC mode by hosts and guests
- Implementing ISDK APIs for handling backend communication

7.19.1 Launching AutoVue in RTC Mode

When creating a third-party AutoVue RTC meeting, the meeting creators (hosts) can invite a list of attendees (guests) to attend the meeting and add list of documents to review during the meeting. From the third-party meeting management GUI, hosts can click to start the meeting that launches AutoVue, displays a meeting document, enters RTC mode, and presents a default collaboration markup. Guests can click to join a meeting which then launches AutoVue into RTC mode and are presented in the same AutoVue GUI as that on the host side.

7.19.2 Hosts Initiate RTC

The following information is needed for hosts to launch AutoVue to initiate a RTC.

- `DMS` is the URL for the DMS servlet (main class) of your ISDK-based integration
- `MEETINGID` is a number identifying the RTC meeting and holds the same value as `CSI_ClbSessionID` mentioned below.
- `CSI_ClbSessionData` can hold more information in addition to the `CSI_ClbSessionID` (`MEETINGID`), but your integration should know how to parse the `CSI_ClbSessionData` to retrieve the `CSI_ClbSessionID`.
- `CLBUSERS` are comma separated strings that represent the list of attendees who have been invited to the RTC by AutoVue.
  Note: This value is not supported by the current AutoVue server.
- `FILENAME` is a file among the list of documents intended to be reviewed during the RTC meeting.

```
var session = "CSI_ClbDMS=" + DMS + ";" +
        "CSI_ClbSessionData=" + MEETINGID + ";" +
        "CSI_ClbSessionSubject=DemoRealTimeCollaboration;" +
        "CSI_ClbSessionType=public;" +
        "CSI_ClbUsers=" + CLBUSERS + ";";
```

When creating an AutoVue applet to initiate a RTC for the first time, the following parameters should be provided:

```
        <PARAM NAME="FILENAME" VALUE="' + FILENAME +'">
        <PARAM NAME="COLLABORATION" VALUE="INIT:' + session + '">
```

When reusing an AutoVue applet for RTC, the following needs to be set using AutoVue applet APIs. The `FILENAME` is the new file to collaborate on.

```
        japplet.setFileThreaded(FILENAME);
        japplet.collaborationInit(session);
```

7.19.3 Guests Join RTC

The following information is needed for guests to launch AutoVue to join a RTC. Note that only `CSI_ClbDMS` and `CSI_ClbSessionData` are needed.

```
var session = "CSI_ClbDMS=" + DMS + ";" +
        "CSI_ClbSessionData=" + MEETINGID + ";";
```

When creating an AutoVue applet for joining a RTC for the first time, the `COLLABORATION` parameter should be provided. There is no need for `FILENAME` parameter.

```
        <PARAM NAME="COLLABORATION" VALUE="INIT:' + session + '">
```

Refer to `RTCDemo.jsp`, `RTCDemo_init.jsp` and `RTCDemo_join.jsp` in the RTC Sample for detailed implementation.

7.19.4 ISDK APIs for RTC

To support RTC, ISDK-based integration needs to support a series of requests sent by AutoVue Server.

When the host initiates a RTC meeting or guests join a RTC, the first request sent by the AutoVue server is `CSI_ClbSessionID`. In response, the integration retrieves the session ID by passing `CSI_ClbSessionData` sent in the request. In the Sample Integration for Filesys, this request is handled by `GetPropCSI_ClbSessionID` class. In Filesys, `ClbSessionData` simply comprises `ClbSessionID`. Here is the sample code.

```
public class GetPropCSI_ClbSessionID … {
  …
  public DMSProperty execute(…) throws VuelinkException {
        …
        String sClbSessionData = query.getClbSessionData();
        m_logger.debug("ClbSessionData : " + sClbSessionData);
        String sClbSessionID = sClbSessionData;
        m_logger.debug("ClbSessionID : " + sClbSessionID);
        return new DMSProperty(DMSProperty.CSI_ClbSessionID, sClbSessionID);
  }
}
```

One important request sent by the AutoVue server for Real-Time Collaboration is the `GetProperties` request for `CSI_Collaboration` property. When users select an AutoVue Collaboration action `Invite`, `Session Information`, or `Close Collaboration Session`, the AutoVue server sends this request to retrieve information. ISDK integration responds with a single *`CSI_Collaboration`* DMSProperty that includes the following DMSProperties:

- `PROP_GUI`: DMSProperty with an array of children:
  - `PROP_GUI_DISPLAYOPTS`: DMSProperty having multiple child DMSProperties for enabling/disabling GUI items in the Invitation dialog.
  - `PROP_GUI_DISPLAY`: DMSProperty having multiple child DMSProperties listing attributes to be displayed in the Session selection dialog along with the width (number of characters) to reserve for the attributes display.
  - `CSI_ClbInvitation`: DMSProperty wrapping a `List` identified as `CSI_ClbUsers`: These users are displayed in the left side of AutoVue Collaboration's Invitation dialog. Users on this list can be invited to attend a RTC by AutoVue.
    Note: This value is not supported by the current AutoVue server.
  - `CSI_ClbUsers`: DMSProperty listing users that have already been invited to a RTC. The list of users will be shown in the `User` section of the AutoVue Collaboration Session Information dialog.
    Note: This value is not supported by the current the AutoVue Server.
  - `CSI_ClbSession`: DMSProperty having multiple child DMSProperties that show session information such as session title, id, type, subject, duration, start time, and so on. It includes also a `CSI_ClbSaveChat` indicates whether the backend system component supports saving chat transcript. By default, `CSI_ClbSaveChat` is set to `FALSE`.

In the Sample Integration for Filesys, `CSI_Collaboration` is handled by `GetPropCSI_Collaboration` class. The `buildProperty()` method in this class is responsible for generating the `CSI_Collaboration` DMSProperty to return.

```
private DMSProperty buildProperty(…) throws VuelinkException {
        DMSProperty clbProps[] = new DMSProperty[3];

        /* GUI section */
        DMSProperty guiProps[] = new DMSProperty[3];

        /* DisplayOptions sub-section */
        // For enabling/disabling GUI items in Invitation dialog
        DMSProperty dispOptArr[] = new DMSProperty[4];
        dispOptArr[0] = new DMSProperty("AllowAdd","true");
        dispOptArr[1] = new DMSProperty("AllowAddNew","true");
        dispOptArr[2] = new DMSProperty("AllowRemove","true");
        dispOptArr[3] = new DMSProperty("AllowLayerColor","true");
        guiProps[0] = new DMSProperty(DMSProperty.PROP_GUI_DISPLAYOPTS, dispOptArr);

        // Lists the attributes to be displayed in the Session selection dialog
        // along with the width to reserve for the attributes display.
        // The property names match those defined in the following "Session" Section.
        // For example, dispArr[i] = new DMSProperty(attr_name, attr_width);
        DMSProperty[] dispArr = new DMSProperty[2];
        dispArr[0] = new DMSProperty("Originator", "14");
        dispArr[1] = new DMSProperty("Meeting Duration", "14");
        guiProps[1] = new DMSProperty(DMSProperty.PROP_GUI_DISPLAY, dispArr);

        /* Invitation sub-section */
        // Lists users who can be invited to the collaboration session
        Property[] invitationArr = new Property[1];
        String defaultUser = null;
        String[] users = null;
        boolean readOnly = false;
        users = be.clbUsers(beSession, sClbSessionID);
        invitationArr[0] = new GUIElementList(DMSProperty.CSI_ClbUsers, "user",
                        defaultUser, users, readOnly);
        guiProps[2] = new DMSProperty(DMSProperty.CSI_ClbInvitation, invitationArr);

        clbProps[0] = new DMSProperty(DMSProperty.PROP_GUI, guiProps);
        /* End of GUI section */

        /* ClbUser Section */
        String[] invitedUsers = be.clbInvitedUsers(beSession, sClbSessionID);
        clbProps[1] = new DMSProperty(DMSProperty.CSI_ClbUsers, invitedUsers);

        /* Session Section */
        // The current collaboration session information:
        Vector<DMSProperty> sessionAttr = new Vector<DMSProperty>();
        sessionAttr.add(new DMSProperty(DMSProperty.CSI_ClbSessionID, sClbSessionID));
        String sClbSessionType = "public";  // It can be private
        sessionAttr.add(new DMSProperty(DMSProperty.CSI_ClbSessionType, sClbSessionType));
        String sClbSaveChat = "true";
        sessionAttr.add(new DMSProperty(DMSProperty.CSI_ClbSaveChat, sClbSaveChat));
        // Here are sample meeting attributes for RTC demo. In real implementation,
        // they might be retrieved from the backend system.
        sessionAttr.add(new DMSProperty("Originator", "rtc"));
        sessionAttr.add(new DMSProperty("Meeting Duration", "60 minutes"));
        DMSProperty[] arr = new DMSProperty[sessionAttr.size()];
        clbProps[2] = new DMSProperty(DMSProperty.CSI_ClbSession,
                sessionAttr.toArray(arr));

        /* End of all sections */
        m_logger.debug("Got the Collaboration GUI elements: " + clbProps);

        return new DMSProperty(DMSProperty.CSI_Collaboration, clbProps);
}
```

Based on the above implementation, during a RTC meeting, the collaboration's Session Information dialog is similar to the following figure.

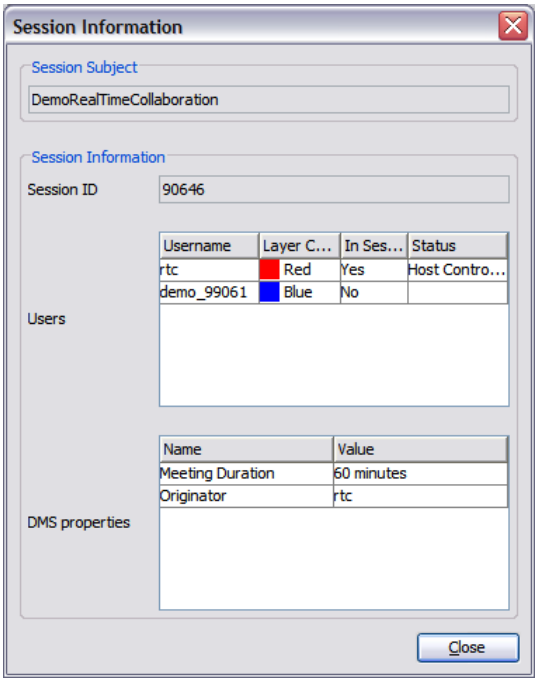


Figure 6-12 Session Information dialog

If `CSI_ClbSaveChat` is set to `true` in your integration's `CSI_Collaboration` response, and when the RTC is closed by selecting **Collaboration** from the **Close Collaboration Session** menu of the AutoVue GUI, the AutoVue server sends a Save request with `CSI_ClbDocType` property set to value `chat` and attaches the chat content during the RTC session for saving. In this case, ISDK integration communicates with the backend system to save the chat transcript at a desired location. In the Sample Integration for Filesys, `ActionSave` class handles the saving of the chat transcript.

```
public class ActionSave … { …
  public Object execute(…) throws VuelinkException { …
   final Property[] props = query.getDMSArgsProperties(); …
   boolean bSaveChat = false; …
   if (props != null) {
      for (int i = 0; i < props.length; i++) { …
          if (Property.CSI_ClbDocType.equals(name)) {
              if (prop.getValue() != null && prop.getValue().equalsIgnoreCase("chat")) {
                    bSaveChat = true;
              }
          } …
    } …
   InputStream fIn = null;
   … // Put chat file content in a fIn
   …
   if (bSaveChat) {
       String clbData = query.getClbSessionData();
       String clbSessionID = clbData;
       docName = context.getInitParameter("RootDir") + File.separator + "chat_" +
                 clbSessionID + ".txt";
       m_logger.debug(" for session " + clbSessionID + " to: " + docName);
       return be.saveChat(beSession, docName, fIn);
   } …
  } …
}
```

During a RTC session, the AutoVue server sends notifications as a part of `SetProperties` request to notify that certain actions have been completed by AutoVue. For example, a notification is send if a RTC is initialized or closed, users join or leave a session, or when a new file is opened in which to collaborate. These actions correspond to `CSI_ClbInitSession, CSI_ClbCloseSession, CSI_UserJoined, CSI_UserLeft` and `CSI_DocumentSet` property.

In the Sample Integration for Filesys, these notifications are handled by the `ActionSetProperties` class that mainly generates debug information when receiving these notifications. In the case of a `CSI_DocumentSet` notification is sent when collaboration users switch documents to collaborate on in the middle of a RTC meeting, Filesys adds newly viewed document information to a text file, meetingfiles.txt, that holds all the meeting documents information by `clbDocumentSet()` method in `FilesysDMSBackendImp` class.

```
public class ActionSetProperties … { …
  public Object execute(…) throws VuelinkException { …
    Property[] props = query.getProperties(); …
    for (int i = 0; i < props.length; i++) {
       String value = props[i].getValue(); …
       if (props[i].getName().equalsIgnoreCase(Property.CSI_ClbCloseSession)) {
              m_logger.debug("CSI_ClbCloseSession : " + sClbSessionID);
              continue;
       }
       if (props[i].getName().equalsIgnoreCase(Property.CSI_ClbInitSession)) {
              m_logger.debug("CSI_ClbInitSession : " + value);
              continue;
       }
       if (props[i].getName().equalsIgnoreCase(Property.CSI_UserJoined)) {
              m_logger.debug("User Joined : " + value);
              continue;
       }
       if (props[i].getName().equalsIgnoreCase(Property.CSI_UserLeft)) {
              m_logger.debug("User Left : " + value);
              continue;
       }
       if (props[i].getName().equalsIgnoreCase(Property.CSI_DocumentSet)) {
              m_logger.debug("Document Set = " + value);
              context.getBackendAPI().clbDocumentSet(
                     context.getBackendSession(session, query), value);
              continue;
       } …
    } …
  } …
}
```

During a RTC meeting, the document to be reviewed can be changed by using **DMS Browse** or **DMS Search**. From the AutoVue menu bar, select **File**, **Open URL**, and then **DMS Browse**. The File Open dialog appears. The AutoVue server sends `CSI_ListItems` request to retrieve the `DMS Browse` result. As a part of handing this request, ISDK-based integration should allow users to browse the list of documents to be reviewed during the meeting by communicating with the backend system to retrieve the list. In Filesys-based RTC demo, a text file meetingfiles.txt holds the list of meeting documents; `GetPropCSI_ListItems` class handles the response for `CSI_ListItems` request and it finally calls the `getInstanceIDs()` method in `Browse` class to retrieve this list. In the sample dialog from Filesys shown below, three meeting documents are listed under the Meeting folder. Meeting

users can select documents listed under Meeting to review. Additionally, they can open documents under other folders to review. When a new document is opened, a `CSI_DocumentSet` notification is sent by AutoVue Server. By handling this notification, ISDK-based integration can add the new document information to the existing meeting document list by communicating with the backend system.

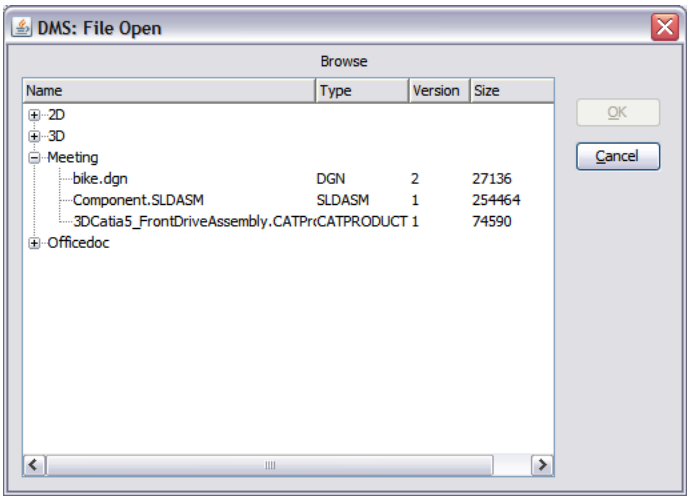


Figure 6-13 DMS: File Open dialog

```
package com.cimmetry.vuelink.filesys.dms.domain;

class Browse{ …
  public Vector<DocInfo> getInstanceIDs() throws Exception{
        Vector<DocInfo> v = new Vector<DocInfo>();
        File browseFile = (File)m_docID.getFile();

        … //list meeting files from "Meeting/meetingfile.txt"
        … //list elements from other folders
}
```

```
public class FilesysDMSBackendImp … { …
  public void clbDocumentSet(DMSBackendSession session, String sDocID) … { …
      if (sDocID == null) return;
      … // Add document name to the meetingfiles.txt if this is a new document
  }
  …
}
```

7.19.5 Summary

In order to support RTC and meeting management, ISDK-based integration should be able to gather information and launch the AutoVue applet to enter Real-Time Collaboration mode when the host starts a meeting and when guests join meeting from a third-party meeting management system. The integration should implement responses to `CSI_CollaborationID`, `CSI_Collaboration`, `SetProperties` with different notifications, `CSI_ListItems` and Save requests sent by the AutoVue server to handle RTC-specific tasks. To accomplish the above response, it needs to communicate with the backend system to perform the following:

- Retrieve a list of users who have been added to the meeting's attendees when hosts create a meeting from the third-party meeting management system.
- Retrieve a list of documents to collaborate on when hosts create a meeting from the third-party meeting management system.
- Save chat content during RTC when hosts close a collaboration session.
- Perform additional processes for notification messages. For example, user joined and left, document change, collaboration session initialized and closed.

7.20 Implementing Oracle Enterprise Visual Framework Support

Oracle Enterprise Visual Framework (OEVF) is designed to add Enterprise Visualization capabilities to enterprise applications and to provide a generic structure for accessing documents stored in the backend system through the concepts of *Asset ID* and *Workflow ID*. Both concepts are defined in enterprise application systems rather than the document ID of the document in the backend system. Asset ID and Workflow ID are unique identifiers associated to an asset and an enterprise workflow (such as a service request or work order), respectively, in the backend system.

Using internal mapping, a document in the backend system can be connected to multiple assets and/or multiple workflows in an enterprise application system so that the document can be retrieved using the Asset/Workflow IDs. Usually the mapping is stored as part of the document's record inside the backend system. As a result, some custom attributes should be added to the backend system. For example, the document's record can have an OEVF AssetID attribute that holds a set of AssetIDs and an OEVF WorkflowID attribute that holds a set of WorkflowIDs. In the Sample Integration for Filesys, this relationship is represented in an XML file.

In OEVF, each AssetID/WorkflowID can be associated with its own set of asset/workflow markups and each has a default asset/workflow markup. The markups viewable in AutoVue should be in the context of the certain asset and/or workflow.

7.20.1 Most Common Use Cases for OEVF

1 Administrator logs into Enterprise Application:
   - Administrator navigates to **Asset** info page and chooses to **Edit** asset using AutoVue.
   - AutoVue applet opens the associated file either as popup or embedded.
   - AutoVue automatically opens the asset markup if already exists or creates a new one if does not exist:
     - Administrator is able to modify and save the asset markup.
     - Administrator is not able to rename asset markup.
     - Administrator is not able to open or save any markups other than asset markup.
     - Only one asset markup is allowed per asset.

2 End user logs into Enterprise Application:
   - User navigates to **Asset** info page and selects to **View** asset using AutoVue.
   - AutoVue applet opens the associated file either as popup or embedded.
   - AutoVue will automatically open the asset markup if exists:
     - User will not be able to modify or save the asset markup.
     - User will not be able to create/open/delete any markups.

3 End user logs into Enterprise Application:

  - User navigates to **workflow** info page and selects to **View** the related asset in AutoVue.
  - AutoVue applet opens the asset file and asset markup if exists (read only) and default workflow markup (edit mode).
  - When user tries to list the markups, only markups related to given asset and workflow are listed.
  - User can open, edit and save any workflow related markup.

- In all use cases if file has any XRefs they are loaded.
- User can view and/or include UCM Properties of document in print output.

7.20.2 OEVF Launching URL and Parameters

The OEVF launching URL is dynamically constructed and invoked from the enterprise application system to launch AutoVue within the context of an asset or a workflow. This URL passes OEVF parameters to a customized page (as with `frmApplet.jsp` in the following OEVF launching URL sample) that is part of your ISDK customization component on the enterprise application side.

The following parameters can be passed in an OEVF launching URL:

Asset ID: ID that uniquely identifies an Asset in the enterprise application system that has been mapped to a document in the backend system through the OEVF Asset ID attribute.

Workflow ID: ID that uniquely identifies a Workflow in the enterprise application system that has been mapped to a document in the backend system through the OEVF Workflow ID attribute. When launching OEVF with Workflow ID, if the default workflow markup does not exist, it is created automatically and markup entities can be added to it. Besides the default workflow markup, any number of workflow markups can be created for the workflow represented by the Workflow ID. The default workflow markup cannot be deleted. However, other workflow markups can be deleted. If Workflow ID parameter is present, then the EditMode parameter is ignored**.**

- **Embedded (optional):** This parameter decides if the AutoVue applet should be launched in the same window (`Embedded` =1) or a new window (`Embedded` is not equal to 1 or not presented).
- **GuiFile (optional):** This parameter decides if the default GUI should be overwritten. If not present, the default GUI is use. Otherwise the value indicated by this parameter (the name of the GUI file) is used by AutoVue for the applet interface. The actual GUI file is located in the AutoVue server's work directory.

Document ID (optional): This is the document ID that is mapped to Assets or Workflows. This option is needed only when a particular revision of a document is required (for example, if a workflow is mapped to earlier version of an Asset). If the document ID is not provided, then the Asset ID is used to locate the latest revision of the mapped document.

EditMode: If `EditMode=1`, AutoVue applet is launched in Asset Editing mode and a customized `assetEdit.gui` should be in use. This overrides the `GuiFile` parameter.

In this mode, the asset markup can be edited, modified and saved to the backend system. The **New Markup** and **Save As** options should be disabled in this mode. If the asset markup exists, it loads automatically. If no asset markup exists, it is created automatically and markup entities can be added to it. Asset markup is unique per asset and cannot be deleted.
If `EditMode=0` or not presented, `assetView.gui` will be in use. The asset markup, if it exists, loads automatically in read-only mode and no activity related to markup can be performed except viewing it.
The assetEdit.gui and assetView.gui files should be put in the <jVue_home>/bin/Profiles directory.
Edit Mode parameter control only the behavior of Asset markup.

The following are some OEVF launching URL samples assuming that the backend system is deployed on a Web server with `appserver:port` and if `frmApplet.jsp` is the customized page responsible for constructing and launching AutoVue applet.

http://appserver:port/jsp/frmApplet.jsp?aID=<Asset ID>&EditMode=<Mode>&embedded=<Option>
http://appserver:port/jsp/frmApplet.jsp?wID=<Workflow ID>&guiFile=<CustomizedGuiFilename>
http://appserver:port/jsp/frmApplet.jsp?wID=<Workflow ID>&aID=<AssetID>
http://appserver:port/jsp/frmApplet.jsp?dID=<Document ID>&aID=<AssetID>
http://appserver:port/jsp/frmApplet.jsp?dID=<Document ID>&wID=<WorkflowID>

7.20.3 OEVF Customization Page

As mentioned above, a customized page residing on the enterprise application side is responsible for constructing and launching the AutoVue applet in the context of an OEVF object. The frmApplet.jsp file in the Demo Integration for Filesys is serves this purpose. The code excerpt from frmApplet.jsp that is related to OEVF is shown in the following figure.
Note: The special case of Filesys, the variable DocID in frmApplet.jsp is used only for constructing OEVF document ID to open a file in AutoVue and the file to be opened for non-OEVF is passed in by calling `setFile()` of frmApplet.jsp in `ListDirServlet` class when users browse the filesys file tree structure. Generally, your ISDK implementation might need to handle both cases in the same jsp file using the document ID passed in the URL parameter. Refer to `CreateReusableApplet()` function of `frmApplet.jsp` for code sample to set the GUI file and OEVF document ID in order to reuse the pop up AutoVue applet.

```
<%@ page … %>
<%
        …
        String docID = request.getParameter("docID");
        String assetID = request.getParameter("aID");
        String workflowID = request.getParameter("wID");
        …
        String sEmbedded = request.getParameter("embedded");
        boolean embedded = true;  // By default, launch AutoVue applet in embedded mode
        if (sEmbedded != null && sEmbedded.length() > 0 &&
                sEmbedded.equalsIgnoreCase("0")) {
                embedded = false;  // Launch AutoVue applet in pop up window
        }
         …
        String guiFile = request.getParameter("guiFile");
        String DocID = null;  // OEVF DocID
        if((assetID != null && assetID.length() > 0)||(workflowID != null &&
                workflowID.length() > 0)){
                DocID = "oevf://dID=" + docID+ "&aID=" + assetID + "&wID=" + workflowID;
        }

        String EditMode = request.getParameter("EditMode");
        if ( (assetID != null && assetID.length()>0 )
                && (workflowID == null || workflowID.length()<1) ) {
                if ("1".equalsIgnoreCase(EditMode)) {
                        guiFile="assetEdit.gui";
                } else {
                        guiFile="assetView.gui";
                }
        }

        if(EditMode != null && EditMode.length() > 0){
           DocID +="&EditMode=" + EditMode;
        }
%>

<html> <head> …
<script> <!--
…
var DOCUMENT_ID = '<%=DocID%>'; // OEVF Document ID
var GUIFILE = '<%=guiFile%>';
var EMBEDDED = '<%=embedded%>';
…
// -->
</script></head>
<body>
<script language="JavaScript">
<!--
    …
    var jvapp = '<HTML>…' + … +
        '\n<PARAM NAME="EMBEDDED" VALUE="TRUE">' + …;

        if (DOCUMENT_ID != 'null') {
                jvapp +=  '\n<PARAM NAME="FILENAME" VALUE="' + DOCUMENT_ID + '">';
        }
        if (GUIFILE != 'null') {
                jvapp += "\n<PARAM NAME=\"GUIFILE\" VALUE=\"" + GUIFILE + "\">";
        } else …
        …
        if (EMBEDDED == 'true' ) {
                CreateApplet();  // Create embedded AutoVue applet
        } else {
                if ( validatePopups() == true) {
                        CreateReusableApplet();  // Create pop up AutoVue applet
                } …
        }
    …
// end script hiding from old browsers -->
</script>
</body></html>
```

7.20.4 ISDK APIs for OEVF

To support OEVF, ISDK needs to enhance its implementation corresponding to the AutoVue server's Open, Save, Delete requests and `GetProperties` request for *`CSI_Markups`*. These requests are handled by classes `ActionOpen`, `ActionSave`, `ActionDelete` and `GetPropCSI_Markups` in the Sample Integration for Filesys.

7.20.4.1 ActionOpen

When opening a document, ISDK needs to distinguish between the OEVF cases, regular cases, and constructs to return a unique DocID for the document to open.

In the case of OEVF involvement, if the OEVF launching URL only has an AssetID and/or Workflow ID without Document ID, then ISDK communicates with the backend system to find out the Document ID to which the Asset ID and/or Workflow ID is connected to and check the consistency, if necessary. If there is a Document ID passed in addition to AssetID and/or Workflow ID, then IDSK needs to verify the consistency.

In the Sample Integration for Filesys, `ActionOpen` class calls `openFile()` method of `FilesysDMSBackendImp` class to get the Document ID and the `openFile()` calls `findByOEVF()` method of `FilesysDMSFacade` class to parse the mapping of Document IDs between Asset IDs and Workflow IDs. Your ISDK should communicate with your backend system to find the Document ID.

```
public DocID openFile(… )…{
        …
        FilesysDMSDocID docID = null;
        String origURL = params.get("origURL");
        String aID = DMSUtil.getAssetID(origURL);    // Get Asset ID parameter
        String wID = DMSUtil.getWorkflowID(origURL); // Get Workflow ID parameter
        String dID = DMSUtil.getUrlValue(origURL, "dID");  // Get Document ID parameter
        String relPath = null; …
        if (!DMSUtil.isNullOrBlank(aID)) { // If Asset ID parameter presents
                // Find OEVF document using Asset ID
                String filePath = m_filesysInfo.findByOEVF(dID, aID, ASSETID);
                … // Find out OEVF file real path or return error message if not found
        } else if(!DMSUtil.isNullOrBlank(wID)){ // If Workflow ID parameter presents
                // Find OEVF document using Workflow ID
                String filePath = m_filesysInfo.findByOEVF(dID, wID, WORKFLOWID);
                … // Find out OEVF file real path or return error message if not found
        }
        …
        // Construct Filesys DocID to return
        docID = new FilesysDMSDocID(relPath, null, version,aID, wID);
        return docID;
}
```

```
public String findByOEVF(String dID, String oevfID, String oevfField){
        try{
                return OevfParser.parseOevfXml(dID, oevfID,oevfField);
        }catch(Exception e){
                m_logger.error("Failed to parse OEVF info xml . " + e.getMessage());
        }
        return null;
}
```

7.20.4.2 GetPropCSI_Markups

The `GetPropCSI_Markups` implementation to handle `GetProperties` request for *`CSI_Markups`* will be enhanced to add the following functionalities.

- It handles Asset Edit mode by loading asset markups as a master and editable markup if in Asset Edit mode, or as master and read-only markup in other cases.
    - In Asset Edit mode, it generates an empty asset markup if such a markup does not exist for the given Asset ID before returning markup list. It loads the default asset markup as a master markup.

- It will generate an empty default workflow markup if such a markup does not exist for the given Workflow ID before returning markup list. It loads the default workflow markup and all existing workflow markups as a master and editable markup all the time.

- It only list markups related to the given Asset and Workflow IDs. No other markup can be listed. If both Asset and Workflow IDs are given, the default Workflow markup opens after the Asset markup is opened (make the former the active one).

The following code is extracted from the `GetPropCSI_Markups` class of the Sample Integration of Filesys. It treats asset and workflow markups as master markups and checks whether they should be read-only. The `getInstanceIDs()` method in `Markup` class is responsible for retrieving the markup list that includes only OEVF markups in the context and, if needed, creates default asset and workflow markups. The default markup is created by copying an existing empty markup `BlankMarkup.mrk` distributed with the Sample Integration of Filesys. Your ISDK integration can make use of it also.

```
public class GetPropCSI_Markups … { …
    private Property[] buildMarkupProperty(FilesysDMSBackend be,…) … { …
        //Gets the list of markups from the DMS
        Vector<DocInfo> mrkDocIds = be.dmsListMarkups(beSession, docID);
        …
        for (int i = 0; i < mrkDocIds.size(); i++) {
            // Treat asset markup as master and read-only if not in Asset Edit mode
            // Treat workflow markup as master markup and editable all the time
            boolean bReadOnly = false;
            boolean editable = true;
            if ( … ) { // If markup is Asset or Workflow markup
                mrkType = "master";
                String oevfType = … ;  // Get OEVF markup type
                if (oevfType.equalsIgnoreCase(Markup.ASSETS)){ // If asset markup
                   if (!(Boolean)beSession.getAttribute("EditMode") // Not Asset EditMode
                   || ((Boolean)beSession.getAttribute("EditMode") &&
                        !DMSUtil.isNullOrBlank(docID.getWorkflowID()))
                   ){
                      editable = false;
                   }
                }
            }
            if (!editable) { // default asset markup in non-editable mode
                bReadOnly = true;
            } …
        } …
    } …
}
```

```
public class Markup{ …
   public Vector<DocInfo> getInstanceIDs(DMSSession session) … { …
        String aID = m_docID.getAssetID();
        String wID = m_docID.getWorkflowID();
        Vector<DocInfo> vector = new Vector<DocInfo>();
        if(DMSUtil.isNullOrBlank(aID) && DMSUtil.isNullOrBlank(wID)){
           … // Non OEFV handling
        } else{
           if (!DMSUtil.isNullOrBlank(aID)){
                File assetMarkup[] = …; // Retrieve asset markup
                if (assetMarkup != null && assetMarkup.length > 0){ // Asset markup exists
                   … // There should only be one asset markup for a given asset ID.
                   … // Add the asset markup to the return vector
                } else  // Asset markup does not exist
                        if ((Boolean)session.getAttribute("EditMode") &&
                             DMSUtil.isNullOrBlank(wID)){ // If in Asset EditMode and no
Workflow ID presents in the OEVF launching URL
                           … // create and add default asset markup to the return vector
                        }
           }

           if (!DMSUtil.isNullOrBlank(wID)){
                File workflowMarkups [] = …; // Retrieve workflow markups
                if(workflowMarkups != null){ // Workflow markups exist
                        … // Add the workflow markups to the return vector
                }else{  // No workflow markup related to the Workflow ID exist yet
                        … // create and add default workflow markup to the return vector
                }
           }
           return vector;
   } …
}
```

7.20.4.3 ActionSave

If your ISDK implementation has special naming convention for automatically generated OEVF default asset markups or default workflow markups during the `ActionOpen` process, then `ActionSave` implementation should prevent new saving markups from overwriting these default OEVF markups. An alert should notify users to use an alternative name. For example, in the Sample Integration of Filesys, it uses the name of the Asset ID or Workflow ID as the default asset or workflow markup name.

If your ISDK wants to save OEVF markups to a special location or do any extra work, they all should be added to your implementation of `ActionSave`. For example, the Sample Integration of Filesys saves asset markups to assets folder and saves workflow markups to workflows folder inside the related document's markups folder. This is done by `saveInstance()` method of `Markup` class.

7.20.4.4 ActionDelete

`ActionDelete` implementation should prevent users from deleting default asset markup and default workflow markup. An alert should notify users when they try to do so.

7.20.5 DOCID

To support OEVF, the `DocID` in your ISDK will be replaced with a new structure that includes Asset ID and Workflow ID in addition to your existing DocID structure.

7.21 Implementing UI Customization

When designing DMS Extension to launch the AutoVue applet, the following functionalities can be supported by using JavaScript code at UI level for your integration:

- Embedded vs. Pop-up Window for displaying AutoVue applet
- Pop-up Blocker detecting
- Prompt to save markups when exiting AutoVue by closing Web browser window

7.21.1 Embedded vs. Pop-up Window

This controls the window used for hosting the AutoVue applet. It focuses on two options:

- Displaying AutoVue applet in a pop-up window which could then be re-used for subsequent file view.
- Displaying AutoVue applet embedded inside the caller's browser window (could be a specific size / frame, and so on).

AutoVue applet can be created in a JSP or a HTML file using JavaScript code. In the Sample Integration for Filesys, it is created by frmApplet.jsp. The following `jvapp` string in frmApplet.jsp contains code that can be used to create the AutoVue applet in either of the above two options.

```
var jvapp = '<HTML><HEAD><TITLE>Powered by AutoVue</TITLE>' +
      '<META HTTP-EQUIV="content-type" CONTENT="text/html;charset=UTF-8">' +
      '\n<Script' +' language="JavaScript">' +
       …
      '\n</Script' + '>\n</HEAD>'+
      '<BODY …>\n' +
      '\n<APPLET NAME="JVue" CODE="com.cimmetry.jvue.JVue.class"' +
      ' ARCHIVE="jvue.jar,jogl.jar,gluegen-rt.jar"' +
      ' CODEBASE="' + CODEBASE + '"' +
      ' HSPACE="0" VSPACE="0" WIDTH=100% HEIGHT=100% MAYSCRIPT>' +
      '\n<PARAM NAME="JVUESERVER" VALUE="'+ JVUESERVER + '">' +
      '\n<PARAM NAME="DMS" VALUE="' + DMS + '">' +
       …
      '\n</APPLET></BODY></HTML>';
```

The function `CreateReusableApplet()` creates the AutoVue applet in a pop-up window which can be re-used while `CreateApplet()` creates the AutoVue applet in embedded mode. Inside `CreateReusableApplet()`, if you want the same user to reuse the same pop-up window, you can name the applet window in a way so that it is specific for one user. When reusing an existing AutoVue applet, you will need to use the public API of the AutoVue applet to set the current file in order to view it.

```
// Create reusable AutoVue applet in a pop-up window
function CreateReusableApplet()
{
   var appletWnd = self;
   …
   appletWnd = window.open("",NAME_OF_THE_POPUP_WINDOW,
'resizable=1,width=770,height=630,location=0,toolbar=0,menubar=0,status=0,left=400,top=
150');

   if (appletWnd != null) {
        appletWnd.focus();
        var doc = appletWnd.document;

        var japplet = doc.applets["JVue"];
        if (japplet != null) {  // AutoVue Applet exists already, reuse it
             …
             japplet.setFileThreaded(FILE_TO_VIEW); // set the file to view
        } else {
             // Fix for Java Plugin on IE only
             if (doc.readyState != null) {
                var i = 0;
                while ( i < 100 && doc.readyState != "complete" ) {
                    appletWnd.setTimeout('dummy()', 1000);
                    i++;
                }
             }

              if(!appletWnd.closed) {
                 doc.open();
                 doc.writeln(jvapp);  // write to create an AutoVue applet
                 doc.close();
              }
        }
         appletWnd.focus();
    }
    …
}
```

```
// Create AutoVue applet embedded in the caller's browser window
function CreateApplet()
{
    var appletWnd = self;
    var doc = appletWnd.document;

    doc.writeln(jvapp);
    doc.close();
}
```

Here is the code to set file in the AutoVue applet.

```
/*
** Sets a file (URL) in the Applet
*/
function setFile(fileURL)
{
    var appletWnd = self;   // Use the same window if embedded
    if (EMBEDDED != 'true') {  // For pop-up window
        appletWnd = window.open("",NAME_OF_THE_POPUP_WINDOW,"");
    }
    if (appletWnd.jVueLoaded) {
        // Load file on a separate thread.
        appletWnd.document.applets["JVue"].setFileThreaded(fileURL);
        appletWnd.focus();
    } else {
        …;
    }
}
```

By default, the Sample Integration of Filesys embeds the AutoVue applet in the caller's browser window unless an `embedded` request parameter is passed in to the frmApplet.jsp. If your URL contains `frmApplet.jsp?embedded=0`, then Filesys creates the AutoVue applet in a separate pop-up window. In your ISDK implementation, you can set your default option and choose the way to detect another option.

```
String sEmbedded = request.getParameter("embedded");
boolean embedded = true;
if (sEmbedded != null && sEmbedded.length() > 0 && sEmbedded.equalsIgnoreCase("0")) {
        embedded = false;
}
```

7.21.2 Pop-up Blocker

The implementation of the following Javascript code can determine whether a Web browser has a pop-up blocker enabled.

In frmApplet.html of the Sample Integration of Filesys, the function `validatePopups()` detects whether a pop-up window can be created.

```
…
<html>
<head>
  <title>AutoVue Web Edition</title>
<script>
…
<!-- //Hide script from old browsers
function validatePopups()
{
        var tinyWindow = null;
        try {
                tinyWindow = window.open("popup.html", "PopupTest", "width=10,
height=10, left=2000, top=2000 ");
        }
        catch (e) {
                return false;
        }

        window.focus();
        if ( tinyWindow ) {
                try {
                        tinyWindow.close();
                }
                catch (e){;}
                return true;
        }
        return false;
} // end validatePopups()

// -->
</script>
</head>
…
```

Before launching AutoVue in a pop-up window, `validatePopups()` is called:

```
if ( validatePopups() == true) {
      CreateReusableApplet();
}
else {
      alert("Please set your pop-up blocker to allow launching AutoVue.");
}
```

7.21.3 Prompt to Save

In order to prompt for saving changes made to markups when exiting AutoVue by closing a Web browser window, AutoVue applet's `saveModifiedMarkups()` and `waitForLastMethod()` methods should be called. For example, when `onBeforeUnLoad` event is fired before the hosting windows closes.

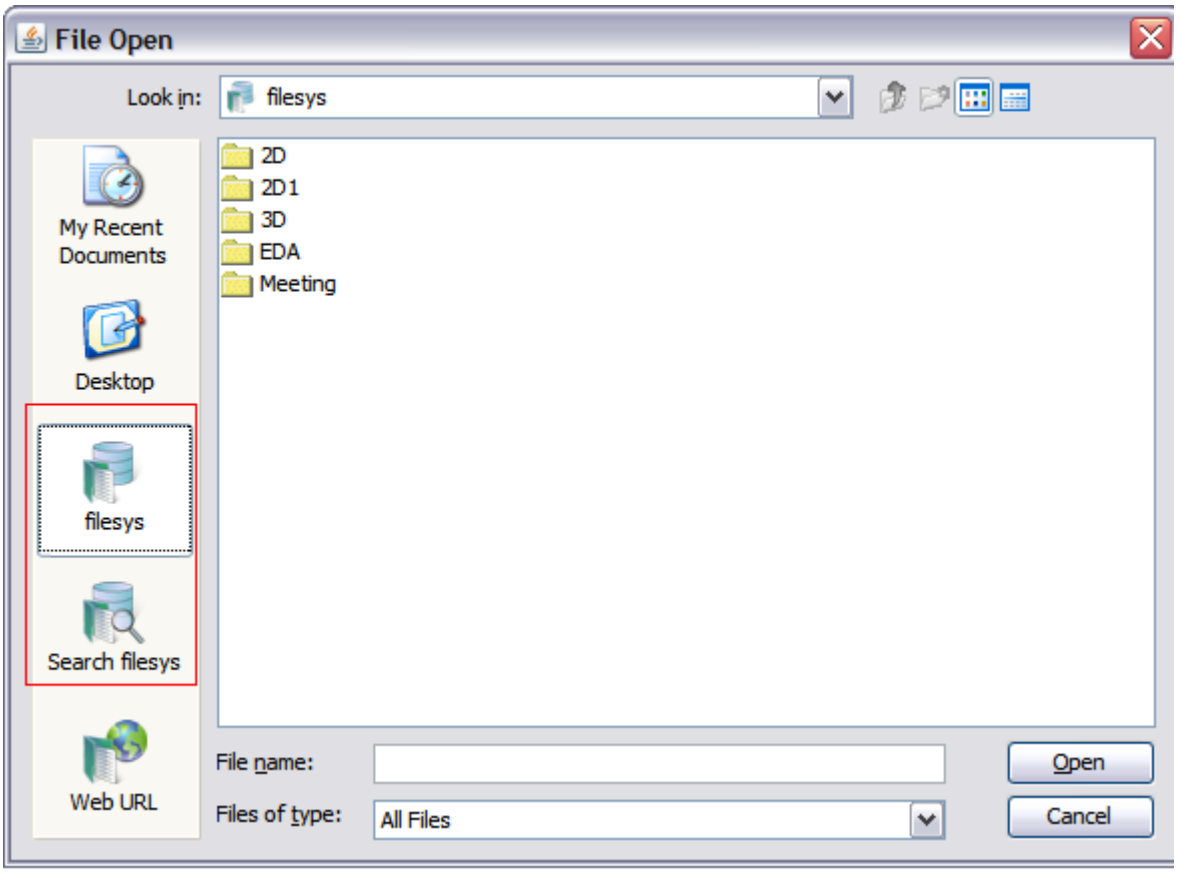
Here is code excerpt from frmApplet.html of the Sample Integration of Filesys.

```
<script language="JavaScript">
<!-- //hide script from old browsers
…
var jvapp = '<HTML><HEAD><TITLE>Powered by AutoVue</TITLE>' +
        …
       '\n<Script' +' language="JavaScript">' +
       '\n <!-' + '- hide script from old browsers' +
       '\n    function SaveMarkups() { ' +
       '\n      window.document.applets["JVue"].saveModifiedMarkups(); ' +
       '\n      window.document.applets["JVue"].waitForLastMethod(); ' +
       '\n    }' +
       '\n //-' + '-> ' +
       …
       '<BODY marginheight="3" marginwidth="3" leftmargin="0" topmargin="0" scroll="no"
onBeforeUnLoad="SaveMarkups();">\n' +
       …
// end script hiding from old browsers -->
</script>
```

7.22 Returning DMS Name

The latest AutoVue server allows browsing and searching multiple DMS backend system through multiple Integration SDKs. Your Integration SDK should handle `GetProperties` request for property GUI with value DMS to return the right name for the DMS backend system.

Here is a sample of the AutoVue server's File Open dialog that displays the DMS name *filesys* for the Sample Integration of Filesys.

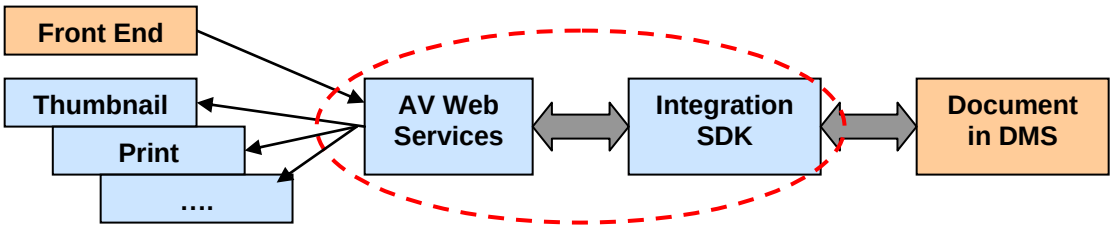


The Integration SDK handles the request in `GetPropGUI` class. Here is the implementation provided in the Sample Integration of FileSys. You need to replace *filesys* with your own DMS name.

```
package com.cimmetry.vuelink.filesys.propactions;
…
public class GetPropGUI implements … {
  …
  public DMSProperty execute(…) throws VuelinkException {
        DMSProperty retProp = null;
        if (property.getValue().equalsIgnoreCase("DMS")) {
           retProp = new DMSProperty(DMSProperty.PROP_GUI, "filesys");
        } else if …
  }
  …
}
```

7.23 Leveraging AutoVue Web Services

The AutoVue Web Services package provides a standard interface for developers to take advantage of AutoVue functionalities such as thumbnail generation, streaming file generation, print, convert, text extraction, and so on. Refer to AutoVue Web Services documentation for detailed description.



7.23.1 Configuring AutoVue Web Services to Communicate with Integration SDK

In order to enable the AutoVue Web Services to communicate with the Integration SDK, the following configurations need to be done on AutoVue Web Services side.

- Updating `vuelinkProtocol` in the web.xml file.

  Suppose that `vuelinkISDK` is the `vuelinkProtocol` for your Integration SDK then it needs to be added to `<AutoVue_Web_Services_Install_Dir>\autovue_webservices\AutoVueWS\WEB-INF\web.xml`.

```
<env-entry>
  <env-entry-name>vuelinkProtocol</env-entry-name>
  …
  <env-entry-value>vuelinkISDK</env-entry-value>
  <injection-target>
  …
  <injection-target-name>vuelinkProtocol</injection-target-name>
  </injection-target>
</env-entry>
```

You may need to update `destinationDIR`, `initialJVueServer`, `vuelinkPropsDir`, and so on. Refer to the AutoVue Web Services Developer Guide for detailed description.

- Creating a properties file naming with the `vuelinkProtocol` defined.

  If `vuelinkISDK` is the `vuelinkProtocol` for your Integration SDK, then a `vuelinkISDK.properties` file should be created and put in the `vuelinkPropsDir` folder defined the web.xml (for example, `%AutoVue_Web_Services_Install_Dir%\autovue_webservices\sample_config` folder). The following is a sample configuration file for the Sample Integration for Filesys.

```
#Integration SDK connection info

DMS=http://FilesysHost:7001/ISDK/servlet/FilesysVuelink
#example: DMS=http://localhost:8080/webtop/com.cimmetry.vuelink.documentum.DMS

#if any DMSArgs is needed add like this
#DMSArgs=someArg1;someArg2
#someArg1=some value
#someArg2=some other value
```

  The DMS value, `http://FilesysHost:7001/ISDK/servlet/FilesysVuelink` in the above sample, should be accessible in the Web browser. It refers to your Integration SDK's main DMS Servlet that extends the `VueLink` class and is defined in the web.xml file of your Integration SDK.

7.23.2 Utilizing AutoVue Web Services at Front End

Your front end can consume AutoVue Web Services using Java Client and .Net Client. You can refer to the "How to use AutoVue Web Services" section in the *AutoVue Web Services Developer Guide* for information on how to generate AutoVue Web Services client.

The following describes how AutoVue Web Services APIs should be used in order to generate thumbnails and streaming files, as well as how to convert documents to TIFF, BMP and PDF format. For more samples on how to use AutoVue Web Services API to retrieve printer information, print document, retrieve document properties, text and external reference information, refer to the "Appendix A – Sample Client Code in Java" section in the *AutoVue Web Services Developer Guide*.

7.23.2.1 Thumbnail Generation

- Provide authorization information using `com.oracle.autovue.services.AuthorizationProxy` class

  The following is used by the Sample Integration for Filesys.

```
AuthorizationProxy authorizationProxy;

authorizationProxy = new AuthorizationProxy();
authorizationProxy.setUsername("demo");
authorizationProxy.setPassword("demo");
```

- Define URI

  The value for URI starts with the vuelinkProtocol for your Integration SDK, followed by `://` and by the original URL used to address document of your Integration SDK. That is: `vuelinkProtocol://OriginalURLForYourISDK`. This original URL is the same as what is being used to set FILENAME when creating AutoVue applet to view a file, for example, inside frmApplet.jsp file of the Sample Integration for Filesys.

  The following is a sample URI for the Sample Integration for Filesys.

```
String URI = "vuelinkISDK:///2D/AutoCAD.dwg/ AutoCAD.dwg(1)/ AutoCAD.dwg";
```

- Set Convert Options

  Set convert options using `com.oracle.autovue.services.options.ConvertOption` class. For example, for thumbnail generation, you can set conversion format to be `Format.JPG` or `Format.PNG`. For more options that can be set, refer to the ConvertOptions class API in the JavaDocs.

```
ConvertOption convertOption = new ConvertOption();
convertOption.setFormat(Format.JPG);

convertOption.setScaleFactor(100);
convertOption.setHeight(640);
convertOption.setWidth(480);
convertOption.setPage(1);
```

- Do Conversion

  Call VueBeanWS's `convert()` method to do conversion. The converted thumbnail file content is returned and can be written to a file. For this to work, your ISDK should have fulfilled the tasks described in Handling Renditions section.

```
VueBeanWS_Service service = new VueBeanWS_Service();
VueBeanWS proxy = service.getVueBeanWSPort();

boolean openAllMarkups = false;

byte[] file = proxy.convert(URI, convertOption, authorizationProxy, openAllMarkups);

FileOutputStream fos = new FileOutputStream("C:/temp/AutoCAD.jpg");
fos.write(file);
fos.close();
```

Since the AutoVue server does not support JPG or PNG conversion, when the conversion format is set to `Format.JPG` or `Format.PNG`, the `VueBeanWS` class of AutoVue Web Services internally passes `Format.BMP` to the AutoVue server and then converts the returned BMP file to JPG or PNG format.

If you want the JPG or PNG file to be checked into DMS automatically when AutoVue Web Service `convert()` method is called, you need to enhance your Integration SDK's rendition handing. If BMP rendition type is detected when handling rendition, you can add extra code to convert the BMP rendition to a JPG or PNG format with desired thumbnail size like AutoVue Web Services does and checked it into DMS so that your application can display a thumbnail for your document.

7.23.2.2 Streaming File Generation

Whenever a VueBeanWS method that has a URI parameter is called, the streaming file for the document in DMS addressed by this URI is generated and checked into DMS automatically.

7.23.2.3 Converting Document to Other Formats

Using AutoVue Web Services, a document can also be converted to TIFF, BMP and PDF format. These renditions are checked into DMS automatically if your Integration SDK implements rendition handling.

The conversion steps are almost the same as steps for Thumbnail Generation, except that you set the `conversionOption` format to `Format.BMP`, `Format.TIF` or `Format.PDF`.

8. APPENDIX A – INTEGRATION SDK SKELETON

The Integration SDK skeleton code acts as a guideline to facilitate custom integration of SDK. It contains all necessary features for an integration (the integration developer must perform the TODO tasks inside the skeleton code. As a sample implementation, the Integration SDK Web Services Client is implemented based on the Integration SDK skeleton code.

8.1 Integration SDK Skeleton Packages

The following packages and classes are included in the ISDK Skeleton:

- VueLink package
  - DMS.java
- actions folder
  - ActionDelete.java
  - ActionDownload.java
  - ActionOpen.java
  - ActionSave.java
  - ActionSetProperties.java
- propactions package
  - GetPropCSI_ClbSessionID.java
  - GetPropCSI_Collaboration.java
  - GetPropCSI_DocDateLastModified.java
  - GetPropCSI_DocName.java
  - GetPropCSI_DocSize.java
  - GetPropCSI_IntelliStamp.java
  - GetPropCSI_IsMultiContent.java
  - GetPropCSI_ListAllProperties.java
  - GetPropCSI_ListItems.java
  - GetPropCSI_Markups.java
  - GetPropCSI_Renditions.java
  - GetPropCSI_Search.java
  - GetPropCSI_UserName.java
  - GetPropCSI_Versions.java
  - GetPropCSI_XREFS.java
  - GetPropDefault.java
  - GetPropGUI.java
- backend package
  - DMSBackendImp.java
- context package
  - DMSContextImp.java
- defs package
  - ISDKDocID

- session package
    - DMSBackendSessionImp

8.2 Integration Steps for Implementing File View Functionality

The first stage of integration is to implement basic view functionality stated in Chapter 6. It includes the following:

- Fulfill TODO list in `DMS` class – the Main DMS Servlet class

.

- Fulfill TODO list in `ISDKDocID` class to defining a unique document identifier
- Fulfill TODO list in `GetPropCSI_UserName` and in related `getProperty()` method of `DMSBackendImp` class to return user name
- Fulfill TODO list in `ActionOpen` class to return the DocID
- Go through `GetPropCSI_IsMultiContent` method and fulfill TODO list in related `getProperty()` method of `DMSBackendImp` class to return multi-content value
- Fulfill TODO list in `GetPropCSI_DocName` and in related `getProperty()` method of `DMSBackendImp` class to return document name
- Go through `GetPropCSI_DocDateLastModified` and fulfill TODO list in related `getProperty()` method of `DMSBackendImp` class to return document date last modified
- Go through `GetPropCSI_DocSize` and fulfill TODO list in related `getProperty()` method of `DMSBackendImp` class to return document size
- Fulfill TODO list in `ActionDownload` class to return document content
- Go through `DMSContextImp` class and fulfill TODO list in `connect()` method of `DMSBackendImp` class to connect to backend DMS. Connection info such as username and password can be hard-coded at this stage in order to connect to DMS.

Refer to Chapter 5 and 6 to assist your implementation of the above classes.
Integration SDK includes a sample csiApplet.jsp in the applet folder for launching AutoVue.
To test file viewing after implementing the classes, do the following:

- Provide the FILENAME variable with your unique document identifier
- Open a browser with URL *http://host:port/IntegrationSDKSkeleton_context/* to launch AutoVue and view file

8.3 Integration Steps for Implementing Advanced Functionality

The next stage of integration is to implement more advanced functionality such as XRefs, markups, compare, renditions, DMS Search & browse,and so on stated in Chapter 7. It includes the following:

- Go though `GetPropCSI_GetPropDefault` class and fulfill TODO list in related `listAllProperties()` method of `DMSBackendImp` class to handle document attributes.

- Go though `GetProCSI_XREFS` class and fulfill TODO list in related `listXRefs()` method of `DMSBackendImp` class to return external references (XREFS).

- Fulfill TODO list in `GetPropCSI_Markups` class and in related methods, for example, `listMarkups()` method of `DMSBackendImp` class to handle Markups.

- Fulfill TODO list in `GetPropCSI_Rendition` class and in related `listRenditions()` method of `DMSBackendImp` class to handle renditions.

- Go though `GetPropCSI_ListAllProperties` class to return the list of all properties of the DMS document.

- Go though `GetPropCSI_ListItem`*s* class and fulfill TODO list in related `listItems()` method of `DMSBackendImp` class and in `buildBrowseGUIProperty()` method in `GetPropGUI` class to implement DMS Browse.

- Fulfill TODO list in `GetPropCSI_Search` class, related `listSearchResults()` method of `DMSBackendImp` class and `buildSearchGUIProperty()` method in `GetPropGUI` class to implement DMS Search.

- Go though `GetPropCSI_Versions` class and fulfill TODO list in related `listVersions()` method of `DMSBackendImp` class to handle document versions.

- Fulfill TODO list in `ActionSave` class and in related `saveChat()`, `saveMarkup()` and `saveRendition()` methods of `DMSBackendImp` class to implement file save action

- Go though `ActionDelete` class and fulfill TODO list in related `deleteMarkup()` method of `DMSBackendImp` class to implement file delete action

- Go though `GetPropCSI_Intellistamp` class and fulfill TODO list in related `getIntelliStamp()` method of `DMSBackendImp` class to support Stamp in

Markup. `GetPropDefault` class needs to be enhanced to handle a pick list for Stamp.

- Go though `GetPropCSI_MarkupPolicy` class and fulfill TODO list in related `getIntelliStamp()` method of `DMSBackendImp` class to support Stamp in Markup.

- Refer to the "Implementing Security and Authentication" section in Chapter 7 to implement security and authentication. Basically, the jsp to launch AutoVue applet, `getDMSContextImp` class and `DMSBackendImp` class are involved.

- Refer to the "Implementing RTC and Meeting Management" section in Chapter 7 to integrate with backend meeting management system.

Refer to Chapter 7 to assist your implementation of the above advanced functionally.

9. APPENDIX B – SAMPLE INTEGRATION FOR FILESYS

The Sample Integration for Filesys DMS included in the AutoVue Integration SDK acts as an example and for getting familiar with the integration framework. The following figure shows the use case diagrams of possible actions available from within the AutoVue interface. A user logs into FilesysDMS through a Web browser and selects a file to view in AutoVue. Once the file is loaded in AutoVue, the user can perform other actions such as markup, conversion, compare, search, browse, and so on.

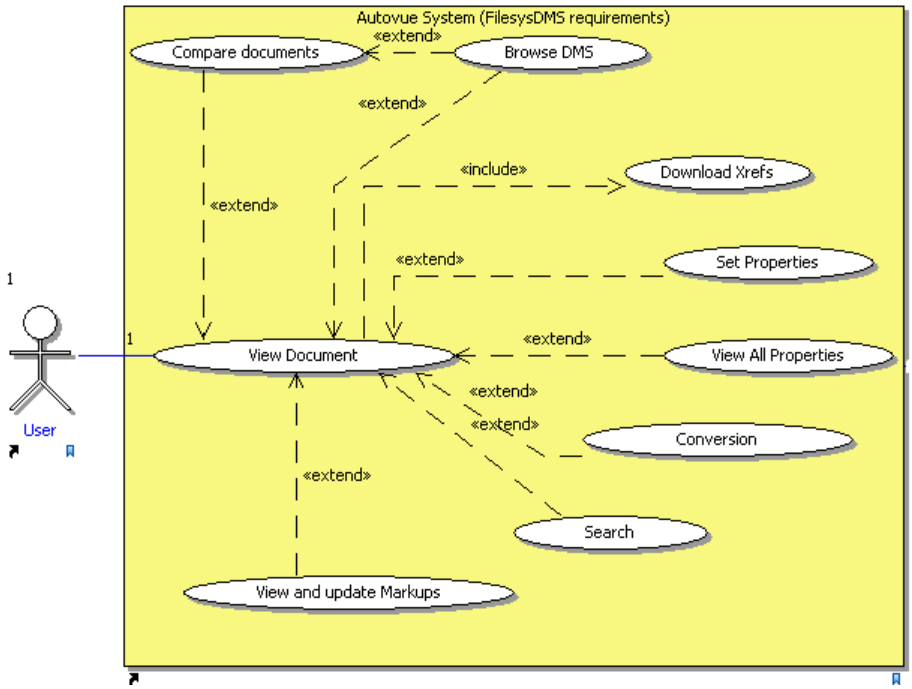


Figure 9-1: Use cases diagram for the FilesysDMS sample

As illustrated in the following figure, we have designed the Vuelink servlet class and three packages for the FilesysDMS integration, as follows:

1. The first package is called `com.cimmetry.vuelink.filesys.actions` and contains all action classes. The common characteristic of these classes is that they all implement the DMSAction<AnyContext extends DMSContext> interface.
2. The second package is called `propactions` and contains a set of classes that all implement the DMSGetPropAction interface.
3. The third package is called *backend* and has three classes: the FilesysDMSBackend class that implements the DMSBackend interface, the FilesysDMS class which is the backend API that talks to FilesysDMS backend system, and the FilesysDocID class which implements the DocID interface and defines the document ID.

Refer to chapters 5 and 6 for more information on the design of the Sample Integration for Filesys DMS.

- actions
  - ActionDelete.java
  - ActionDownload
  - ActionGetProperties
  - ActionOpen
  - ActionSave
  - ActionSetProperties
- backend
  - FilesysDMSBackend
  - FilesysDMSBackendImp
  - FilesysDMSDocID
- propactions
  - GetFilesysProperty
  - GetPropCSI_ClbSesssionID
  - GetPropCSI_Collaboration
  - GetPropCSI_DocDateLastModified
  - GetPropCSI_DocName
  - GetPropCSI_DocSize
  - GetPropCSI_IntelliStamp
  - GetPropCSI_IsMultiContent
  - GetPropCSI_ListAllProperties
  - GetPropCSI_ListItems
  - GetPropCSI_MarkupPolicy
  - GetPropCSI_Markups
  - GetPropCSI_Renditions
  - GetPropCSI_Search
  - GetPropCSI_UserName
  - GetPropCSI_Versions
  - GetPropCSI_XREFS
  - GetPropDefault
  - GetPropGUI
- session
  - FilesysBackendSession
- util
  - Credentials
  - ListParser
  - OevfParser
- vuelink
  - FilesysVuelink
  - FilesysContext

Note: The propactions package does not list all the classes in the package.

The data used by the sample integration is based on a simple file system that has a simple data structure to store and retrieve files (the data structure is described in the next section of this Appendix). The file system includes three packages:

- domain
  - Version
  - Browse
  - Markup
  - XRef
  - DocInfo
  - FolderObj
  - Search
  - IFilesysDMSInfo
  - DocumentObj
  - DocInfoImpl
  - Rendition
  - FilesysDMSFacade
- Util
  - FilesysDataStructureCreator
  - FilesysDataStructureDefs
  - FilesysDataStructureInfos
- Gui
  - ListDirServlet

1. The first package is called `domain` and contains all the classes dedicated to managing the data backend system. When we implemented our actions to retrieve and store files in the backend system, we did it through the `com.cimmetry.vuelink.filesys.dms.domain.`*`IFilesysDMSInfo`* interface. This interface is our plug-in point to the FilesysDMS backend system manager.
2. The second package is called `util` and allows us to add new data to the backend system. The instructions on how to add new data are described in the **User Guide**.
3. The last package is called `gui` and it contains a servlet which allows us to navigate the sample files through a dynamic HTML page and a servlet to manage user login.

9.1 DMSActions

A DMSAction has only one method to implement: `execute()`. It takes four parameters:

- **AnyContext that implements DMSContext:** Represents the context of execution of a `DMSAction` and holds various environment settings.
- **DMSSession:** Represents the session of execution of a `DMSAction` for an arbitrary set of DMS queries.
- **DMSQuery:** Represents a query that a `DMSAction` must handle and holds parameters such as the original document URL (FILENAME param passed in the

AutoVue applet page), the document ID, the collaboration session ID, the collaboration session data, the Authorization and a set of Properties.

- **DMSArgument:** Represents list of objects used to hold special arguments specific to a given DMS action type.

The `execute()` method returns an object instance (the type of the instance depends on the DMS action but it is generally either null, a `DocID`, a File or a Property list). To report failures, execute can throw a `VuelinkException` containing the error code and error message (defined in the `DMSDefs` public interface) that the VueLink servlet uses to build the `<ERROR>` HTTP response.

One important goal of the AutoVue Integration SDK is to make the integration open to extensions and modifications. We achieved that by registering the action classes in the `web.xml` file in init-parameters. The VueLink servlet checks the init-parameters and registers the actions. Each action parameter name has the prefix `dms.action` followed by the name of the action as `dms.action.open` (for example, for Open Action). The value parameter specifies the action name and its location (for example, `com.cimmetry.vuelink.filesys.actions.ActionOpen`). This mechanism allows us to drop any obsolete class and replace it by a new one simply by updating the init-parameter.

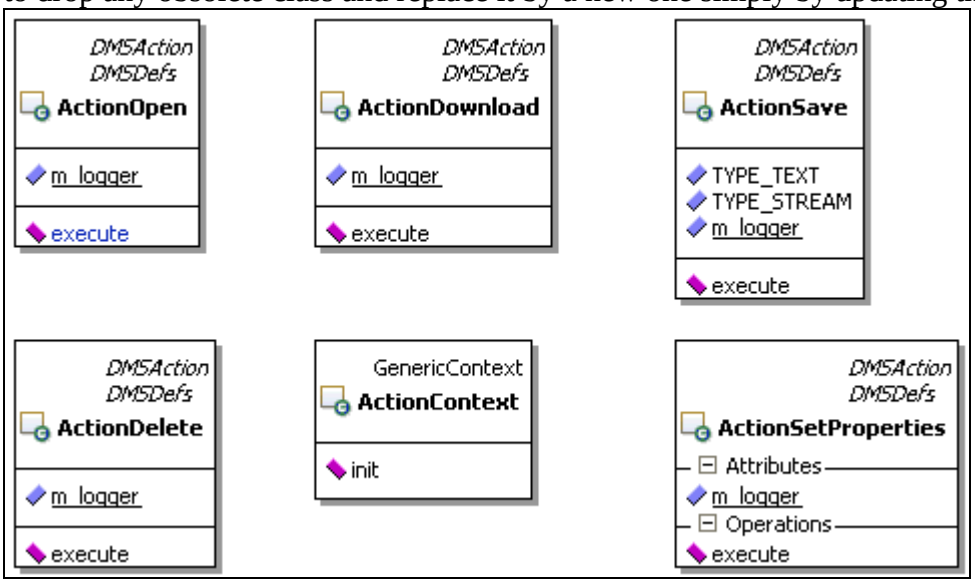


Figure 9-4: Action classes

In the Filesys DMS, we designed the `com.cimmetry.Vuelink.filesys.actions` package which implements all the needed actions. In this section we discuss the `Open`, `Download`, `Save`, and `Delete` actions. The `SetProperties` and `GetProperties` are discussed in the following sections.

Each individual class must be registered in the `web.xml` (web descriptor for your J2EE web application) file init parameters. The name of the parameter has the format dms.getprops.<property name> (for example, dms.getprops.CSI_Markups). The value of the parameter contains the full qualification of the class and has the format "com.<yourCompany>.<package>.<class name>". You can choose the class name you want.

Also, if you prefer, you can choose the default name proposed by framework "GetProp<prop name>" (for example, `GetPropCSI_Markups`).

This makes the code easier to maintain and, more importantly, makes customization a lot easier. If changes to markup handling are required, the `GetPropCSI_Markups` class can be re-implemented without affecting the handling of any of the other properties. This will make the customization easier in the first place, and the customizations will be easier to update when the framework is updated. This will also allow the easy mix-and-match of functionality. For example, if a customized markup handler is done for Customer A, and later Customer B needs similar functionality, the class written for A can be dropped into B's install without impacting any other customizations done for B.

For the Filesys DMS, we designed the `com.cimmetry.vuelink.filesys.propactions` which contains the following property action classes:

- **`GetFilesysProperty`**: Returns all document attributes and saves to reuse. It serves as support to some of the following classes.
- **`GetPropCSI_ClbSessionID`**: Handles the CSI_ClbSessionID property and returns the session ID for a AutoVue Real-Time Collaboration session.
- **`GetPropCSI_Collaboration`**: Handles the CSI_Collaboration property and returns the GUI for a AutoVue Real-Time Collaboration session.
- **`GetPropCSI_DocDateLastModified`**: Handles the `CSI_DocDateLastModified` property and returns the date of the last modification of a document.
- **`GetPropCSI_DocName`**: Handles the `CSI_DocName` property and returns the name of a document.
- **`GetPropCSI_DocSize`**: Handles the `CSI_DocSize` property and returns the size of a document.
- **`GetPropCSI_IntelliStamp`**: Handles the `CSI_Intellistamp` property and returns the Stamp definition file and underlying images if available.
- **`GetPropCSI_IsMultiContent`**: Handles the `CSI_IsMultiContent` property.
- **`GetPropCSI_ListAllProperties`**: Handles the `CSI_ListAllProperties` property and returns an array of DMS properties.
- **`GetPropCSI_ListItems`**: Handles the `CSI_ListItems` and returns an array of items to be displayed in the browse GUI.
- **`GetPropCSI_MarkupPolicy`**: Handles the `CSI_MarkupPolicy` property and returns the content of MarkupPolicy file if available.
- **`GetPropCSI_Markups`**: Handles the `CSI_Markups` property and returns an array of properties concerning markups documents.
- **`GetPropCSI_Renditions`**: Handles the `CSI_Renditions` property and returns an array of properties concerning renditions documents.
- **`GetPropCSI_Search`**: Handles the `CSI_Search` property and returns an array of properties of documents that match the criteria search.
- **`GetPropCSI_UserName`**: Handles the `CSI_UserName` property and returns the username.

- **`GetPropCSI_Versions`**: Handles the `CSI_Versions` property and returns an array of document versions properties.
- **`GetPropCSI_XREFS`**: Handles the `CSI_XREFS` property and return an array of properties concerning the XRefs documents.
- **`GetPropDefault`**: Handles the properties that do not have dedicated individual classes.
- **`GetPropGUI`**: Handles the `GUI` property and returns an array of properties for building the browse GUI or the search GUI or returns a property with proper DMS name.

9.2 Backend API

Note: DMSBackend interface is optional. It is intended as an entry point to your custom code for handling communication with your DMS/PLM system. You can think of the Backend class as a wrapper around your DMS API.

The backend API allows the integration interface to properly talk with the DMS. This API is intended to gather all the custom code for handling communication with the DMS. Our backend class that implemented the `DMSBackend` interface also implemented the `connect` method which allows AutoVue to reuse existing user sessions with the DMS.

The framework locates the object that implements the backend API for an integration inside the `com.cimmetry.vuelink.context.`*`DMSContext`* object. During the initialization of the VueLink servlet, a `DMSContext` object is created which in turn initializes and registers the backend object. This allows you to get a reference to the backend API from a `DMSContext` object. This is always possible since all the `DMSAction` objects and `com.cimmetry.vuelink.propsaction.`*`DMSGetPropAction`* objects hold a `DMSContext` object. If custom registration, saving and loading of the backend API object are needed, you must derive the `GenericContext` class and implement the new overriding methods.

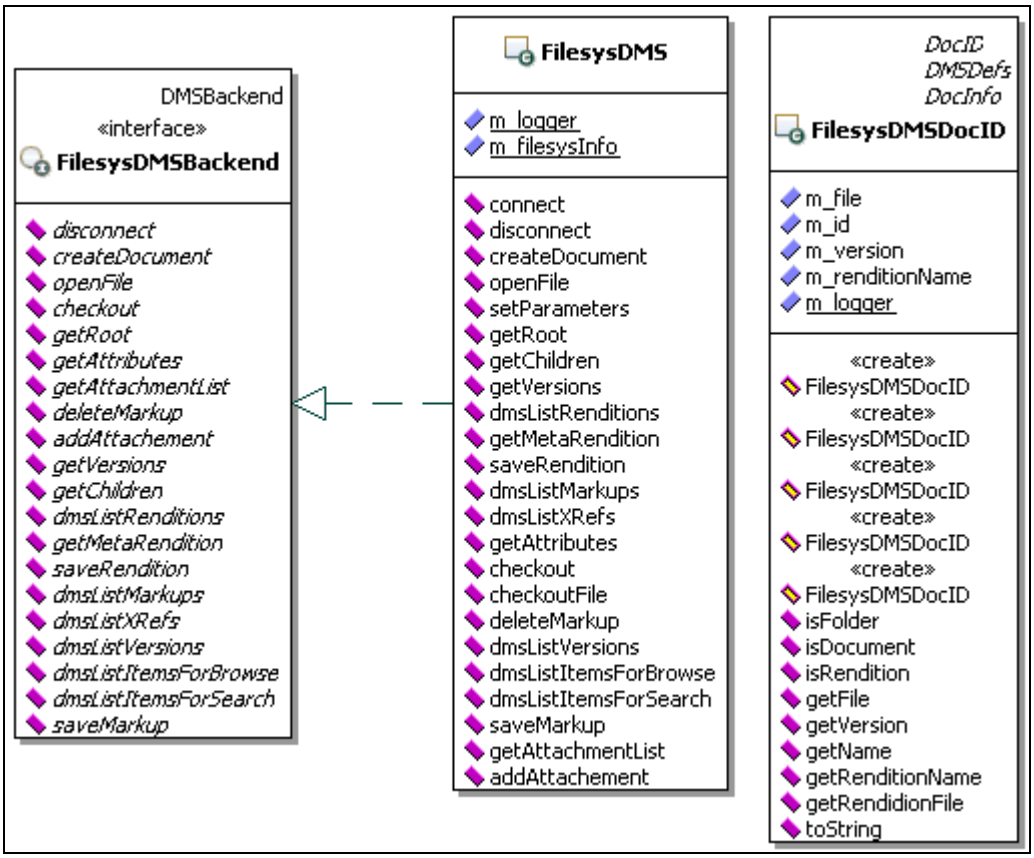


Figure 9-6 Backend classes

First the framework fetches the init parameters for `dms.backend`, the name of the init parameter, and then instantiates the class specified in the value parameter. If it fails, then it looks for the `DMSBackendImp` class as the default name in the current package (That is, in the same location where your DMS servlet is located).

In the Filesys DMS application, the backend API is registered as shown in the following excerpt of code.

```
<init-param>
  <param-name>dms.backend</param-name>
  <param-value>com.cimmetry.vuelink.filesys.backend.FilesysDMS</param-value>
</init-param>
```

The following excerpt of the code shows how to get an instance of the plug-in point to Filesys DMS backend system.

```
/** Instance of FilesysDMS object (singleton) responsible for communicating and providing
Vuelink with the required information */

private static final IFilesysDMSInfo m_filesysInfo = FilesysDMSFacade.getFilesysInstance();
```

9.3 Filesys DMS Backend system Structure

The data used by this sample is based on a file system. This system has a simple structure to store and retrieve files. This structure consists of folders and document objects. Folder objects represent directories and document objects represent files. Folder objects can contain a list of document objects and a list of folder objects (the subfolders).

The access to the root of the FilesysDMS system structure is done through a given specific path. Inside the FilesysDMS structure we categorize the documents to allow flexible and easy document management. Each category is simply represented by an access path. Thus, the FilesysDMS system structure contains all the categories of documents to manage. For example, in Figure 9-7, filesysDatabase is the root directory which contains two documents categories: 2DRepository and 3DRepository.

Inside a category one finds several folders (one folder per document). Each folder has the same name as the base document that it represents, and contains all the versions of this document. Each version is represented by a folder which has the same name as its base document concatenated to the number of the version enclosed between parentheses. For example, in figure 9-7, the category 2DRepository contains three folders which correspond to the base documents *bike.dgn*, *main.dgn* and *myacad12.dgn*. The folder *myacad12.dgn* contains three versions of the base document and *myacad12.dgn(3)* represents its third version.

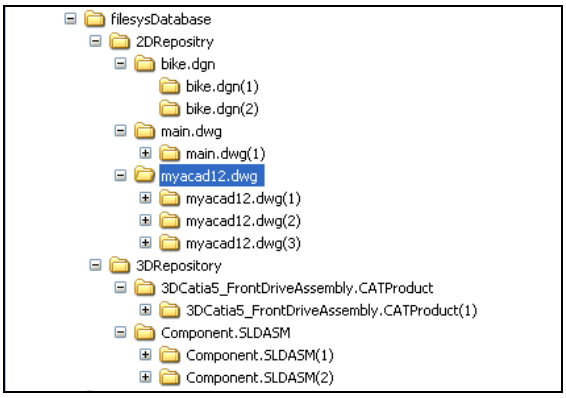


Figure 9-7: FilesysDMS data structure

Each version folder contains all related information (XRefs, markups, renditions, and so on). For example, as illustrated in Figure 9-8, under the folder representing version 2 of the document *myacade12.dgn*, there is the base document and the folders which contain the external references (for example, in the case of a composite document), the markups, and renditions. The **XRefs** folder contains all files which constitute external references. The **markups** folder contains three subfolders which correspond to the different types of markups supported by AutoVue: normal, master and consolidated (see figure 9-9). It might also contain two additional subfolders if OEVF is supported: assets and workflows. Each of the normal, master and consolidated subfolders contain all corresponding markups. Each of the assets and workflows subfolders contain subfolders named by the assetID and workflowID that contain the corresponding asset and workflow markup. Finally, the renditions folder

contains all conversions supported by AutoVue and the streaming files. For example, the tiff in Figure 9-9 folder contains the TIFF rendition. Note that the rendition subfolders have the same names as the rendition types.

Figure 9-84 : A document version structure

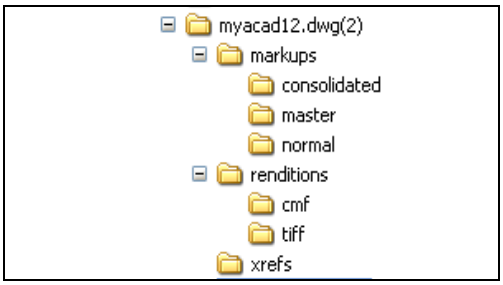


Figure 9-9: Content of version subfolders

This simple structure represents a good starting point when building your own integration based on the integration framework. Managing documents in this structure requires creating folders and copying files.

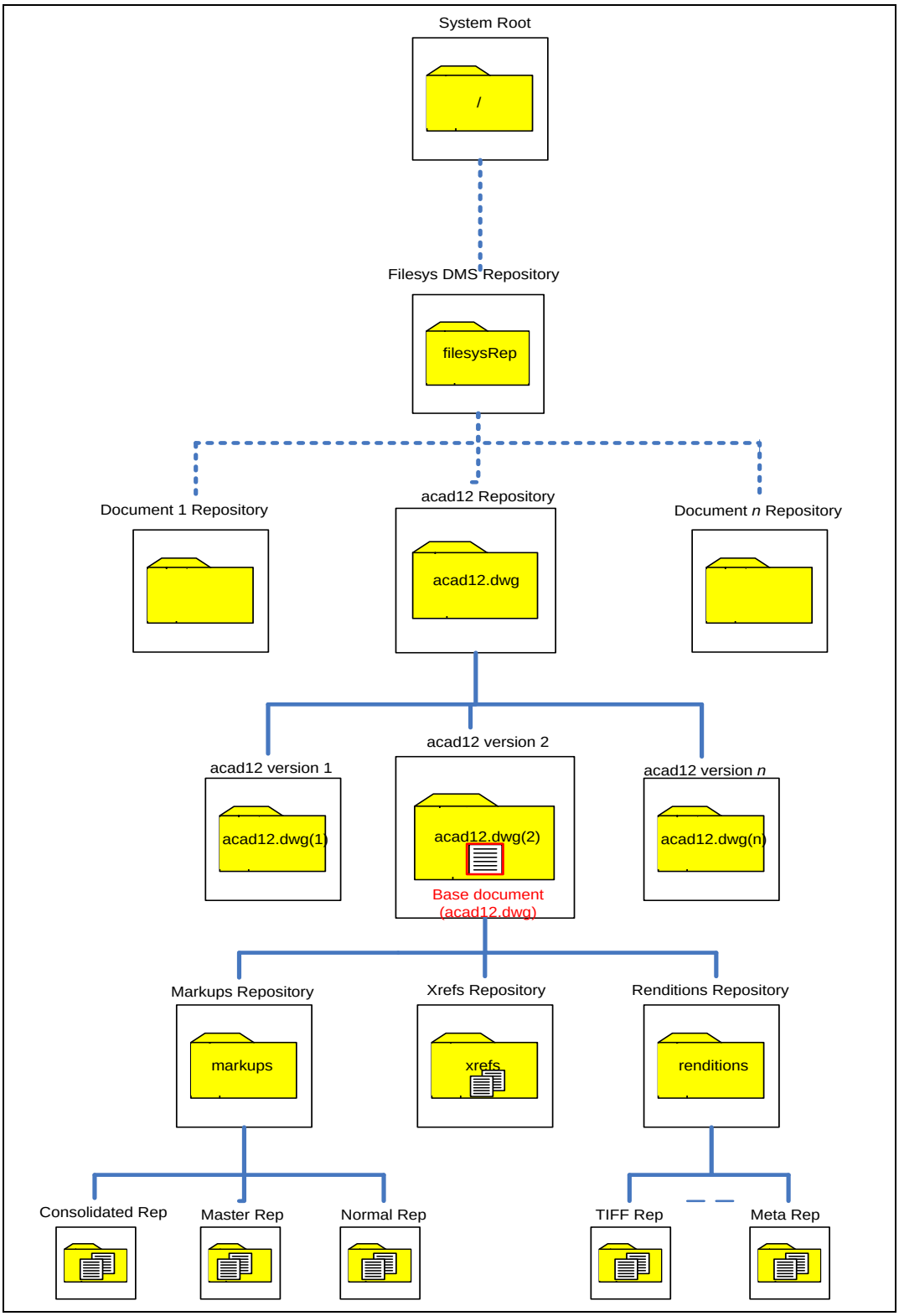


Figure 9-10: Filesys DMS backend system structure

9.4 Sample Integration for Filesys DMS Use Cases

The implementation of the Sample Integration for Filesys DMS involves the implementation of the following functionalities:

- Implementing *DMSAction<AnyContext entends DMSContext>* interface for open/download/save/getproperties, and so on.
- Implementing the backend interface for communicating with the Filesys DMS system.

The requirements for the DMSAction interface are presented in the "Core Use Cases" section and those for the backend interface are presented in the "Backend Use Cases" section.

9.4.1 Core Use Cases

The following six classes implement the DMSAction interface:

- ActionOpen
- ActionDownload
- ActionGetProperties
- ActionSetProperties
- ActionSave
- ActionDelete

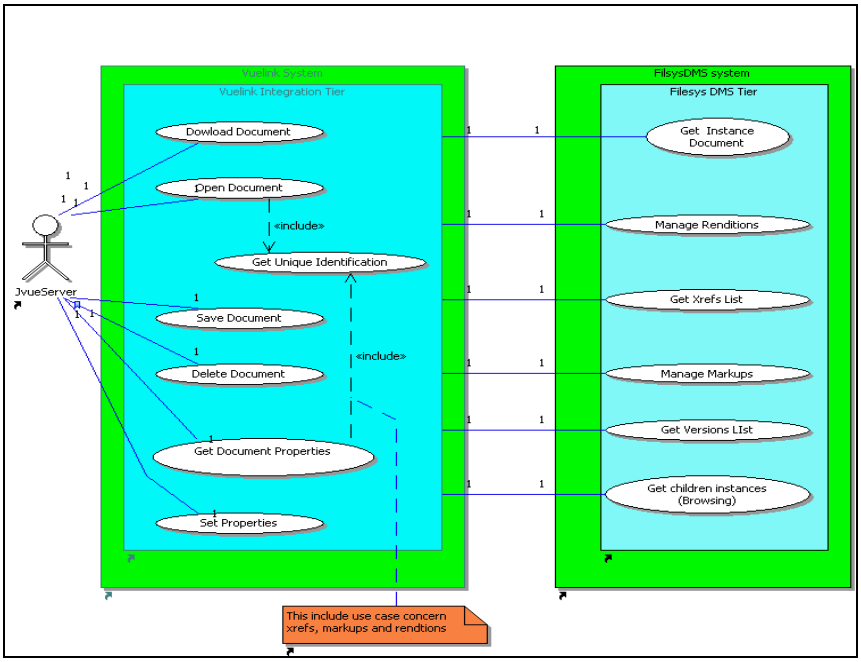


Figure 9-11 DMSAction interface for FilesysDMS and functionalities provided by the Filesys DMS

9.4.1.1 ActionOpen

The exchanged documents between the sample integration and the AutoVue server must have unique identifiers. This is why sample integration must build a unique DocID for each document sent to the AutoVue server.

Use case: Get unique document identification.
Description: The get unique identification use case builds a unique identification for each different document (for example, base document and XRefs documents, and so on) sent to AutoVue server.
Precondition: Sample integration receives an open document request from the AutoVue server.
Deployment constraints: None
Normal flow of events:

1. Sample integration builds a unique identification for each different document returned by *FilesysDMS* and sends it to the AutoVue server.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.1.2 ActionDownload

Sample integration processes the download request when FilesysDMS user wants to view a file from *filesysDMS* backend system.

Use case: Download document
Description: The download document use case communicates to *FilesysDMS* the document to download.
Precondition: *Vuelink* receives a download document request from *AutoVue Server*
Deployment constraints: None
Normal flow of events:

1. *Sample integration* sends a download request to *FilesysDMS* system specified by a unique identifier
2. *Sample integration* returns to *AutoVue Server* the downloaded document

Exception flow of events:

1. *Sample integration* receives the message indicating that the document cannot be downloaded
2. *Sample integration* sends the message to *AutoVue Server*
3. Add the exception to a log

Activity diagram: none
Nonfunctional requirements: None
Open issues: None

9.4.1.3 ActionDelete

Sample integration processes the delete request when *FilesysDMS* user wants to delete markups. The use case below describes this functionality.

Use case: Delete document
Description: The delete document use case communicates to *FilesysDMS* the document to delete.
Precondition: *Sample integration* receives a delete document request from *AutoVue Server*
Deployment constraints: Only markups documents can be deleted.
Normal flow of events:

1. *Sample integration* send a request to *FilesysDMS* system to delete the document specified by a unique identifier

Exception flow of events:

1. *Sample integration* receives a message indicating that the document cannot be deleted
2. *Sample integration* sends the message to *AutoVue Server*
3. Add the exception to a log

Activity diagram: none
Nonfunctional requirements: None
Open issues: None

9.4.1.4 ActionSave

Sample integration processes the save request when *FilesysDMS* user wants to save markups or creates a rendition. When user saves document, *AutoVue Server* sends a request to integration servlet which relays this request to *FilesysDMS* to save the document. The following use case describes this functionality.

Use case: Save document
Description: The save document use case communicates to *FilesysDMS* system the document to save.
Precondition: *Sample integration* receives a save document request from *AutoVue Server*
Deployment constraints: Only markups, renditions (including streaming files) and chat transcript during a Real-Time Collaboration Session can be saved
Normal flow of events:

1. *Sample integration* sends a request to *FilesysDMS* system asking to save a document specified by a unique identifier

Exception flow of events:

1. Sample integration receives a message indicating that the document cannot be saved
2. Add the exception to a log

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.1.5 ActionGetProperties

Sample integration processes the get properties request when *FilesysDMS* user wants to view a file. In this case, the *AutoVue Server* sends several requests to Sample integration asking for information about markups, XRefs, renditions, document properties, and so on. The use case below describes these functionalities.

Use Case: Get properties
Description: The get properties use case takes in charge of multiple requests of *Sample Integration*. The requests concern a set of predefined properties that *Sample integration* must return to *AutoVue Server*. These requests are about XRefs, markups, renditions, GUIs and other information concerning the base document (for example, name, size, and so on.)
Precondition: *AutoVue Server* sends to *Sample integration* request about:

1. Base document properties:
    a. Unique identifier
    b. Last modification date
    c. Size
    d. Name
    e. Author
    f. Type document (for example, folder or file)
    g. Multi content document
2. XRefs properties
    a. Documents unique identifiers of the external references in case of a composite document
3. Markups Properties
    a. Documents unique identifiers and types (normal, master, consolidated) of markups
    b. Markups for all revisions
4. Renditions properties
    a. Unique document identifier when returning a streaming file
    b. A converted document and its type (for example, type rendition)
5. Versions properties
    a. Document Identifier, name, size and version number of the document
6. GetAllProperties property action: a set of properties that characterize a base document (for example, name, size and last modification date, and so on.)
7. GUI properties
    a. The properties that composes the browsing GUI and the search GUI respectively: (1) name, type (folder or file), size and last modification date and (2) document name and extension type.
    b. The property that allow browse functionality

c. The property that allows search functionality
8. Search result: get documents that match the search criteria
9. Browse result: the content of the root backend system folder and the content of the expanded folder until reaching the base document

Deployment constraints: None.
Normal flow of events:
The flow depends on the request of *AutoVue Server*. For each one of the above property request *Sample integration* must provide a response.
Exception flow of events:
a. *Sample integration* is unable to process the request
b. Add the exception to a log
Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.1.6 ActionSetProperties

Sample integration processes the set properties request when *FilesysDMS* user wants to print a file. In this case, *AutoVue Server* sends notification messages when each printed page and when whole document is done printing.

Use Case: Set properties
Description: The set properties use case sends notification messages to *Sample integration*.
Precondition: *AutoVue Server* sends to *Sample integration* notifications about printing.
Deployment constraints: None
Normal flow of events:
Exception flow of events:
Activity diagram: None
Nonfunctional requirements: None
Open issues: CSI_Notifications problem in the JvueServer request is not specified according to the CORE API XML document.

9.4.2 Backend use cases

To provide responses to the AutoVue Server, the integration servlet interacts with FilesysDMS which must provide integration servlet with appropriate information.

9.4.2.1 Get Document Instance

To view a document, *FilesysDMS* must be able to return an instance of the document to *Sample integration*. The use case below describes this functionality.

Use Case: Get document instance
Description: The get document instance use case returns the file instance of a document.
Precondition: *Sample integration* sends to *FilesysDMS* a get document request
Deployment constraints: None
Normal flow of events:
1. *FilesysDMS* finds the document.
2. *FilesysDMS* returns the document to *Sample integration*

Exception flow of events:
1. *FilesysDMS* is unable to find the document
2. Add the exception to a log.

Activity diagram: none
Nonfunctional requirements: None
Open issues: None

9.4.2.2 Manage Renditions

FilesysDMS must be able to manage conversion operations done by the user. It must be able to save a converted documents and streaming files. This functionality is described by the manage renditions use case.

Use case: Manage Renditions
Description: The manage renditions use case manages all operations concerning renditions (for example, (1) save conversions and (2) save and return streaming files).
Precondition: *Sample integration* sends to *FilesysDMS* one of the following renditions requests:
1. Get streaming file instance or
2. Save rendition instance (for example, converted file or streaming file)

Deployment constraints: None
Normal flow of events:
1. *FilesysDMS* finds and returns the streaming file document.

Alternate flow of events:
1. FilesysDMS saves the rendition document (streaming file or converted file).

Exception flow of events:
1. FilesysDMS is unable to find the streaming file document.
2. FilesysDMS is unable to save the rendition document.
3. Add the exceptions to a log.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.2.3 Get XRefs List

In the case of composite document, *FilesysDMS* must provide *Sample integration* with the list of its external references. The use case below describes this functionality.

Use case: Get XRefs list
Description: The get XRefs use case returns a list of external references of a composite document.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for XRefs list documents.
Deployment constraints: None
Normal flow of events:

1. *FilesysDMS* returns the list of XRefs documents.

Exception flow of events:

1. *FilesysDMS* is unable to find the XRefs.
2. Add the exception to a log.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.2.4 Manage Markups

FilesysDMS must be able to provide responses all the requests about markups (for example, return markups list of a document, return markups list of all revisions document, save and delete markups). All these functionalities are described in the following use case.

Use case: Manage markups
Description: The manage markups use case manages all the operations concerning markups.
Precondition: *Sample integration* sends to *FilesysDMS* one of the following requests:

- Get list of markups
- Get list of markups for all revisions
- Save a markup
- Delete a markup

Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns the list of markups.

Alternate flow of events:

1. *FilesysDMS* returns the list of markups for all revisions

Alternate flow of events:

1. *FilesysDMS* saves a markup

Alternate flow of events:

1. *FilesysDMS* deletes a markup

Exception flow of events:

1. *FilesysDMS is unable* to build the list of markups.
2. *FilesysDMS* is unable to build the list of markups of all revisions.
3. *FilesysDMS* is unable to save markup.
4. *FilesysDMS* is unable to delete markup.
5. Add the exceptions to a log.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.2.5 Get Versions List

FilesysDMS must be able to return all the versions of a document when a user needs them to perform a comparison operation. The use case below describes this functionality.

Use case: Get versions list
Description: The get versions list use case returns the list of different versions of a document.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for the list of versions:
Deployment constraints: none
Normal flow of events:

1. *FilesysDMS* returns a list of items representing the different versions of the base document.

Exception flow of events:

1. *FilesysDMS* is unable to return the list of versions.
2. Add the exception to a log.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

9.4.2.6 Get Children Instances

The user must be able to browse the *FilesysDMS* data structure by expanding folders. This is why it must provide *Sample Integration* by the children documents of each expanded folder. The use case Get children instances describes this functionality.

Use case: Get children instances
Description: The get children instances returns a list of items contained in a folder. The user browses the *FilesysDMS* database structure by expanding folders.
Precondition: *Sample integration* sends a request to *FilesysDMS* asking for the list of items contained in the selected folder.
Normal flow of events:

1. Get List of items contained in the specified folder.

Deployment constraints: None
Normal flow of events:

1. *FilesysDMS* returns the list items contained in a specified folder.

Exception flow of events:

1. *FilesysDMS* is unable to return the list of items.
2. Add the exceptions to a log.

Activity diagram: None
Nonfunctional requirements: None
Open issues: None

10. APPENDIX C – ISDK WEB SERVICE CLIENT

10.1 Introduction

This appendix focuses on the Blue Print Web Service Definition Language (WSDL), the Web Service Client package that is built using that WSDL, and the requirements for deploying and connecting the WSDL to your Web Service implementation of Blue Print WSDL file. WS-Security extensions/mechanisms (which are already supported by our client package) and how you can replace them and plug-in other security extensions according to your WS-Security requirement are discussed.

10.2 Architecture

The ISDK Web Service Client is a package built on top of the ISDK Skeleton. It is designed to communicate out of the box with any Web Service (WS) provider that is implementing the BluePrint.wsdl file.

Once the communication between Web Service and the WS Client is established, the rest of the communication (between WS client and the AutoVue server) is already in place.

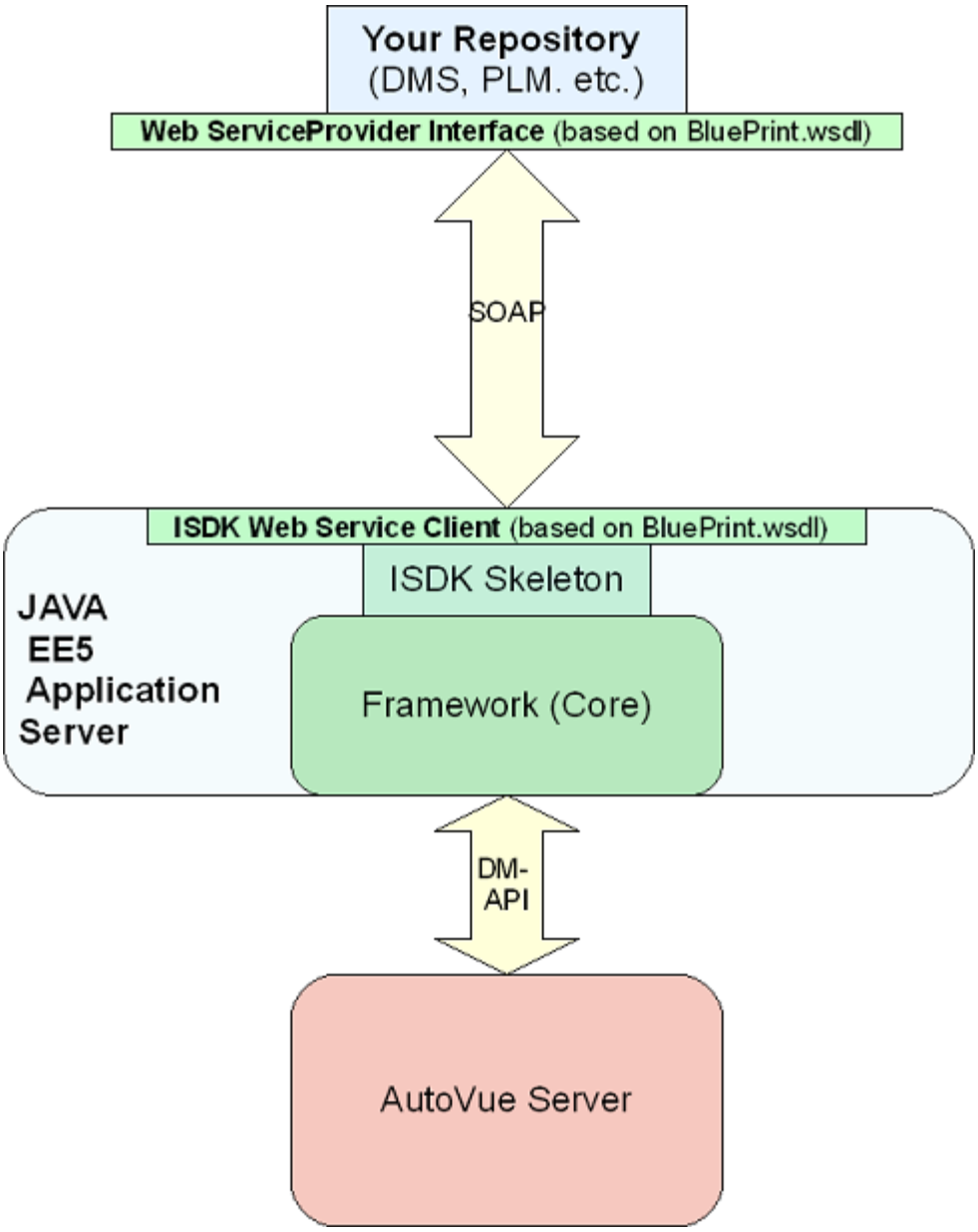


Figure 10-1 Architecture

The benefit of using ISDK Web Service client is that it enables non-Java integration into AutoVue since Web Service communication is a standard XML based protocol. The internal implementation of the Web Service provider on the Repository side can be virtually in any language and on any platform.

Note: If the repository provides any Java API, then it is recommended to use ISDK Skeleton package to build the integration. However, if you are integrating with platforms such as .NET, then it is recommended to use the Web Service client package.

The communication between ISDK Web Service Client and Web Service is based on SOAP (Simple Object Access Protocol), which is a standard protocol.

As a Web Service integrator, the only focus should be on the SOAP channel between your repository and ISDK Web Service client. The blueprint.wsdl and the data model blueprint.xsd are described later in this chapter.

If the repository has security features in place, then it needs to be implemented both on the Web Service and the ISDK Web Service Client package.

By default Web Service Client package has built-in support for two WS-Security policies: HTTPS Basic and HTTPS UserName Token Policy. If your Web Service provider is using one of these two access mechanisms, then communication can be established by enabling proper handler inside the Web Service client package.
If the service provider is using other security mechanisms (for example, certificate, SAML, and so on) then a new handler must be developed and plugged into the Web Service client package.
Web Service client packages provide a flexible mechanism in order to register a new security handler and replace the default behavior. Refer to section "WS-Security" for more information.

10.3 How it Works

As with the Filesys sample, the AutoVue server communicates with the DMS Servlet when accessing the repository. However, the difference is that DMS Servlet relies on ISDK Web Service client to establish communication with the repository using SOAP protocol.

The sequence of activities is similar to what is described in FileSys *Technical Guide,* except that in this case customization needs to be implemented on the repository-side. An example of a simple customization is included in the ISDK Web Service slient package (wsfrmApplet.jsp) which is fairly similar to one included with Filesys (frmApplet.jsp).

For applet parameters in the JSP file, notice that FileName parameter is empty. This id because the parameter must be defined based on what is defined in your repository. For example, it can be an ID number or similar to Filesys they might be a relative path. The bottom line it the FileName parameter is used to find the document on the repository side and construct its proper document ID.

Note: The FileName parameter is empty in the wsfrmApplet.jsp file. This is because the parameter is set by what is defined in the repository. For example, the parameter may be an ID number or a relative path to Filesys.

Assuming the customization is in place and FileName parameters are set, the following is a brief description of how the DMS Servlet works:

1. The client logs into the repository Web Interface and launches AutoVue applet through customization inside the Web browser.

- AutoVue Applet communicates with the AutoVue server through Servlet Tunneling (VueServlet) over an HTTP connection (as defined in the JVUESERVER parameter)

2. The AutoVue server then communicates to the DMS Servlet using a standard HTTP connection (as defined in DMS parameter)

3. The DMS Servlet then uses the ISDK Web Service client package to convert requests to proper Web Service calls. As well, it invokes the Web Service provider on the repository server to handle any request made by the AutoVue Server (such as file fetching).

4. If you try to view a composite file (that is, a file having external references to other files), then DMS Servlet retrieves those files and makes them available to the AutoVue server.

5. Once the file and all related XRefs and/or resources are fetched out of the **DMS**, they are processed by the AutoVue Sserver which renders the files and streams the viewable to the AutoVue applet for display.

6. Once the file displays in the AutoVue applet, you can create new Markups, save Markups into the DMS**,** and open Markups from the DMS.

10.4 Web Service Client Package

The following diagram shows the internal structure of a Web Service client package. This package includes the ISDK core, third-party libraries, and a layer on top of the core that implements the client side for the Blue Print WSDL.

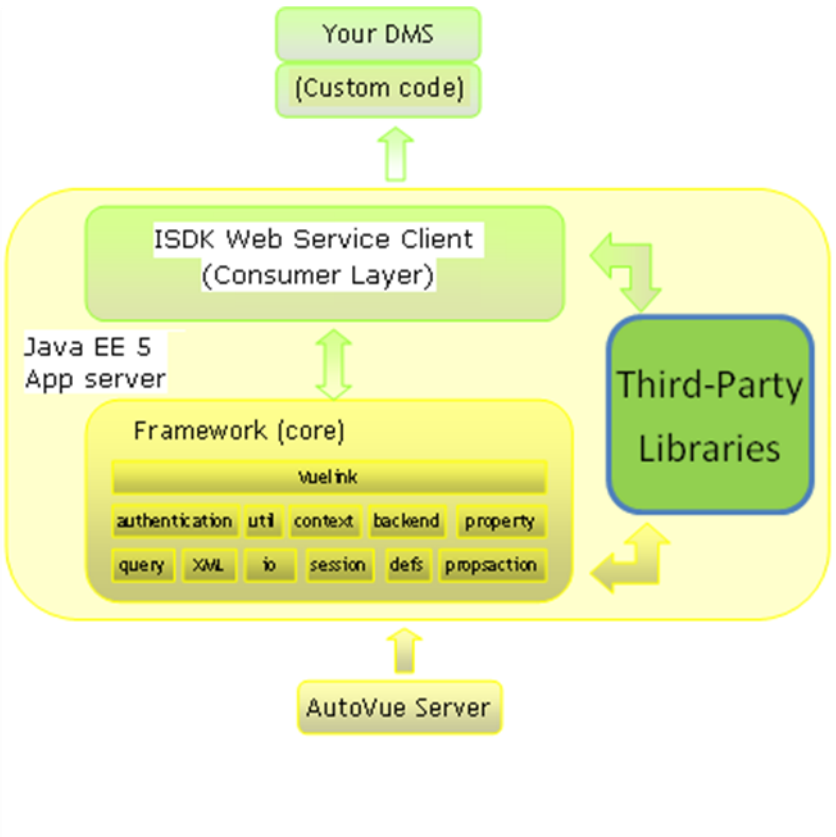


Figure 10-2. Internal Structure of the Web Service Client package

As shown in the figure, the package must be deployed on top of a Java EE 5 application server since the ISDK Web Service client layer depends on the Web Service annotations and JAX-WS (which are part of Java EE 5). Note that Java Standard 6 (JDK 1.6) supports the same Web Service annotations and includes JAX-WS. Unlike the Filesys package, the custom code on the ISDK side is already in place (WS client layer). Additionally, custom code on the repository side is required in order to implement the provider side of Blue Print Web service (blueprint.wsdl).

Note: To implement the custom code you must use the Blue Print WSDL that is described later in this document.

10.5 Sequence

The sequence described here is the same of the section described in Filesys technical document. When a user selects a document to view, the AutoVue server makes several requests to the DMS Servlet. The DMS Servlet provides a response for each request. The scenario of the exchanges established between the AutoVue server and the ISDK package are sketched in the following figure and can be summarized as follows:

- The AutoVue server asks for the docID of the selected document. This is done through the Action Open**,** which obtains the docID from the ISDK.
- The AutoVue server asks for some properties of the document, such as document name, document size and date of the last modification (e.g., sequences 2 and 3 in the

following figure). The reason is that the AutoVue server maintains a cache repository of the document and needs to know if it already has the most recent copy of the document. In which case AutoVue uses the most recent copy rather than downloading the document.

- AutoVue fetches the document through the `Download` Action.

The following sequence diagram shows the flow of communication between AutoVue and your integration, for a typical case of viewing a file from your Repository. As you can see from this diagram, viewing a file triggers many calls to your integration.
Please note the "Your Integration" layer is the combination of Web Service client consumer layer (already included in the package) plus the Web Service provider layer that needs to be done on Repository (e.g. DMS) side.

As you can see by using ISDK web service client package, you are half way through of a SOAP-based integration that has already defined the web service interface, the web methods and the input/output data model.

The above actions are the basic set requests and Reponses between AutoVue and ISDK. There are several other requests/responses that are needed to cover functionalities such as annotation (markups) and collaboration that normally follows the basic set.

10.6 Configuration

Before deploying Web Service Client package (WAR or the open folder) on a JAVA EE5 compliant Application Server you need to update some parameters inside web.xml inside the package.
Note: If your Application Server is using Java 1.6_14 or higher, then the required runtime libraries (JAX-WS 2.1.3+) are already provided by JVM and deployment could be possible on a non EE5 Application Server as long as it is certified to work with Java standard 1.6.

10.7 WSDL Location

You can create a project in Eclipse or JDeveloper by importing the Web Service Client package. Once the project is prepared, open the web.xml file and locate the entry named WSDL (that is, `<param-name>WSDL</param-name>`) then change its associated value (the value inside the <param-value> tag to the actual URL location of web service provider (for example: http://...some sever.../BluePrint?wsdl).
By setting this value, the Web service client package knows where to find the Web service provider.

10.8 WS-Security

Another location to be modified inside web.xml is related to WS-Security. There are several WS security standards defined by the Organization for the Advancement of Structured Information Standards (OASIS). ISDK Web service client package provides out of the box

support for two of these standards: HTTPS- Basic Profile and HTTPS-UserName Token Profile.

While it is easy to enable any of them, none of these two is selected as default in the package because it has to be defined based on the environment. The default setting assumes web service provider is available without any security. Since ISDK is development package it is better test the functionalities first and then enable the security if service provider permits.

10.8.1 HTTPS-Basic Profile

To enable HTTPS- Basic security, first make sure the web service provider is implementing this policy, then locate `<param-name>wsclient.WSHandler</param-name>` inside web.xml and replace its associated value (the value inside its `<param-value>` tag) to `com.cimmetry.vuelink.wsclient.backend.HTTPBasicHandler`
This is the name of the handler class inside the Web Service Client package that will add authentication information to the header of web service requests. The authentication information can be obtained in runtime from the applet.

10.8.2 HTTPS-UserName Token Profile (Metro)

To enable HTTPS-UserName Token, after making sure that the Web service provider is implementing this policy, locate `<param-name>wsclient.WSHandler</param-name>` inside web.xml and replace its associated value (the value inside its `<param-value>` tag) to `com.cimmetry.vuelink.wsclient.backend.UserNameTokenHandler`
This is the name of the handler class inside the Web Service Client package that adds authentication information to the SOAP message requests. The authentication information can be obtained in runtime from the applet.

10.8.3 HTTPS-UserName Token Profile (WebLogic)

If the Web service client package is being deployed on a WebLogic application server, the original class for UserName Token Profile may not work properly. WebLogic server provides some packages that can be used to implement handler for UserName Token Profile. Web Service Client package comes with a Java class that is designed to use WebLogic API. The class is called `WeblogicUserNameTokenHandler` and it is located in the same package as two above classes. Since the class does not work on other application servers (because of WebLogic dependency) it is renamed to `WeblogicUserNameTokenHandler.java.excluded` by default in order to avoid any compilation and runtime error on other application servers.
If you choose to deploy your Web service client on Weblogic, and the security profile between client and Web service provider is UserName Token Profile, then you must rename this class back to Java (by removing `.excluded` from the filename) and making sure Weblogic runtime libraries are available during the compilation. Once there is no compile error, open the web.xml and locate `<param-name>wsclient.WSHandler</param-`

name> inside web.xml and replace its associated value (the value inside its <param-value> tag) to
com.cimmetry.vuelink.wsclient.backend.WeblogicUserNameTokenHandler
By doing so, the WeblogicUserNameTokenHandler is registered as the handler class for UserName Token profile. Its handling is the same as UserNameTokenHandler but instead it directly uses Weblogic API.

10.8.4 Other WS-Security Profiles

If any other type of WS-Security profile is being implemented on Web Service provider (for example, certificate, SAML, and so on) you must write a client side handler and register it into Web Service Client package. The registration is similar to what described above, by setting the class name into wsclient.WSHandler parameter. The important note is that any implementation will require extending WSHandler class that is provided by Web Service Client package. This is true for all three classes that are discussed above.

10.8.4.1 Extending WSHandler

WSHandler class is provided in the same package (com.cimmetry.vuelink.wsclient.backend).
By creating a new class that extends this class, you must replace the implementation of one of the two methods that are provided in WSHandler (depending on where the authentication data is supposed to be).

In most cases the authentication data should be included inside the SOAP message. If this is the case, then the following method should be implemented in your custom handler.

```
public boolean handleMessage(SOAPMessageContext context)
```

Since the input parameter is SOAPMessageContext, any part of the SOAP message can be accessed and modified before it is sent to the server.
The following code snippet shows how this is done in the UserNameTokenHandler class:

```
public boolean handleMessage(SOAPMessageContext context)  {

       m_logger.debug("UserNameTokenHandler handleMessage() called");
       Boolean outboundProperty =
              (Boolean)context.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
       SOAPMessage message =context.getMessage();

       if (outboundProperty.booleanValue()) {
          m_logger.debug("\n (client protocol handler) Outbound message:");

           try {
                String user = (String)connectInfo.get("username");
                if (user != null) {
                      SOAPEnvelope envelope =
                             context.getMessage().getSOAPPart().getEnvelope();
                      SOAPHeader header = envelope.getHeader();
                      if (header == null ) {
                         header = envelope.addHeader();
                      }

                      SOAPElement security = header.addChildElement("Security", "wsse",
                                                                    WSSE_NAMESPACE);
                      SOAPElement usernameToken =
                             security.addChildElement("UsernameToken", "wsse");
                      usernameToken.addAttribute(new QName("xmlns:wsu"),
                                                 WSU_NAMESPACE);
                      SOAPElement username = usernameToken.addChildElement("Username",
                                                 "wsse");
                      username.addTextNode(user);
                      String pass = (String)connectInfo.get("password");
                      if (pass != null) {
                             SOAPElement password =
                             usernameToken.addChildElement("Password", "wsse");
                             password.addTextNode(pass);
                      }
                }
          } catch (Exception e) {
                 m_logger.error("Failed to add username token profile security", e);
          }

       } else {
          m_logger.debug("\n (client protocol handler) Inbound message:");
       }

       if (m_logger.isDebugEnabled()) {
             try {
                //message.writeTo(System.out);  // for testing
                System.out.println("");   // just to add a newline
             } catch (Exception e) {
                m_logger.warn("Exception in soap handler: " , e);
             }
          }

       return true;
       }}
```

If the authentication data should be added to the header of HTTP request (not be confused with SOAP header) then implementing following method should be considered in your custom handler.
`public void handleProxyRequest()`
This method has no input parameter, but you have access to request objects through the Web service proxy object. By obtaining access to request objects you can add authentication information into the request header.
The `handlerProxyRequest()` method is called inside the `connect` method of the backend implementation class right after the handler is set into the chain of handlers.
This should guarantee whatever is defined in this method is executed before the Web service call is made.
The following code snippet shows how this method is implemented in is done in the `HTTPBasicHandler` class**:**

```
public void handleProxyRequest() {
      m_logger.debug("HTTPBasicHandler , handleProxyRequest called");
      if (connectInfo.get("username") == null || connectInfo.get("password") == null) {
            return;
      }
       Map<String, Object> request = ((BindingProvider) proxy).getRequestContext();
            if (connectInfo.get("username") != null) {
                  request.put(BindingProvider.USERNAME_PROPERTY,
                        (String)connectInfo.get("username"));
            }
            if (connectInfo.get("password") != null) {
             request.put(BindingProvider.PASSWORD_PROPERTY,
                        (String)connectInfo.get("password"));
            }
       // this is to maintain any session initiated by server
       request.put(BindingProvider.SESSION_MAINTAIN_PROPERTY, true);

  }
```

Please note that every time web.xml is modified, the application should be redeployed inside the application server for the changes to take effect.

10.9 BluePrint WSDL

This section describes BluePrint WSDL. This WSDL is provided in the Web Service Client package and the client package implementation is based on this WSDL (blueprint.wsdl) and the XSD file (blueprint.xsd) that accompanies it. These two files should be used for implementing the Web Service provider that the Web Service package communicates with.

10.9.1 Web Services Methods

This section provides a review of the available Web Services methods inside wsdl file

Note: For information on non-standard data structure refer to section "BluePrint XSD".
In following Web Services method description, if Parameters and Returns are an array, they could be List in some implementation.

Web Services Method	Description
delete	Delete a markup document in backend repository **Parameters:** arg0: WsDocID – A unique identifier of a markup in your integration. arg1: SessionData – Session information used to connect to backend repository **Returns:** boolean: Returns TRUE if deletion successfully, otherwise FALSE.
download	Download original file, markup file, supported file (for example, XRefs), and so on from backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a downloading document in your integration. arg1: SessionData – Session information used to connect to backend repository **Returns:** byte[]: Content of the file.
openFile	Get a document ID for a given document. **Parameters:** arg0: String – Information to identify a document in backend repository. arg1: SessionData – Session information used to connect to backend repository **Returns:** WsDocID: A unique document identifier in your integration.
setAttributes	Set a given document or collaboration properties in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: Attribute[] – An array of Attribute objects. Each element contains name and value(s) per attribute which will be modified in backend repository. arg2: SessionData – Session information used to connect to backend repository **Returns:** boolean: Returns TRUE if set properties successfully, otherwise FALSE.
getUserName	Get current user name who connects to backend repository **Parameters:** arg0: SessionData – Session information used to connect to

	backend repository **Returns:** String: Current user name.
getAllAttributes	Get all available properties of a given document in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: SessionData – Session information used to connect to backend repository **Returns:** A DocAttribute object that contains all the properties of a document in backend repository.
getDmsConfig	Get all basic settings of backend repository. **Parameters:** **Returns:** A DmsConfig object that contains basic settings for backend repository: the function of Browse repository supported or not, the function of Search repository supported or not, the function of Redirect supported or not, and a String value used for customizing Browse and Search button
getBasicAttributes	Get basic properties of a given document in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration arg1: SessionData – Session information used to connect to backend repository **Returns:** A BasicAttribute object that contains basic properties for a document in backend repository (for example, document ID, name, size, last modified date, and so on)
getRevisions	Return all other revisions for a given document in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: SessionData – Session information used to connect to backend repository **Returns:** Return an array of RelatedDocument objects. One element represents a revision for a given document.
getXrefs	Get external reference files for a given document. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: SessionData – Session information used to connect to

	backend repository **Returns:** An array of RelatedDocument objects. One element represents an external reference file.
getRendition	Get all supported rendition formats and streaming file document ID if it exists in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: SessionData – Session information used to connect to backend repository **Returns:** An Rendition object that contains a list supported rendition formats and document ID of the streaming file
isBrowseEnabled	Return whether backend repository supports browse function or not. **Parameters:** **Returns:** TRUE if backend repository supports browse function, otherwise FALSE.
isSearchEnabled	Return whether backend repository supports search function or not. **Parameters:** **Returns:** TRUE if backend repository supports search function, otherwise FALSE.
isRedirect	Return whether backend repository is a distributed environment that can redirect AutoVue Download/Save requests to another ISDK-based integration deployed on remote server. **Parameters:** **Returns:** TRUE if backend repository supports redirect function, otherwise false.
listMarkup	Return all markup documents associated with a given document in backend repository. **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: Field [] – An array of Field objects. Each element represents one attribute displayed in Markup Open dialog. Each item in the return array should include the values for the attributes specified by this argument. arg2: SessionData – Session information used to connect to backend repository **Returns:** An array of MarkupList objects. Each element represents a markup document.

saveMarkup	Save a markup for a given document to backend repository **Parameters:** arg0: byte [] – Markup file content arg1: WsDocID – A unique identifier of a base document in your integration. arg2: WsDocID – A unique identifier of a markup document in your integration. arg3: String – Name of markup file arg4: String – Type of markup file (for example, normal, master, consolidated) arg5: Field[] – An array of Field objects. One element represents one property (name/values) of the markup file. arg6: SessionData – Session information used to connect to backend repository. **Returns:** The newly saved markup document ID.
saveRendition	Save rendition file for a given document to backend repository **Parameters:** arg0: byte [] – Rendition file content arg1: WsDocID – Unique identifier of a base document in your integration. arg3: String – The type of rendition file (for example, PCRS_TIF) arg4: SessionData – Session information used to connect to backend repository **Returns:** The newly saved rendition document ID.
saveChat	Save chat content created during real-time collaboration meeting to backend repository. **Parameters:** arg0: byte [] – Chat content arg1: String – Real-time collaboration session data arg2: SessionData – Session information used to connect to backend repository **Returns:** The newly saved chat document ID.
getIntellistamp	Get Stamp definition file and background image files inside the definition file **Parameters:** arg0: SessionData – Session information used to connect to backend repository **Returns:** A Stamp object that contains Stamp definition file and an array of RelatedDocument objects, each element represents one background image
getMarkupPolicy	Get markup policy file that controls the markup creation, modification and deletion.

	Parameters: arg0: SessionData – Session information used to connect to backend repository **Returns:** A string that contains markup policy file.
getClbSessionID	Get real-time collaboration meeting session ID from collaboration data. **Parameters:** arg0: String – Real-time collaboration data arg1: SessionData – Session information used to connect to backend repository **Returns:** A string that represents real-time collaboration session ID.
clbCloseMeeting	Process information in backend repository when real-time collaboration meeting is finished. **Parameters:** arg0: String – Real-time collaboration data arg1: String – Value to be processed. arg2: SessionData – session information used to connect to backend repository **Returns:** True if successfully, otherwise false.
clbDocumentSet	Process information in backend repository when collaboration users switch documents to collaborate on in the middle of a RTC meeting. **Parameters:** arg0: String – Real-time collaboration data arg1: String – S string represents document ID. arg2: SessionData – Session information used to connect to backend repository **Returns:** TRUE if successful, otherwise FALSE.
clbInitSession	Process information in backend repository when real-time collaboration meeting is started **Parameters:** arg0: String – Real-time collaboration session data arg1: String – Value to be processed in backend repository. arg2: SessionData – Session information used to connect to backend repository **Returns:** TRUE if successful, otherwise FALSE.
clbGui	Specify real-time collaboration GUI properties. **Parameters:** arg0: String – real-time collaboration session data arg1: SessionData – session information used to connect to backend repository

	Returns: A RtcGui object that contains a RtcDisplayOption object used for enabling/disabling GUI items in Invitation dialog and an array of Field objects used for listing attributes to be displayed in Session Information dialog.
getRtcCollaboration	Get real-time collaboration information (i.e. the users to be invited, invited user, and collaboration session information) **Parameters:** arg0: String – Real-time collaboration session data arg1: Field [] – An array of Field objects. One element represents one attribute to be displayed in Session Information dialog. Each RtcSession in the return should get the values for the attributes specified by this argument. arg2: SessionData – Session information used to connect to backend repository **Returns:** A RtcCollaboration object that contains a list of users to be invited, a list of already invited users, and an array of collaboration session information.
clbUserJoined	Process information in backend repository when a user joins the real-time collaboration meeting. **Parameters:** arg0: String –Real-time collaboration session data arg1: String – Value (name of joined user) arg2: SessionData – Session information used to connect to backend repository **Returns:** True if successfully, otherwise false.
clbUserLeft	Process information in backend repository when a user leaves the real-time collaboration meeting. **Parameters:** arg0: String – Real-time collaboration session data arg1: String – Value (the name of left user) arg2: SessionData – Session information used to connect to backend repository **Returns:** TRUE if successful, otherwise FALSE.
clbMarkupSaved	Process information in backend repository when host saves markup for the collaboration session. **Parameters:** arg0: String – Real-time collaboration session data arg1: String – Value (markup name) arg2: SessionData – Session information used to connect to backend repository **Returns:** TRUE if successful, otherwise FALSE.

getGUIDMS	Get the value used for customizing Browse and Search button. **Parameters:** **Returns:** A String.
dmsBrowse	Return all the items that are direct children of a node (e.g. folder) **Parameters:** arg0: WsDocID – A unique identifier of a parent folder in your integration. arg1: Field [] – An array of Field objects. Each element represents an attribute displayed in Browse dialog. Each item in the return array should get the values for the attributes specified by this argument. arg2: SessionData – Session information used to connect to backend repository **Returns:** An array of DocList objects. Each element represents a child node (for example, a folder or document).
getSearchCriteria	Specify search criteria. **Parameters:** **Returns:** An array of Attribute objects. Each element represents one search criteria (for example, Name and possible values) in Search dialog.
dmsSearch	Return all the items that meet search criteria **Parameters:** arg0: WsDocID – A unique identifier of a document in your integration. arg1: Field [] – An array of Field objects. Each element represents an attribute displayed in Search dialog. Each item in the return array should get the values for the attributes specified by this argument. arg2: Attribute [] – An array of Attribute objects. Each element contains one name and value(s) per search criteria arg3: SessionData – Session information used to connect to backend repository **Returns:** An array of DocList objects. Each element represents a child node (for example, a folder or document).
getSearchGui	Specify the attributes that are displayed in Search dialog **Parameters:** arg0: SessionData – Session information used to connect to backend repository **Returns:** An array of Field objects. Each element holds a property's name displayed on Search dialog and display length for the property.
getBrowseGui	Specify the attributes that are displayed in Browse dialog

	Parameters: arg0: SessionData – Session information used to connect to backend repository **Returns:** An array of Field objects. Each element holds a property's name displayed in Browse dialog and display length for the property.
getMarkupGui	Specify the attributes that are displayed in Markup Open dialog and Markup Save dialog. **Parameters:** arg0: SessionData – Session information used to connect to backend repository **Returns:** A markup object that contains the information used Markup Open dialog and Markup Save dialog.

10.9.2 BLUEPRINT XSD

This section provides a review of all classes that represent custom outputs and custom inputs for different Web Services methods.

Note: In following description for custom data structures, if attribute is an array, they could be List in some implementation.

Attribute	A property of a document in backend repository. **Attributes:** name: String – Property name values: String [] – Values for the properties isMultiValues: boolean – Is multi-value property or not optionList: OptionList – A object contains predefined values that user can select.
BasicAttribute	Basic properties about a document in backend repository. **Attributes:** docID: WsDocID – A unique identifier of a document in your integration. name: String – Document name size: String – Document size lastModifiedDate: String – Last modified date of the document multiContent: String – How many files are contained in the document. folder: String – Folder name of the document.
DmsConfig	Basic settings for a backend repository.

	Attributes: isBrowseEnabled : boolean – Backend repository supports browse function or not; isSearchEnabled: boolean – Backend repository supports search function or not; isRedirect: boolean – Whether backend repository is a distributed environment that can redirect AutoVue Download/Save requests to another ISDK-based integration deployed on remote server or not. dmsGui: String – Text used for customizing Search/Browse button
DocAttribute	It is a subclass of BasicAttribute. It holds all properties of a document **Attributes:** optionalFields: Attribute [] – An array of Attribute objects. Each element holds one property other than basic properties(for example, name, size) of a document.
DocList	It contains information about a document in backend repository and is used as return type of the methods *dmsSearch* and *dmsBrowse.* **Attributes:** docID: WsDocID – An identifier of a document in your integration. name: String – Document's name optionField: Field [] – An array of objects. Each element represent one property (for example, name/value) of a document.
Field	Represent one property object with name/value. **Attributes:** name: String – Property name value: String – Property value
`Intellistamp`	Represents the return type of the method of `getIntellistamp()` **Attributes:** definition: String – The content of Stamp definition file. image: RelatedDocument []: An array of RelatedDocument objects. Each element represents a background image inside Stamp definition file.
`MarkupDisplayOption`	It is used by the method `getMarkupGui()`. It specifies whether or not users are allowed to perform some operations on markups in Markup Open dialog. **Attributes:** allowDelete: boolean –- Can delete markup or not? showPreviousVersions: boolean – Can display the markups from other version of the document or not?

	allowNew: boolean – Can create a new markup or not? allowImport: boolean – Can import a markup or not? allowExport: boolean – Can export a markup or not? allowNewLayers: boolean – Can create a new layer for a markup or not? allowModifyLayers: boolean – Can modify a layer of a markup or not?
`MarkupGui`	It is used as return type of the method `getMarkupGui()`. It specifies the structure of the GUIs for markup with which the user will interact. The GUI part itself is composed of three sections: Display Options, Edit, and Display. **Attributes:** displayOption: MarkupDisplayOption – Specifies whether or not users are allowed to perform some operations on markups in Markup Open dialog; displayLabel: Field [] – An array of Field objects. Each element holds a property's name displayed in Markup Open dialog and display length for the property. editAttribute Attribute [] – An array of Attribute objects. Each element represents an attribute whose value user should specify in Markup Save dialog;
`MarkupList`	It is a subclass of class DocList and used as return type of the method `listMarkup()`. It contains the properties about a markup document. **Attributes:** readOnly: boolean – Is a markup read-only? baseRevision : String – Version of the base document to which a markup is attached markupType: String – Normal/master/consolidated
`OptionList`	It is used to specify predefined values for a property of a document in backend repository. **Attributes:** isFixed: boolean – If it is TRUE, cannot add other value to the predefined list? Otherwise FALSE. options: String [] – An array of String. Each element represents a value in predefined list.
`RelatedDocument`	It is mainly used as return for the methods `getRevisions()` and `getXrefs()`. **Attributes:** docName: String – Document name WsDocID docID – A unique identifier of a document in your integration.
`Rendition`	It is used as return type of the method `getRendition()`. **Attributes**: supportedRenditions : rendType [] – An array of objects. Each

	element represents one rendition format (for example, CSI_META, PCRS_TIF) wsDocID : WsDocID – the identifier of a streaming file in your integration.
`RtcCollaboration`	It is used for the method `getRtcCollaboration()` and contains the information about a real-time collaboration meeting. **Attributes:** userToBeInvited: String [] – List of users to be invited to the meeting. userInvited: String [] – List of users are already in the meeting. rtcSession: RtcSession [] – An array of objects. Each element represents the information per real-time collaboration meeting;
`RtcDisplayOption`	It is used for enabling/disabling GUI items in Invitation dialog. **Attributes:** allowAdd: boolean – Can add a user? allowAddNew: boolean – Can add a new user? allowRemove: boolean – Can remove a user? allowLayerColor: boolean – Can modify layer's color?
`RtcGui`	It is used as return type of the method `clbGui()`. **Attributes:** displayOption: RtcDisplayOption – Enable /disable real-time collaboration meeting in Invitation dialog; displayLabel: Field [] – An array of objects. Each element holds a property's name displayed in Session Information dialog and display length for the property.
`RtcSession`	It represents session information such as session title, id, type, subject, duration, start time, and so on. **Attributes:** clbSessionId: String – real-time collaboration session ID; clbSessionTypeIsPublic: boolean – TRUE if it is public, otherwise FALSE. clbSaveChat: boolean – TRUE if the backend system component supports saving chat transcript. label: Field[]– An array of Field objects. Each element represents a property's name and its value displayed in Session Information dialog.
`SessionData`	It represents session information needed to connect to backend repository. **Attributes:** expired: boolean – TRUE if it is invalid. data: Field[] – An array of Field objects that are needed to connect to backend repository.
`WsDocID`	An unique identifier of a document in your integration **Attributes:** id: String – A unique identifier of a document in your backend

	repository version: String – Version number assetID :String – Asset ID associated with the document workflowID: String – Workflow ID associated with the document isFolder: boolean – Document is folder?

10.10 Steps for Implementing BASIC Integration Based on Web Services

This section outlines the minimum Web Services methods which are defined in BluePrint.wsdl that should be implemented on Web Services provider side in order to add file view capabilities using Web Service package with AutoVue.

- `getDmsConfig()`
- `openFile()`
- `getBasicAttributes()`
- `download()`

Other Web Services methods are not necessary to be implemented and you can just provide **null** as the return value for them.

Integration SDK Web Service Client project includes a sample backend extension file (wsfrmApplet.jsp) in the *applet* folder for launching AutoVue. You should modify it or create your own backend extension file (for example, a .asp file) and put it in correct location according to your backend system. The user can click a button in backend system UI to launch the file in the AutoVue applet.

In the backend extension file, do the following:

- Provide the FILENAME variable with your unique document identifier.
- Provide the JVUESERVER variable with your VueServlet (for example. http://hostname:port/servlet/VueServlet).
- Provide the DMS variable with your Web Services client DMS (for example, http://hostname:port/servlet/DMS).

10.11 Steps for Implementing Advanced Integration Based on Web Services

To implement additional functionality such as XRefs, markups, compare, renditions, DMS Search/Browse, and so on you should implement the rest of the methods listed in section "**Web Services Methods**". It is assumed that you have already implemented the file view functionality in your backend system as outlined in previous section.

10.12 Sample Approaches to Generate Web Services Provider Artifacts

10.12.1 How to generate Java web services code from ISDK WS WSDL file

To generate Java Web services code, call `wsimport` from the command line with the `-keep` option and pass the WSDL's file: `wsimport -keep` *`wsd_file-location`*

For example: `wsimport -keep L:\temp\WebServiceClient\WSDL\BluePrint.wsdl`

10.12.2 How to generate .Net web services code from ISDK WS WSDL file

Enter the following command line: `wsdl.exe /Language:CS /si wsdl_location xsd_location`

Then open the file that you just generated, locate the following line and then change `Name` from `BluePrintBinding` to `BluePrint`.

```
[System.Web.Services.WebServiceBindingAttribute(Name="BluePrintBinding",
Namespace="artifact.wsclient.vuelink.cimmetry.com")]
```

After you generate the Web services server artifacts using either of the above approaches, you should create a class to implement each Web services method.

10.13 BluePrint WSDL and XSD

You can access the BluePrint.wsdl and BluePrint.xsl files from the <ISDK install folder>WebServiceClient\WSDL directory. Refer to the "Installation" section of the *Installation Guide* for more information on the location of the files.

11. APPENDIX D – ISDK WEB SERVICES SAMPLE SERVER

The Web Services Sample Server project is a sample implementation of the Web Services provider in the C# language and uses the Filesys repository as the backend DMS. For general information on implementing integrations with the Web Services provider, refer to section "Steps for Implementing Basic Integration Based on Web Services" and "Steps for Implementing Advanced Integration Based on Web Services".

This sample server implements the Web Services methods defined in the BluePrint WSDL file. For information .refer to "BluePrint WSDL".

The ISDK Web Services Sample Server project is located under the WebServiceIntegration/WebSserviceSampleFolder folder. Refer to the *ISDK Installation and Configuration Guide* for more information.

Source code for implementing the Web Services sample server, Service1.asmx.cs, is provided in the <ISDK Installation Directory>\WebServicesIntegration\WebServicesSampleServer\C# directory.

12. APPENDIX E - UPGRADING EXISTING INTEGRATION

This section is intended for anyone who has built an integration based on a pre-20.2 version AutoVue Integration SDK and is going to upgrade the existing integration to work with AutoVue release 20.2 and the AutoVue Integration SDK framework of this release (vuelinkcore.jar).

12.1 Upgrading from the 20.1 Release

1. Replace vuelinkcore.jar in WEB-IN/lib folder with the new one.
2. Replace vueservlet.jar in WEB-IN/lib folder with the vueservlet.jar in AutoVue 20.2 bin folder.
3. Replace jvue.jar, jogl.jar and gluegen-rt.jar in the jvue folder with the files of the same names in AutoVue 20.2 bin folder.
4. Run the ISDK 20.2 installer to a different installation folder than your pre 20.2 installation.
5. Copy the esapi-2.0.1.jar file from the <ISDK 20.2 Installation Directory>\ ISDKSkeleton\WebApplication\isdk_skeleton\WEB-INF\lib to your integration's WEB-INF\lib directory.
6. Note that file path names are case-sensitive. As a result, you must make sure that the file paths defined in the web.xml file are correct.
7. Copy and configure the ESAPI property files as described in the ISDK Security Guide.

12.2 Upgrading from a pre-20.1 Release

1. Replace vuelinkcore.jar in WEB-IN/lib folder with the new one.
2. Replace vueservlet.jar in WEB-IN/lib folder with the vueservlet.jar in AutoVue 20.2 bin folder.
3. Replace jvue.jar, jogl.jar and gluegen-rt.jar in the jvue folder with the files of the same names in AutoVue 20.2 bin folder.

4. Update your own DocID implementation class (for example, FilesysDMSDocID in the Sample Integration for Filesys and ISDKDocID in the SDK Skeleton):
   - Changing the class declaration from implementing the DocID interface to extending the DocID abstract class.

```
public class MyDocID extends DocID implements DMSDefs { …}
```

   - Overwrite two new methods:
     `public String DocID2String();`
     `public FilesysDMSDocID String2DocID(String docid);`

     For example, the Integration SDK Skeleton has the following implementation in `com.mycompany.autovueconnector.defs.ISDKDocID`.

```
public String DocID2String() {
        // TODO Return all fields information as a string with separator
        return m_sDocID;
}

public ISDKDocID String2DocID(String sDocID) {
        if (sDocID == null) {
                return null;
        }

        ISDKDocID docID = new ISDKDocID(sDocID);
        return docID;
}
```

5. Replace all method calls to `query.getDocID()` in your integration to `new MyDocID().String2DocID(query.getDocID())`. It is because the `query.getDocID()` method returns a String representation of the DocID instead of the DocID object in the DMSQueryImp class of the new framework. Here *`MyDocID`* is your own DocID implementation. These replacements are located in actions and propactions packages.

6. (Optional) In actions and propactions package, replace all class declaration and the first parameters of the `execute()` method to eliminate casting of context in your code.

   The `com.cimmetry.vuelink.propaction.DMSAction` and the `com.cimmetry.vuelink.propaction.DMSGetPropAction` interface in the new framework use generic class declarations and new signature for `execute()` method as below. A covariat parameter type AnyContext is used instead of the original DMSContext parameter in the `execute()` method.

```
package com.cimmetry.vuelink.propsaction;
                                                …
public interface DMSAction<AnyContext extends DMSContext> {
   …
   public Object execute( final AnyContext    context,  //covariat parameter type
                          final DMSSession    session,
                          final DMSQuery      query,
                          final DMSArgument[] args
                          ) throws VuelinkException;
}
```

```
package com.cimmetry.vuelink.propsaction;
…
public interface DMSGetPropAction<AnyContext extends DMSContext> {
   …
   public Object execute( final AnyContext    context,
                          final DMSSession    session,
                          final DMSQuery      query,
                          final DMSArgument[] args,
                          final Property      property
                          ) throws VuelinkException;
}
```

You can change your code to make use of this new functionality. If you do not make this change, your original code still compiles.

If you change your code, it should be similar to the following code snippet from the Integration SDK Skeleton and you can use your own context class instead of the DMSContextImp of the Skeleton.

```
public class ActionDelete implements DMSAction<DMSContextImp>, DMSDefs {

    private static final Logger m_logger = LogManager.getLogger(ActionDelete.class);

    @Override
    public Object execute(
                final DMSContextImp context,
                final DMSSession session,
                final DMSQuery query,
                final DMSArgument[] args
               ) throws VuelinkException {
          … // use of the context variable directly without casting to DMSContextImp
     }
}
```

```
public class GetPropCSI_DocName implements DMSGetPropAction<DMSContextImp>, DMSDefs {

    private static final Logger m_logger = LogManager.getLogger(ActionDelete.class);

    @Override
    public DMSProperty execute(
                DMSContextImp context,
                DMSSession session,
                DMSQuery query,
                DMSArgument[] args,
                Property property
                ) throws VuelinkException {
          … // use of the context variable directly without casting to DMSContextImp
     }
}
```

7. The framework in this release drops support for the vuelink.properties file. You need to do the following:
   - Move all the properties defined in your vuelink.properties to web.xml as init-param for your servlet.

     For example, you can add an initial parameter MyPropertyMoved in web.xml that is originally defined in vuelink.properties.

```
<servlet id="csi_servlet_1">
 <servlet-name>DMS</servlet-name>
 <servlet-class>com.mycompany.autovueconnector.DMS</servlet-class>
 …
 <init-param>
       <param-name>MyPropertyMoved</param-name>
       <param-value>MyPropertyValue</param-value>
 </init-param>
 …
</servlet>
```

   - Any call of `getVuelinkPropByName(String name)` method in your code should be replaced with `getInitParameter(String paramName)` method of the your context class.

     All the initial parameters defined in web.xml for your main servlet are automatically picked up by the framework and are saved in a hash table of your context designated to hold all the initial parameters of your context, for example, in the `com.cimmetry.vuelink.context.GenericContext` class, it is the m_initParamters variable. This hash table can be retrieved and set through method calls of the context class. To save your effort, your context class should extend the GenericContext class.
     Here a sample method call to get the value of the RootDir parameter defined in web.xml.

```
package com.cimmetry.vuelink.filesys.actions;
…
public class ActionOpen implements DMSAction<FilesysContext>, DMSDefs {   …
   public Object execute(final FilesysContext  context,
                         final DMSSession     session,
                         final DMSQuery       query,
                         final DMSArgument[] args
                         ) throws VuelinkException {
       …
       String rootDir = context.getInitParameter("RootDir");
       …
   }
}
```

   - Any reference to the vuelinkProp variable (defined by the framework Vuelink servlet previously) should be replaced since this variable is no longer available.

     Thus any call of `vuelinkProp.setProperty(String name, String Value)` method should be replaced if you had code in your existing integration to update the vuelinkProp variable after the properties has been retrieved.

- You can call `context.setInitParameter(String name, String value)` method if your context object is available.
- If you have to modify it during the main servlet initialization stage, you can realize the same functionality by overwriting the `saveInitParameter(String name, String value)` method in your Context class. For example, if you need to update the value of MyPropertyMoved parameter previously in the `init()` method of your main servlet code, you can do it now in your context class similar to the follow code:

```
package com.mycompany.autovueconnector.context;
…
public class DMSContextImp extends GenericContext {
   …
   public void saveInitParams(ServletConfig config, ServletContext context) {
        super.saveInitParams(config, context);

        String value = getInitParameter("MyPropertyMoved"));
        String newValue = …; // Process the value
        if (newValue != null) {
            setInitParameter("MyPropertyMoved ", newValue);
        }
   }
   …
}
```

13. FEEDBACK

If you have any questions or require support for AutoVue please contact your system administrator. If at any time you have questions or concerns regarding AutoVue, please contact us.

General AutoVue Information:

Telephone: +1.514.905.8434 or 1.800.363.5805
Web Site: http://www.oracle.com/us/products/applications/autovue/index.html
Blog: http://blogs.oracle.com/enterprisevisualization/

Oracle Customer Support:
Web Site: http://www.oracle.com/support/index.html

My Oracle Support AutoVue Community:
Web Site: https://communities.oracle.com/portal/server.pt

Sales Inquiries:
E-mail: autovuesales_ww@oracle.com