DOCUMENTATION

Developer Guide

Kablink® Vibe™ OnPrem

3.1

June 28, 2011

Legal Notices

Novell, Inc., makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc., reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc., makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc., reserves the right to make changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. See the Novell International Trade Services Web page (http://www.novell.com/info/exports/) for more information on exporting Novell software. Novell assumes no responsibility for your failure to obtain any necessary export approvals.

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.
www.novell.com

Online Documentation: To access the latest online documentation for this and other Novell products, see the Novell Documentation Web page (http://www.novell.com/documentation/team_plus_conf/).

Novell Trademarks: For Novell trademarks, see the Novell Trademark and Service Mark list (http://www.novell.com/company/legal/trademarks/tmlist.html).

Third-Party Materials: All third-party trademarks are the property of their respective owners.

Contents

About This Manual 9

1 Web Services Overview 11

1.1 Vibe Web Services Terminology . . . 11
1.2 Web Services Implementation . . . 12
1.2.1 Java Web Services . . . 12
1.2.2 Sample Clients . . . 13
1.3 Authentication . . . 14
1.3.1 HTTP Basic Authentication Access (ssr) . . . 14
1.3.2 Web Services Security Access (ssf) . . . 14
1.4 Server Endpoints . . . 15
1.5 Categories of Operations . . . 15
1.6 Client Stubs . . . 16
1.7 Managing Data . . . 16
1.7.1 Working with Java Objects . . . 17
1.7.2 Adding Folders and the Binder Configuration Identifier . . . 18
1.7.3 Attaching Files . . . 19
1.7.4 Fetching Attachments . . . 20
1.7.5 Adding Calendar Entries . . . 20
1.7.6 Binder Pages and search_getWorkspaceTreeAsXML . . . 20
1.8 Extending Vibe Web Services . . . 22

2 Creating and Packaging Extensions for Deployment 23

2.1 Understanding the Differences between Extensions and Remote Applications . . . 23
2.2 Creating an Extension . . . 24
2.3 Packaging an Extension . . . 24
2.3.1 Examples of the Archive Format . . . 25
2.3.2 Extension Metadata . . . 26
2.4 Deploying an Extension . . . 26
2.4.1 Deploying an Extension from the Vibe Interface . . . 26
2.4.2 Deploying an Extension from the Vibe Server . . . 26
2.5 Updating an Extension . . . 26
2.6 Locating an Extension in the Vibe Directory Structure . . . 27
2.7 Retaining an Extension When Updating Your Vibe Software . . . 27

3 Remote Applications 29

3.1 Understanding the Differences between Extensions and Remote Applications . . . 29
3.2 Remote Application Overview . . . 29
3.2.1 Processing Flow for a Remote Accessory . . . 30
3.2.2 Processing Flow for a Remote Form . . . 32
3.2.3 Setting Access Control for Remote Applications . . . 33
3.2.4 Reviewing Supporting Source Code . . . 34
3.3 Creating a Remote Application . . . 34
3.3.1 Reviewing the Class File . . . 35
3.3.2 Reviewing the Servlet-Definition File . . . 36
3.3.3 Reviewing the JSP File . . . 36
3.4 Related Sections . . . 37

3.4.1 Registering a Remote Application . . . 37
3.4.2 Configuring an Accessory to Show a Remote Application . . . 37
3.4.3 Controlling the Access of Remote Applications . . . 37

4 Creating JavaServer Pages (JSPs) 39

4.1 Overview of JSP Support . . . 40
4.1.1 Directory Structure . . . 40
4.1.2 Applicable Pages . . . 41
4.1.3 JSPs and the Vibe Designers . . . 41
4.1.4 Indexing Issues . . . 44
4.1.5 JSPs and Vibe Data Access . . . 44
4.1.6 Text Display in the HTML Editor . . . 45
4.1.7 Standard Styles . . . 46
4.2 Examples of Custom Entries . . . 46
4.2.1 A JSP That Defines Only One Data Element . . . 46
4.2.2 A JSP-Defined Entry (W-4 Form) . . . 51
4.3 Examples of Complex, HTML Data Types . . . 65
4.3.1 Radio Buttons . . . 66
4.3.2 Check Boxes . . . 66
4.3.3 Select Boxes . . . 67

A Web Services Operations 69

admin_destroyApplicationScopedToken . . . 72
admin_getApplicationScopedToken . . . 72
binder_addBinder . . . 73
binder_copyBinder . . . 74
binder_deleteBinder . . . 74
binder_deleteTag . . . 75
binder_getBinder . . . 76
binder_getBinderByPathName . . . 77
binder_getFileVersions . . . 77
binder_getFolders . . . 78
binder_getSubscription . . . 79
binder_getTags . . . 80
binder_getTeamMembers . . . 81
binder_getTrashEntries . . . 81
binder_indexBinder . . . 82
binder_indexTree . . . 83
binder_modifyBinder . . . 84
binder_moveBinder . . . 84
binder_preDeleteBinder . . . 85
binder_removeFile . . . 86
binder_restoreBinder . . . 86
binder_setDefinitions . . . 87
binder_setFunctionMembership . . . 88
binder_setFunctionMembershipInherited . . . 89
binder_setOwner . . . 89
binder_setSubscription . . . 90
binder_setTag . . . 91
binder_setTeamMembers . . . 92
binder_testAccess . . . 92
binder_uploadFile . . . 93

definition_getDefinitionAsXML 94
definition_getDefinitionByName 95
definition_getDefinitions 95
definition_getLocalDefinitionByName 96
definition_getLocalDefinitions 97
folder_addEntry 98
folder_addEntryWorkflow 99
folder_addMicroBlog 99
folder_addReply 100
folder_copyEntry 101
folder_deleteEntry 101
folder_deleteEntryTag 102
folder_deleteEntryWorkflow 103
folder_getEntries 103
folder_getEntry 104
folder_getEntryByFileName 105
folder_getEntryTags 106
folder_getFileVersions 106
folder_getSubscription 107
folder_modifyEntry 108
folder_modifyWorkflowState 108
folder_moveEntry 109
folder_preDeleteEntry 110
folder_removeFile 110
folder_reserveEntry 111
folder_restoreEntry 111
folder_setEntryTag 112
folder_setRating 112
folder_setSubscription 113
folder_setWorkflowResponse 114
folder_synchronizeMirroredFolder 115
folder_unreserveEntry 115
folder_uploadFile 116
folder_uploadFileStaged 117
ical_uploadCalendarEntriesWithXML 118
ldap_synchAll 119
ldap_synchUser 119
license_getExternalUsers 120
license_getRegisteredUsers 121
license_updateLicense 121
migration_addBinder 122
migration_addBinderWithXML 122
migration_addEntryWorkflow 123
migration_addFolderEntry 124
migration_addFolderEntryWithXML 125
migration_addReply 126
migration_addReplyWithXML 127
migration_uploadFolderFile 128
migration_uploadFolderFileStaged 130
profile_addGroup 131
profile_addGroupMember 132
profile_addUser 132

profile_addUserWorkspace 133
profile_deletePrincipal 134
profile_getFileVersions 134
profile_getGroup 135
profile_getGroupByName 136
profile_getGroupMembers 137
profile_getPrincipals 137
profile_getUser 138
profile_getUserByName 139
profile_getUsers 140
profile_getUserTeams 141
profile_modifyGroup 141
profile_modifyUser 142
profile_removeFile 142
profile_removeGroupMember 143
profile_uploadFile 144
search_getFolderEntries 145
search_getTeams 146
search_getWorkspaceTreeAsXML 146
search_search 147
template_addBinder 148
template_getTemplates 149
zone_addZone 150
zone_deleteZone 151
zone_modifyZone 151

B Deprecated Web Services Operations 153

addFolder 155
addFolderEntry 155
addReply 157
addUserWorkspace 158
getAllPrincipalsAsXML 159
getDefinitionAsXML 160
getDefinitionConfigAsXML 161
getDefinitionListAsXML 161
getFolderEntriesAsXML 162
getFolderEntryAsXML 163
getPrincipalAsXML 164
getTeamMembersAsXML 165
getTeamsAsXML 165
getWorkspaceTreeAsXML 166
indexFolder 167
migrateBinder 168
migrateEntryWorkflow 169
migrateFolderEntry 171
migrateFolderFile 172
migrateFolderFileStaged 173
migrateReply 175
modifyFolderEntry 177
setDefinitions 177
setFunctionMembership 178

setFunctionMembershipInherited 180
setOwner 180
setTeamMembers 181
synchronizeMirroredFolder 182
uploadCalendarEntries 183
uploadFolderFile 184

C Migrating from Forum to Kablink Teaming 187

C.1 Sequence of Migration Operations 187
C.2 Migration Overwrite Operations 188
C.3 Migrating Users 188
C.4 Migrating Files 188
C.5 Migrating Custom Commands and Workflow 189

About This Manual

The *Kablink VibeOnPrem 3.1 Developer Guide* presents ways to extend the functionality of Kablink Vibe. The guide is divided into the following sections:

- Chapter 1, “Web Services Overview,” on page 11
- Chapter 2, “Creating and Packaging Extensions for Deployment,” on page 23
- Chapter 3, “Remote Applications,” on page 29
- Chapter 4, “Creating JavaServer Pages (JSPs),” on page 39
- Appendix A, “Web Services Operations,” on page 69
- Appendix B, “Deprecated Web Services Operations,” on page 153
- Appendix C, “Migrating from Forum to Kablink Teaming,” on page 187

Audience

This guide is intended for programmers who want to write extensions for Vibe.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the most recent version of this manual, visit the Kablink Vibe OnPrem 3.1 Documentation Web site (http://www.novell.com/documentation/kablinkvibe_onprem31).

Additional Documentation

You can find more information in the Kablink Vibe documentation, which is accessible from the Kablink Vibe OnPrem 3.1 Documentation Web site (http://www.novell.com/documentation/kablinkvibe_onprem31).

To access the *Kablink Vibe User Guide* from within Vibe, click the *Help* icon (question mark).

Web Services Overview

1

Much of the information in this section references Teaming, which is the product name for the Vibe product for versions prior to Vibe 3. This information applies to Vibe 3 as well as earlier versions of the product.

Novell offers a set of operations that you can use in client programs to exchange information with a server that is running an installation of Kablink Teaming 2.0 or later.

In addition to the overview information in this chapter, see Appendix A, "Web Services Operations," on page 69, for reference information about the latest operations for the new interface. For reference information about earlier Web Services operations that have been superseded by the current release, see Appendix B, "Deprecated Web Services Operations," on page 153.

- Section 1.1, "Vibe Web Services Terminology," on page 11
- Section 1.2, "Web Services Implementation," on page 12
- Section 1.3, "Authentication," on page 14
- Section 1.4, "Server Endpoints," on page 15
- Section 1.5, "Categories of Operations," on page 15
- Section 1.6, "Client Stubs," on page 16
- Section 1.7, "Managing Data," on page 16
- Section 1.8, "Extending Vibe Web Services," on page 22

1.1 Vibe Web Services Terminology

The following Kablink Vibe definitions that can assist you when using the Vibe Web services:

- **binder:** A place such as a workspace or folder.
- **binder configuration ID:** A number that identifies the template used to create and configure a new workplace or folder. This number represents a set of information that Vibe uses to establish configuration settings, such as the default view, allowable views, allowable workflow, and workflow associations.
- **binder ID:** A unique number that identifies a specific workspace or folder.
- **data item name:** A tag value that maps an HTML form element to a value stored in the Vibe database.
- **definition ID:** A unique 32-character hexadecimal identifier that maps to a definition for a specific type of entry. (You modify and create definitions by using the designers in the administration portlet.) You need to specify this value when creating a new entry in a folder.
- **endpoint:** The URL that you use to connect your client application to the Vibe server.
- **page:** A level in the workspace hierarchy that represents a subset of binders. Most often used to group personal workspaces into sets that are convenient for display in the user interface. Section 1.7.6, "Binder Pages and search_getWorkspaceTreeAsXML," on page 20 provides additional information about this hierarchical level.
- **principal:** A registered user or a group.

- **principal ID:** A unique number that identifies a specific user or group.
- **stub:** A proxy on the client. The stub code performs SOAP calls to the server.

1.2 Web Services Implementation

- Section 1.2.1, “Java Web Services,” on page 12
- Section 1.2.2, “Sample Clients,” on page 13

1.2.1 Java Web Services

Vibe implements Java Web services, which provide a set of operations that client programs can use to exchange information with Vibe. The alphabetized reference section in this documentation provides syntax for these operations (Appendix B, “Deprecated Web Services Operations,” on page 153).

You can view a list of available operations online:

```
http://localhost:8080/ssf/ws
```

The latest operations are listed under the *TeamingServiceV1* header, and the deprecated operations are listed under the *Facade* header.

You can also access the Vibe Web Services Description Language (WSDL) file:

```
http://localhost:8080/ssf/ws/TeamingServiceV1?wsdl
```

In the previous two examples, replace the `localhost` specification with the host and port for your Vibe installation.

NOTE: Vibe does not currently publish its WSDL file with Universal Description, Discovery, and Integration (UDDI) or the Web Services Inspection Language (WSIL). Use the alphabetized reference section in this manual (Appendix A, “Web Services Operations,” on page 69) or the URL-generated WSDL file to understand the Vibe operation interface. For reference information about earlier Web Services operations that have been superseded by the current release, see Appendix B, “Deprecated Web Services Operations,” on page 153.

When you make calls to Vibe Web services, there are two ways that you can implement lower-level Simple Object Access Protocol (SOAP) calls:

- Unzip client-side routines on the system running your application. These routines are Java classes and other files that produce a stub. Your application can use an interface with these stub routines, which make the SOAP calls to and from the server. See Section 1.6, “Client Stubs,” on page 16, for more information about implementing these client-side routines on your application’s system.
- Have your application perform the SOAP calls by using, for example, routines from the Apache Axis toolkit.

Vibe Web services accepts and provides data by using Java objects and methods defined in the Vibe source code. (Visit the Open Source Community page (http://www.kablink.org) for more information about downloading the source code.) Although this section provides tips for locating object and method definitions, you might want to apply a tool such as Javadoc to the sources, so that you have reference pages to assist you in working with the Vibe objects and methods.

The primary method of learning to use Vibe Web services is by reviewing sample clients and their source code, which are provided in the Vibe sources.

1.2.2 Sample Clients

Vibe provides sample clients in its product code base that can assist you in learning how to use its Web services. The sample clients are located within the source code:

`/ssf/samples/wsclient`

The following sample clients are provided. The are listed in the order of how helpful they are in learning how to make Web service calls:

- **teamingservice-client-with-stub.bat (Teaming 2.0+):** Uses client-side routines to implement a Windows batch file for simple operations. This is the recommended method. Using this batch file requires the installation of the client-side routines.
- **teamingservice-client-with-call.bat (Teaming 2.0+):** Uses the Axis Call object when making Web service calls, as a way to implement a Windows batch file for simple operations.
- **facade-client.bat (V1+):** Uses the deprecated Web services interface.
- **wsExport.bat and wsImport.bat (Teaming 2.0+):** Takes data from a portion of the workspace and folder hierarchy and reproduces it on another file system. These tools are not a complete import and export facility, because they do not retain the workflow states, access-control settings, and history of the original objects.

You can find the source files for the sample clients here:

`/ssf/samples/wsclient/src/org/kablink/teaming/samples/wsclient`

The `TeamingServiceClientWithCall.java` file extends the `WSClientBase.java` file, which is also located in the `/ssf/samples/wsclient/src/org/kablink/teaming/samples/wsclient` directory.

Enabling the .bat clients (Windows systems only)

Before executing the sample `.bat` programs on a Windows system, you need to do some work in your build to enable them.

1. Execute the `build` Ant target in `/ssf/samples/wsclient/build.xml` by entering `ant` from the command line.

To use one of the batch files:

1. Use a command line window to `cd` to the `/ssf/samples/wsclient` directory.
2. Type the filename for the batch file you want to execute.

   To see a list of legal commands and arguments for one of the `teamingservice` or `facade` batch files, type only the filename of the batch file, then press the Return key.
3. On the same line, just after the name of the batch file, type a command name and desired arguments.
4. Press the Return key.

If the command executes successfully, Vibe displays the return value in the command line window.

1.3 Authentication

Before determining how to connect your client application to the server, it is important to decide on the authentication method that you want to use. Vibe and its Web services support two types of authentication:

- Section 1.3.1, “HTTP Basic Authentication Access (ssr),” on page 14
- Section 1.3.2, “Web Services Security Access (ssf),” on page 14

1.3.1 HTTP Basic Authentication Access (ssr)

For basic authentication, use calls from your client application to pass a username and password as you establish an HTTP session. Then, perform SOAP calls or calls using the client-side routines. If you want to use basic authentication, you muse use the `/ssr/secure/ws` endpoint when connecting to the server.

HTTP Basic Authentication is the existing transport authentication to authenticate the Web services client. HTTP Basic Authentication uses a username and password to authenticate a service client to a secure endpoint. To use this authentication mechanism, use `/ssr/secure/ws` endpoint. To enable this service on the Vibe side, select the *Enable Basic Authentication (recommended)* check box during product installation.

See Section 1.4, “Server Endpoints,” on page 15, for more information about connecting to the server.

1.3.2 Web Services Security Access (ssf)

For WSS authentication, you need to place the authentication information (username and password) in the SOAP calls. If you want to use this method of authentication, use the `/ssf/ws` endpoint to connect to the server.

Web Services Security (WSS) is a standard protocol from Oasis that provides a means for applying security to Web services. Unlike security mechanisms that rely on the use of transport layer services, WSS provides authentication at the message layer by using a SOAP header. To use this authentication mechanism, use `/ssf/ws` endpoint. The deprecated Web services operation is accessed only through this mechanism. This service is enabled on the Vibe side by selecting the *Enable WSS Authentication (recommended)* check box during product installation.

If you choose to use WSS authentication instead of HTTP basic authentication:

- Use the `teamingservices-client-with-call.bat` client and its sources to see an example of this type of authentication.
- You must use the `/ssf/ws` endpoint (see Section 1.4, “Server Endpoints,” on page 15, for more information).
- You must use password-text methods.

  Password-digest is still supported in Teaming 2.0 and earlier but support is dropped with Teaming 2.1. We strongly recommended that you use only the password-text method.

On the client side of the Web services transaction, the client code uses password-text to provide a username and password to the Web services framework, and the framework passes the password as plain text.

On the server side, the security framework allows Vibe to retrieve the clear-text password from the operation by using an application programming interface (API) call. Vibe applies its internal password encryptor and compares the result with the password stored in the database for the user when the password is retrieved.

Although it is easy to code, this method is not secure, because the password is transmitted in plain text. Systems requiring a higher level of security should connect to Vibe over SSL.

To use this service with the `teamingservice-client-with-call.bat`, edit the script and set the value of the `-Dauthmethod` switch to `wss_text`.

See Section 1.4, “Server Endpoints,” on page 15, for more information about connecting to the server.

1.4 Server Endpoints

An endpoint is the URL that you use to connect your client application to the Vibe server. Depending on the authentication method you want to use and other factors, you must choose one of the following five endpoints to specify in your client application:

- **/ssf/ws/TeamingServiceV1:** Use this endpoint if you want to use WSS authentication with the latest Web services operations. See Section 1.3, “Authentication,” on page 14.
- **/ssf/ws/Facade:** Use this endpoint if you want to use the deprecated Web services operation. This endpoint requires WSS authentication.
- **/ssr/secure/ws/TeamingServiceV1:** Use this endpoint only if you are using HTTP Basic Authentication with the latest Web services operations.
- **/ssr/token/ws/TeamingServiceV1:** Use this endpoint when you are making a Web services call as a remote application.
- **/ssr/ws/TeamingServiceV1:** Use this endpoint when you want to access Vibe as an anonymous user (not specifying any username or password).

1.5 Categories of Operations

To assist you in locating the operation you need to perform, the name of each operation is prefaced with its category name. For example, one category is called `folder`, and one operation within that category is `folder_getEntry`.

The following categories of Web services operations are available:

- **binder:** Operations that are specific to workspaces, common to workspaces and folders, or that are to be applied to the workspace tree beginning at a specific node in the tree.
- **definition:** Operations for obtaining and using definitions. Definitions are created by using the designers within the Vibe UI.
- **folder:** Operations that affect only folders and their contents (entries and comments).
- **ical:** The operation that adds a calendar entry.
- **ldap:** Operations that work with LDAP data.
- **license:** Operations used for license compliance.
- **migration:** Operations that assist migration from the SiteScape Forum product to Vibe. See Appendix C, “Migrating from Forum to Kablink Teaming,” on page 187.

- **profile:** Operations affecting users and groups.
- **search:** Operations that assist in locating information based on criteria other than the defined type.
- **template:** Operations that create workspaces and folders, or that get lists of available templates. (To create a completely configured folder, use `template_addBinder` and not `binder_addBinder`.)
- **zone:** Operations that work with different Vibe starting points within the same installation. Each starting point contains its own unique workspace hierarchy.

1.6 Client Stubs

A stub is a proxy on the client. The stub code performs SOAP calls to the server. Vibe provides pregenerated Java stub classes that are included in the Kablink Vibe Web Services Java client library. To obtain the Kablink Vibe Web Services Java client library, see Section 1.7.1, “Working with Java Objects,” on page 17.

The following example is the `deleteFolderEntry` method defined in the sample Java class `TeamingServiceClientWithStub.java` file. The `TeamingServiceClientWithStub.java` file makes SOAP calls to Vibe through the use of the pregenerated Java stub classes. This method uses the folder_deleteEntry Web services operation to delete an entry from Vibe. This code assumes that your client is running on the same machine that is running the Vibe server (localhost). It uses the Basic Authentication mechanism for authentication.

```
  private static final String TEAMING_SERVICE_ADDRESS_BASIC   = "http://
localhost:8080/ssr/secure/ws/TeamingServiceV1";

  private static final String USERNAME = "admin";
  private static final String PASSWORD = "test";
      .
      .
      .
  public static void deleteFolderEntry(long entryId) throws Exception {
    TeamingServiceSoapServiceLocator locator = new
TeamingServiceSoapServiceLocator();
    locator.setTeamingServiceEndpointAddress(TEAMING_SERVICE_ADDRESS_BASIC);
    TeamingServiceSoapBindingStub stub = (TeamingServiceSoapBindingStub)
locator.getTeamingService();
    WebServiceClientUtil.setUserCredentialBasicAuth(stub, USERNAME, PASSWORD);

    stub.folder_deleteEntry(null, entryId);

    System.out.println("ID of the deleted entry: " + entryId);
  }
```

1.7 Managing Data

Some operations are less intuitive than others for messages. This section provides additional information for those operations and includes the following subsections:

- Section 1.7.1, “Working with Java Objects,” on page 17
- Section 1.7.2, “Adding Folders and the Binder Configuration Identifier,” on page 18
- Section 1.7.3, “Attaching Files,” on page 19

- Section 1.7.4, “Fetching Attachments,” on page 20
- Section 1.7.5, “Adding Calendar Entries,” on page 20
- Section 1.7.6, “Binder Pages and search_getWorkspaceTreeAsXML,” on page 20

1.7.1 Working with Java Objects

The Web services operations often pass and return data within model objects as defined within the Kablink Vibe software. This is beneficial because it cuts down on the amount of code required to prepare, send, receive, and interpret data. For example, parsing XML strings requires more coding. For users who develop Web services client applications in Java, Kablink Vibe provides a client-side library that they can use directly for added convenience. Users who develop Web services client applications in a language other than Java must rely on their own tools for understanding and coding the Kablink Vibe Web interfaces that have been defined and exposed by the corresponding WSDL.

Regardless of the language and tools that are used to develop Web services applications, it is helpful to familiarize yourself with some of the Vibe source code in order to understand the model objects and methods that are used to pass parameters and receive returned data.

To obtain the Kablink Vibe Web Services Java client library, download the Kablink Vibe product distribution tar/zip file from the Kablink Web site (http://kablink.org/teaming), and expand file in a directory. This product distribution tar/zip file contains `teaming-2.*.*-wsclient.zip`. This file contains:

- The Axis-generated Java source and class files for the client side stubs and model classes.

  `kablink-teaming-wsclient.jar`

- Search utility classes that aid in building search queries.

  `kablink-teaming-util-search.jar`

- All third-party libraries needed on the client side to run generated stubs.

The `kablink-teaming-wsclient.jar` file contains the Java source that defines model objects that are passed between the Web services client and the Vibe server as either input arguments to or return values from various Web service operations. These model classes are located in the `org/kablink/teaming/client/ws/model` Java package. A significant number of the model classes build upon the base class `DefinableEntity`. The `TeamingServiceSoapBindingStub.java` class is the main stub class that application programs need to interact with in order to invoke various Web service operations.

To access Java sample programs that use the Kablink Vibe Web Services Java client library, download the Vibe source code from the Open Community Source page (http://www.kablink.org/) and examine the source code and scripts located in the `/ssf/samples/wsclient` directory. For example, the `TeamingServiceClientWithStub.java` class in `/ssf/samples/wsclient/src/org/kablink/teaming/samples/wsclient` demonstrates how to use the supplied stub and model classes to invoke Vibe Web services operations with minimum coding effort.

The `kablink-teaming-wsclient.jar` is also found with in the source tree in the `/ssf/ws-client` directory. To implement a client-side application of your own, all of the necessary libraries must be defined as being in your class path. When the sample program is run in `/ssf/samples/wsclient`, the accompanying `build.xml` Ant build script performs this function for you. It can be viewed as a template.

The names of the Web services operations use categories to organize the operations so they are easier for you to locate and understand. In general, the categories describe an item within Vibe that is the focus of the operation, such as folder, entry, binder, or attachments.

1.7.2 Adding Folders and the Binder Configuration Identifier

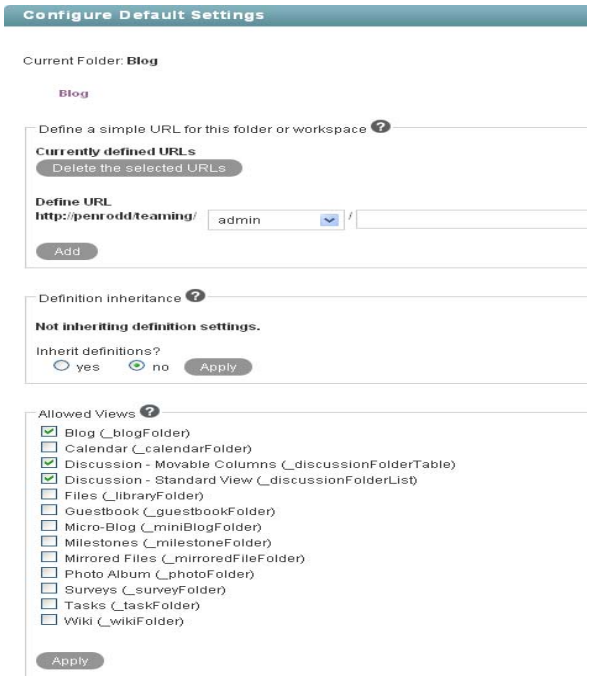
When you add a fully configured folder such as template_addBinder (page 148), you need to specify a binder configuration identifier, which identifies the template used to configure a folder of a particular type. For example, the blog-folder template specifies settings used to configure a new blog folder.

To review the blog-folder template:

1. Log in to Vibe as the Vibe administrator.
2. Click the *Administration* icon  in the upper-right corner of the page.

   The Administration page is displayed.
3. Under Management, click *Workspace and Folder Templates*.
4. In the *Standard Templates* section, click *Blog*.
5. Click *Manage This Target* > *Configure*.

   The Configure Default Settings page is displayed.



The following configuration settings are available in the template:

- Definition inheritance
- Allowed Views
- Default View
- Default Entry Types

- Workflow Associations
- Allowed Workflows

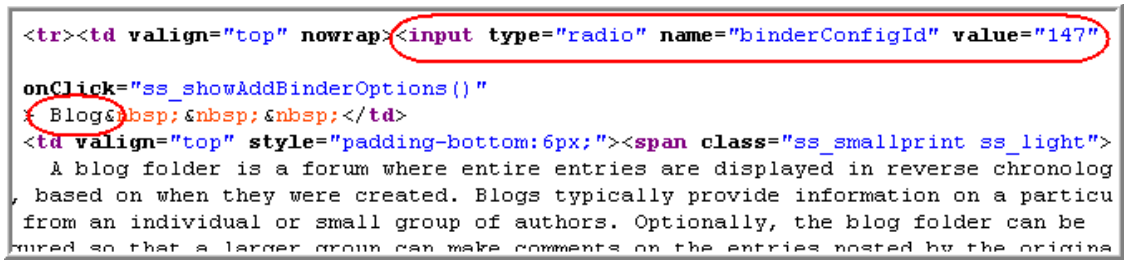
At the time of this writing, Vibe does not provide a message that you can use to retrieve the binder configuration identifier for a particular type of folder. Use the following procedure to obtain the binder configuration identifier for the folder you want to create:

1. View any workspace or folder.
2. Click *Manage* > *Add folder*.
3. While viewing the *Add new folder* page, use your browser to view the HTML source code for the page.
4. Search for the type of folder you want to create (for example, discussion, blog, or calendar).
5. In the `input` HTML tag that creates the radio button for that type of folder, note the `name="binderConfigId"` and `value="nnn"` pair of tag elements.

   The number specified by the `value` element is the binder configuration identifier of the folder you want to create.

The following figure shows an example of the binder configuration information for a blog folder, as found in the HTML source for the *Add new folder* page:

Figure 1-1 *The Binder Configuration Identifier in Source Code*



1.7.3 Attaching Files

In Kablink Vibe, attachments are files that are associated with an entry. An entry can have more than one attached file.

For Web services, an attachment is a file exchanged in conjunction with an operation being passed between the client and server. Vibe recognizes only the first file attachment to an operation being sent to the server and ignores all other attachments.

To attach more than one file to an entry in Vibe, you must use one of the upload operations multiple times. For example, to attach 17 files to an entry in Vibe, you must use `folder_uploadFile` 17 times. Your client source code establishes where in the file system it finds or places files used as attachments to messages.

The `folder_uploadFile` operation requires that you pass a data item name. This identifier maps to the value specified in the `name` attribute of the `input` HTML tag used to upload the file; this value is also used in a `hidden` HTML tag that communicates values between the HTML form and the Vibe database.

To upload a file into the standard form element used to contain attachments, specify `ss_attachFile` as the data item name. If you are uploading files into a custom form element, create an instance of that custom entry, use an operation to get the name of the hidden field, then use the name when attaching files to the entry you actually want to affect.

1.7.4 Fetching Attachments

When you use `folder_getEntry` to obtain information about an entry, you use a Boolean parameter to indicate if you want the entry's attachments. If you specify that you do want the attachments, your client establishes where on its system it places the attached files.

1.7.5 Adding Calendar Entries

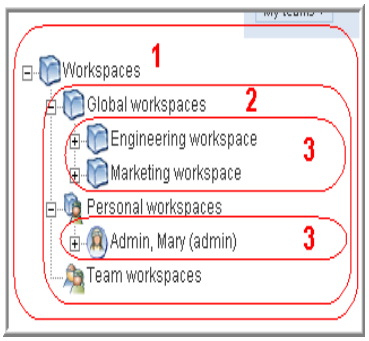
When you pass the `ical_uploadCalendarEntriesWithXML` operation to the server, the Web services framework uses an XML formatted string of iCal data passed as the second parameter to the operation (`<doc><entry>iCal data</entry></doc>`).

1.7.6 Binder Pages and search_getWorkspaceTreeAsXML

When you use `search_getWorkspaceTreeAsXML` to obtain information about the hierarchical workspace tree, Kablink Vibe returns XML formatted information about nodes in the tree, within the levels of the hierarchy you specify. Each node in the tree is a binder, which is typically a place (a workspace or folder). Sometimes, the XML element returned for a node is called a page.

The following graphic shows the workspace tree, which is expanded to show five levels of the workspace hierarchy:

Figure 1-2 *Workspace Hierarchy Levels as Seen in the UI*

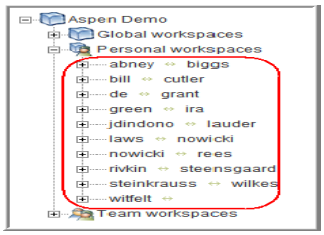


In the graphic, each of the workspaces and folders are nodes in the workspace tree. The Workspaces workspace is the only binder at level 1. Level 2 binders include Global workspaces, Personal workspaces, and Team workspaces. The only binder shown at level 3 is the Corporate web site binder. Level 4 binders include folders and the December 2008 redesign workspace. The Calendar binder is located at level 5. If a binder has a plus sign next to it (for example, both the Global workspaces and Personal workspaces binders are preceded by plus signs), it means that there are hierarchy levels of binders that are not displayed in the UI.

If you use `search_getWorkspaceTreeAsXML` to get one level of the tree starting at the Workspaces node, Vibe returns information about Global workspaces, Personal workspaces, and Team workspaces.

As mentioned, some nodes in the tree are pages:

Figure 1-3 *Pages as They Appear in the UI*



The `/ssf/web/docroot/WEB-INF/classes/config/ssf.properties` file contains a property called `wsTree.maxBucketSize`, which, by default, is set to `25`. This means that the maximum number of sub-workspaces allowed is 25. If a folder or workspace has more subplaces, Vibe creates virtual buckets called pages. Each line in Figure 1-3 corresponds to a page. The Personal workspaces workspace has two pages.

When you use `search_getWorkspaceTreeAsXML` to retrieve information about nodes in the workspace tree, it can return more than one hierarchical level as you specify, unless it encounters a page. To expand the tree beyond a page, you must call `search_getWorkspaceTreeAsXML` again, pass the binder identifier of the page, and pass the number of levels beyond the page you want to retrieve.

Consider the following:

Figure 1-4 *A Page Containing Sub-Workspaces*

The *wong//zeeman* page contains workspaces. The workspaces listed (*Wong, Charles (cwong)*, and *Zeeman, Skip szeeman)*) are one level beyond the page.

When you receive page information as a node in the workspace tree, you receive `page` and `tuple` attributes. For example, `page="2"` and `pageTuple="charles_wong (cwong)//skip_zeeman (szeeman)"`. To obtain information about the contents of this page, you need to specify the identifier of the page's parent, the number of hierarchy levels you want expanded, and a concatenation of the page number and tuple values, as shown in this example:

```
search_getWorkspaceTreeAsXML 24 3 "2//charles_wong//skip_zeeman"
```

This code begins at binder number 24, accesses page number 2, and returns two hierarchical levels of data for all users between Charles Wong and Skip Zeeman.

Given the structure of the Vibe pages and how Web services returns tree data, it is easiest to retrieve page data in this way. However, if you choose, you can actually retrieve paged tree data regardless of page number. To do this, specify any page number (Vibe actually ignores it), and specify a tuple in the correct order in which it appears in the tree, even if the set of users crosses pages. Vibe returns

hierarchical information for all users in between the tuple values. However, if the number of returned nodes exceeds the value specified in the `wsTree.maxBucketSize` property (by default, 25 users), Vibe pages the data.

Finally, if you want to see all tree information without any page specifications, specify `-1` as the value of the hierarchy levels you want returned.

1.8 Extending Vibe Web Services

Because Kablink Vibe is open source software, you have the source code that implements our Web services, and you can extend it. However, we invite you to operate within the spirit of an open source community by participating in the Kablink Vibe online community (http://www.kablink.org), sharing your code with others, and working with the Novell engineers to incorporate your Web services extensions into the base product. In this way, you make the product and community stronger, and you avoid doing work that might need to be redone in future versions of Kablink Vibe because of engineering changes.

Of course, whether you participate in the community or upgrade to future versions of the software is up to you. Regardless of your decision, Kablink Vibe includes an example that provides a structure that enables users of all versions of our software to extend our Web services in the most optimal way, minimizing work that you might need to do to maintain the extensions for every upgrade.

Kablink Vibe includes an extended Web services example, which adds the `folder_getFolderTitle` operation to the base Vibe web services, and also adds the `getFolderTitle` command to the `teamingservice-client-with-call.bat` sample client. The source code for the extension is located in this directory and in its subdirectories:

```
/ssf/samples/extendedws
```

This directory contains the `readme.txt` file, which provides simple directions for establishing the extension.

Creating and Packaging Extensions for Deployment

2

Much of the information in this section references Teaming, which is the product name for the Vibe product for versions prior to Vibe 3. This information applies to Vibe 3 as well as earlier versions of the product.

Extensions in Kablink Vibe enable you to bundle definition and template XML, custom workflow calls that implement the workflow action and condition interfaces, custom JSP files, and other Web-visible resources that are needed by the application browser interfaces.

Extensions can help you customize your Kablink Vibe site, adding increased functionality that can help you solve specific business problems.

This section describes how to create an extension and then package it for deployment. For information on how to deploy an extension after it has been created and packaged, see "Adding Software Extensions to Your Vibe Site" in the *Kablink Vibe OnPrem 3.1 Administration Guide*.

- Section 2.1, "Understanding the Differences between Extensions and Remote Applications," on page 23
- Section 2.2, "Creating an Extension," on page 24
- Section 2.3, "Packaging an Extension," on page 24
- Section 2.4, "Deploying an Extension," on page 26
- Section 2.5, "Updating an Extension," on page 26
- Section 2.6, "Locating an Extension in the Vibe Directory Structure," on page 27
- Section 2.7, "Retaining an Extension When Updating Your Vibe Software," on page 27

2.1 Understanding the Differences between Extensions and Remote Applications

Extensions and remote applications can be used to accomplish many of the same functions; however, the way in which they are created and how they are implemented can differ dramatically. Before you create a Vibe extension or remote application, consider how you want to create it (such as what coding language you want to use), as well as the environment in which you want your extension or remote application to run (such as in an external Web application or Web server).

Table 2-1 depicts important technical differences between extensions and remote applications.

Table 2-1 *Technical Differences between Extensions and Remote Applications*

	Extension	Remote Application
Web Container	Tomcat only	Any container (for example, Tomcat, Apache, IIS)

	Extension	Remote Application
Coding Language	Java and JSP only	Any language (for example, PHP, Ruby, .NET)
Web Application	Must run inside the Vibe Web application	Runs outside of the Vibe Web application
Server	Runs on the same server as Vibe	Can either run on the same server as Vibe (but on a separate Web application), or on an external server

2.2 Creating an Extension

Vibe extensions are made up of various files that are commonly used when designing a Web page.

To create an extension for your Kablink Vibe site:

1. Create all of the necessary files.

   Vibe extensions are made up of various files that are commonly used when designing a Web page, such as `JSP`, `HTML`, `XHTML`, `CSS`, `JS`, `GIF`, `JPG`, `PNG`, `CLASS`, `JAR`, `XML`, `PROPERTIES`, and `TXT`.

2. After you have created all the necessary files, you must properly package the files, as discussed in Section 2.3, "Packaging an Extension," on page 24.

2.3 Packaging an Extension

After you create an extension as described in Section 2.2, "Creating an Extension," on page 24, you must package the extension before it can be deployed on the Vibe site.

1. Create a `ZIP` file that contains all of the files needed for your extension.

   The `ZIP` file should have a relative directory structure that mirrors the layout of the `teaming/`*`tomcat`*`/webapps/ssf` directory found in your Vibe installation.

   For examples of how the extension directory structure should look, see Section 2.3.1, "Examples of the Archive Format," on page 25.

   Table 2-2 lists key files and their appropriate locations in this directory structure.

Table 2-2 *File Locations for Vibe Extensions*

File Type	Location
Web-visible resources, such as graphics, static html pages, `js` files, and `css` files.	*`tomcat`*`/webapps/ssf` You can create additional folders inside the `ssf` directory. For example, you might want to create a `js` directory where you can place all your javascript files.
Templates	`WEB-INF/classes/config/templates`
Definitions	`WEB-INF/classes/config/definitions`
`jsp`	`WEB-INF/jsp`

File Type	Location
jar	WEB-INF/lib

2.3.1 Examples of the Archive Format

Your ZIP file should be structured in the Archive format. This section provides two examples of the archive format. Archive: VideoEntry.zip includes externally referenced files in the swf/, img/, and js/ directories. Archive: twitter.zip references only those files that are located within the WEB-INF directory.

- "Archive: VideoEntry.zip" on page 25
- "Archive: twitter.zip" on page 25

Archive: VideoEntry.zip

```
install.xml
js/
js/flashembed.min.js
swf/
swf/FlowPlayerClassic.swf
swf/FlowPlayerLight.swf
swf/FlowPlayerLP.swf
swf/FlowPlayerDark.swf
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/config/
WEB-INF/classes/config/definitions/
WEB-INF/classes/config/definitions/VideoEntry.xml
WEB-INF/classes/config/definitions/VideoFolder.xml
WEB-INF/classes/config/templates/
WEB-INF/classes/config/templates/Video Folder Template.xml
WEB-INF/jsp/
WEB-INF/jsp/view.jsp
img/
img/no-flash.png
img/no-flash.svg
```

Archive: twitter.zip

```
install.xml
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/config/
WEB-INF/classes/config/definitions/
WEB-INF/classes/config/definitions/_user.xml
WEB-INF/classes/config/definitions/pubToTwitter.xml
WEB-INF/src/
WEB-INF/src/TwitterWorkflowAction.java
WEB-INF/lib/
WEB-INF/lib/TwitterExtension.jar
WEB-INF/lib/commons-httpclient-3.1.jar
WEB-INF/lib/twitter4j-2.0.8.jar
WEB-INF/jsp/
WEB-INF/jsp/password.jsp
```

2.3.2 Extension Metadata

The `install.xml` file should contain the following information:

```
<?xml version="1.0" encoding="utf-8"?>
<extension version="1.0" >
<title>Twitter Extension</title>
<author>Author's Name</author>
<creationDate>August 27 2009</creationDate>
<license>http://www.gnu.org/licenses/gpl-2.0.html GNU/GPL</license>
<authorEmail>nbjensen@novell.com</authorEmail>
<authorUrl>www.mysite.com</authorUrl>
<description>This Plugin is a sample.</description>
<usage>Instructions on how to use this extension.</usage>
</extension>
```

2.4 Deploying an Extension

After you have created and packaged an extension, it is ready to deploy into your Vibe site.

NOTE: If your extension requires additional configuration after it is deployed into the Vibe site, ensure that you provide installation instructions on the page where the extension is downloaded.

You can deploy a Vibe extension in the following two ways:

- Section 2.4.1, “Deploying an Extension from the Vibe Interface,” on page 26
- Section 2.4.2, “Deploying an Extension from the Vibe Server,” on page 26

2.4.1 Deploying an Extension from the Vibe Interface

For information on how to deploy an extension by using the Vibe interface, see “Adding Software Extensions to Your Vibe Site” in the *Kablink Vibe OnPrem 3.1 Administration Guide*.

2.4.2 Deploying an Extension from the Vibe Server

1. Copy the `ZIP` file that contains all the necessary files for your extension to the following location on the Vibe server: `/var/opt/teamingdata/extensions/kablink/pickup`

   Vibe periodically checks this directory and deploys the file on the Vibe server.

2.5 Updating an Extension

1. Navigate to and unzip the `ZIP` file that you created that contains all of the files for your extension.
2. Modify the `install.xml` file and any other files that you want to update.
3. Create a `ZIP` file with the updated files.

   The name of this file must be the same name as your original extension.
4. Deploy your extension, as described in “Adding Software Extensions to Your Vibe Site” in the *Kablink Vibe OnPrem 3.1 Administration Guide*.

Vibe automatically updates your extension files with the latest files from your newly deployed extension.

2.6 Locating an Extension in the Vibe Directory Structure

You might need to locate an extension in the Vibe directory structure in the following scenarios:

- When you are developing and building the file structure for the extension
- When you are troubleshooting problems associated with the extension after it has been deployed

You can locate the WEB-INF contents and Web-visible contents of the extension at the following locations in the Vibe directory structure:

WEB-INF Contents: `ssf/WEB-INF/ext/`*`zoneKey`*`/`*`extensionName`*

Web-Visible Contents: `ssf/ext/`*`zoneKey`*`/`*`extensionName`*

The name of the extension is the name of the `ZIP` file that you created in Section 2.3, “Packaging an Extension,” on page 24.

2.7 Retaining an Extension When Updating Your Vibe Software

When extensions are located in the `/var/opt/teamingdata/extensions` directory, they are not lost during a Vibe software update.

However, if you install a new extension that has the same name as an existing extension, the existing extension is overwritten.

Remote Applications

3

Much of the information in this section references Teaming, which is the product name for the Vibe product for versions prior to Vibe 3. This information applies to Vibe 3 as well as earlier versions of the product.

You can set up your Kablink Vibe installation so that remote applications—which often run on other server machines—provide HTML for segments of Vibe pages. Using this customization method gives you several advantages:

- Provides more control over the format and content than using Vibe designers or JSP files.
- Allows customization designers to work primarily within a familiar development environment (for example, using PHP or Perl) instead of requiring them to do most of their work within the Vibe environment.
- Protects the customization from the effects of any future updates to the Vibe source code.
- Eliminates the risk of implementing a customization that destabilizing the base product.

This topic includes the following sections:

- Section 3.1, “Understanding the Differences between Extensions and Remote Applications,” on page 29
- Section 3.2, “Remote Application Overview,” on page 29
- Section 3.3, “Creating a Remote Application,” on page 34
- Section 3.4, “Related Sections,” on page 37

NOTE: This chapter presents a “hello world” example application that you can implement quickly on your Kablink Vibe server, using the Eclipse build environment and Tomcat. The registration and application sections assume use of this example application. For more information about the example application, see Section 3.3, “Creating a Remote Application,” on page 34.

3.1 Understanding the Differences between Extensions and Remote Applications

Extensions and remote applications can be used to accomplish many of the same functions; however, the way in which they are created and how they are implemented can differ dramatically.

For information about the technical differences between extensions and remote applications, see Section 2.1, “Understanding the Differences between Extensions and Remote Applications,” on page 23.

3.2 Remote Application Overview

This section provides a conceptual overview about the interoperability between Kablink Vibe and remote applications. If you prefer to learn by doing, you might want to skip this section and begin with Section 3.3, “Creating a Remote Application,” on page 34.

Use the following very high-level steps as an overview for setting up a customization based on a remote application:

1. A site administrator registers a remote application, specifying a name and a URL for the application (for more information, see Section 3.4.1, “Registering a Remote Application,” on page 37).
2. After using configuration or designer tools to include HTML from a remote application, a page in Vibe performs an HTTP POST operation to the application’s URL and provides user-context information.
3. As an option, the remote application can use Web services to obtain Vibe data needed to generate its section of the page.
4. The remote application responds, providing its HTML to the Vibe code that is drawing the page.

   Because Vibe has already begun to draw the page, providing structuring HTML tags (such as `html`, `title`, `head`, and `body`), the remote application should not specify those tags.

In Vibe, a remote application is treated as a principal; examples of other principals are users and groups. Vibe treats the application like a user, and users are subject to access control.

When Vibe makes a request of the remote application, it provides the user identifier for the user currently viewing the page, and it provides a security token. Depending upon the type of Vibe page that incorporates HTML from the remote application, the security token can be session-based (valid until the user signs out) or request-based (invalid immediately after the remote application responds to the request). In most cases, Vibe uses a combination of the registration information for the remote application, rights assigned to the user currently viewing the page, and rights granted to the remote application itself.

There are several ways to apply HTML generated from remote applications:

- As an accessory (session-based token)
- As any portion of a custom form or view (session-based token)
- As a consequence of a workflow state change (request-based token)
- As a call from a tag within a custom JSP file (session-based token)

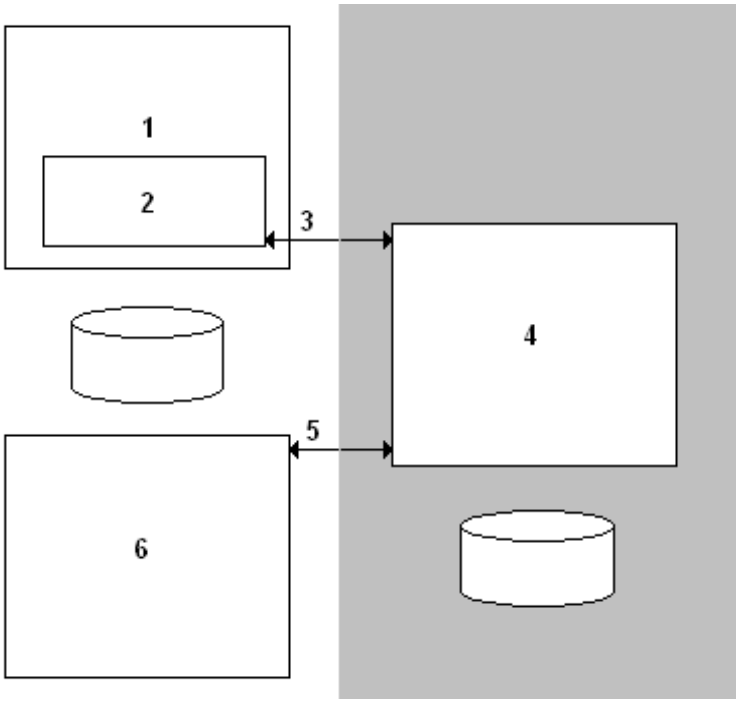
For more information, see the following subsections:

- Section 3.2.1, “Processing Flow for a Remote Accessory,” on page 30
- Section 3.2.2, “Processing Flow for a Remote Form,” on page 32
- Section 3.2.3, “Setting Access Control for Remote Applications,” on page 33
- Section 3.2.4, “Reviewing Supporting Source Code,” on page 34

3.2.1 Processing Flow for a Remote Accessory

This section describes the flow of information when someone configures an accessory to use HTML from a remote application to provide its content. Consider the numbers in this graphic and the descriptions that follow it:

Figure 3-1 *Conceptual Diagram for an Accessory Calling a Remote Application*



1. The enclosing rectangle in the graphic represents a workspace or folder page.

   Vibe draws the page until it encounters the accessory.

2. The enclosed rectangle represents an accessory that is configured to use HTML generated from a remote application.
3. Vibe calls the remote application by using the HTTP POST method, and it provides the user ID of the person viewing the page and a security token.

   The gray portion of the diagram indicates that control has been passed to the remote application, which might be located on a server machine separate from the one running Kablink Vibe.

4. The rectangle in the gray area represents the remote application.

   The remote application can access databases external to Vibe as part of its process for constructing the HTML needed to generate the accessory. The remote application can run in any environment that can generate an HTML response (for example, using PHP or Pearl scripts in its application environment).

5. As an option, the remote application can use the Vibe user ID and security token to log into the Vibe Web services and make calls, which can gather Vibe data in preparation for generating the HTML for the accessory.
6. This rectangle represents Vibe Web services.

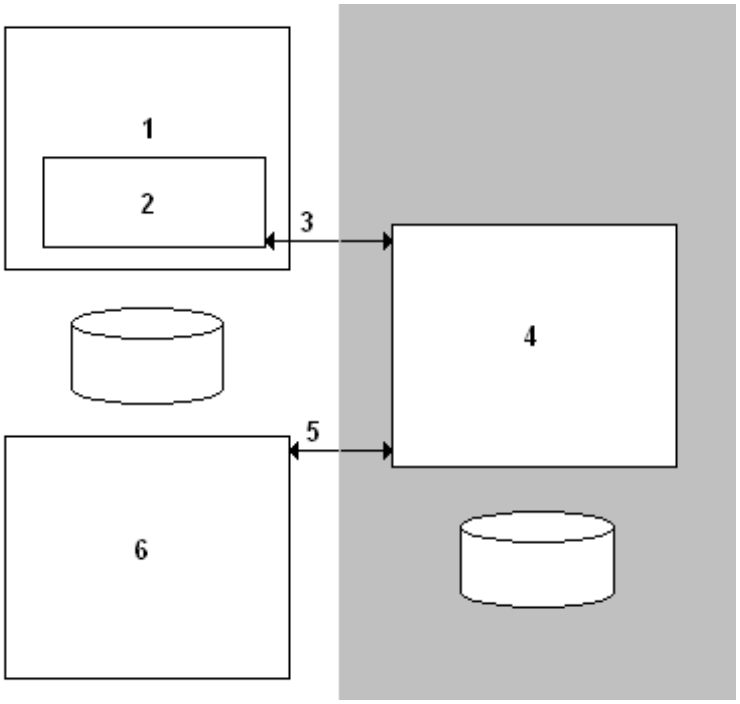
   In the graphic, both Web services and the Vibe user interface have access to information in the Vibe database.

After the remote application responds by returning HTML for the accessory, Vibe adds the HTML to the output stream and finishes drawing the page. Then, Vibe revokes the security token so that it can no longer be used.

3.2.2 Processing Flow for a Remote Form

This section describes the flow of information when someone uses the Vibe designers to specify that a remote application provides the HTML for the form used to create an entry. Consider the numbers in this graphic and the descriptions that follow it:

Figure 3-2 *Conceptual Model for an Add-Entry Form Generated Remotely*



1. The enclosing rectangle represents the add-entry page.
2. The enclosed rectangle represents the form used to add an entry.
3. Vibe does an HTTP POST to the remote application, sending the folder ID of the enclosing folder, the user ID of the person currently viewing the add-entry page, and a session-level security token.
4. The remote application builds the HTML needed to construct the form to be displayed on the Vibe page. The URL specified to the `action` element in the `form` tag points to the remote application.
5. The remote application can use the user ID and security token to log into Vibe Web services, which can gather data used to construct the form.
6. This rectangle represents the Vibe Web services.

After the remote application responds by returning HTML for the form, Vibe adds the HTML to the output stream and finishes drawing the page. The user completes and then submits the form. Information on the form goes directly to the remote application, which then uses the folder ID, user ID, and security token to make Web services calls, which, in turn, create a new entry in the folder based on the information provided in the form.

When the user logs out, Vibe revokes the security token.

3.2.3 Setting Access Control for Remote Applications

- "Understanding Access Control Settings for Remote Applications" on page 33
- "Setting Access Controls on a Remote Application" on page 33

Understanding Access Control Settings for Remote Applications

As mentioned, Vibe treats a remote application as a Kablink Vibe user, which allows for the application of access control. However, for non-workflow uses of remote applications, Vibe considers these factors when determining access control:

- Rights granted to the user currently viewing the page that contains HTML from the remote application.
- Rights granted to the remote application by the owner of the containing binder (workspace or folder).
- The Trusted designation, which a Site Administrator can grant to a remote application upon registration.

Vibe considers the role of the user viewing the page and the role assigned to the remote application, and Vibe grants the less powerful role to the application. For example, if the user viewing the page has the role of Visitor, and if the workspace or folder owner places the remote application in the role of Participant, the application has the rights associated with a Visitor. As another example, if the user is a Site Administrator, and if the application is a Participant, the application has the rights associated with a Participant.

When the site administer selects the *Trusted* check box while registering a remote application, then Vibe disregards any access control applied to the application and uses only the access control granted to the user viewing the page.

Finally, if Vibe calls a remote application as the result of a workflow state change, access control is determined by settings in the workflow definition, which can result in the application having the rights of either the entry owner or the folder owner.

Setting Access Controls on a Remote Application

Unless you are certain that an application is trusted (for example, an application that you maintain and run on the same server machine as Kablink Vibe), you should strongly consider placing access controls on the remote application itself. For example, if the site administrator uses the access-control tools to place the remote application into the Participant role in the top Workspace, all workspaces and folders that inherit access control automatically apply the same restrictions on the remote application. Then, if a site administrator views a page calling the remote application, the application is restricted to the rights granted to a Participant instead of those granted to a Site Administrator.

If you do not make any attempt to restrict the rights of a remote application, the remote application has the same rights as the user viewing the page that calls the application. For example, if a site administrator views the page, the remote application can use Site Administrator rights to do anything in Vibe using Web services calls.

To set access controls on a remote application:

1. Log in to the Vibe site as the Vibe administrator.
2. Click the *Administration* icon  in the upper-right corner of the page.

The Administration page is displayed.

3. Under System, click *Access Control for Zone Administration Functions*.

   The Configure Access Control page is displayed.

4. In the provided table, click *Add an Application.*
5. In the *Add an Application* field, start typing the name of the remote application for which you want to set access controls. When the remote application appears in the drop-down list, select it.
6. Click the *Add a role* link at the top of the table.
7. From the drop-down list, select the role that you want to associate to the remote application.

   Vibe adds a new column for the new role.

8. In the row of the remote application, select the check box that is located in the role column that you want to assign to the remote application.
9. Click *Save Changes*.

3.2.4 Reviewing Supporting Source Code

To learn more details about the interaction between Kablink Vibe and a remote application, you can look at example source code. To download the Kablink Vibe code base, visit the Open Source Community page (http://www.kablink.org/).

After downloading the Kablink Vibe sources, review this code:

- **SOAP wrappers for objects:** These wrappers are located within the Kablink Vibe source code in `/ssf/samples/wsclient/src/org/kablink/teaming/samples/wsclient/TeamingServiceClientWithStub.java`.

  For more information about Web services and SOAP wrappers, see Chapter 1, “Web Services Overview,” on page 11.

- **Passed information:** To see details about the information passed between Kablink Vibe and a remote application, see `/ssf/main/src/org/kablink/teaming/remoteapplication/impl/RemoteApplicationManagerImpl.java`.

3.3 Creating a Remote Application

This section demonstrates how to create a remote application using the hello-world application as an example.

The hello-world program is common code written by people just beginning to learn a programming language or paradigm. Usually, upon execution, a hello-world application writes a greeting to standard output.

This section presents a hello-world application that ships in the Kablink Vibe source code, which you can quickly deploy and implement in your Vibe installation. When it is executed as a remote application from a Vibe page, this application uses a Web services operation to greet the Vibe user by name.

To download the Vibe code base, visit the Open Source Community page (http://www.kablink.org/). After installing the code base, you can locate this example code here:

`/ssf/samples/remoteapp`

There are four important files in the `/remoteapp` hierarchy:

- **The class file:** This file contains the source code for the remote application.

  `/remoteapp/src/org/kablink/teaming/samples/remoteapp/web/HelloWorldServlet.java`

- **The servlet-definition file:** The servlet-definition file establishes the hello-world application as a Tomcat servlet.

  `/remoteapp/war/WEB-INF/web.xml`

- **The JSP file:** The JSP file is the content of the response for the application.

  `/remoteapp/war/WEB-INF/jsp/hello_world/view.jsp`

- **The build file:** The build file deploys the application in Tomcat, making it available for use as a remote application in Vibe.

  `/remoteapp/build.xml`

The subsections that follow provide more detailed explanation about three of these files:

- Section 3.3.1, “Reviewing the Class File,” on page 35
- Section 3.3.2, “Reviewing the Servlet-Definition File,” on page 36
- Section 3.3.3, “Reviewing the JSP File,” on page 36

3.3.1 Reviewing the Class File

To review the Java source code used to implement this application, locate the following file:

`/remoteapp/src/org/kablink/teaming/samples/remoteapp/web/HelloWorldServlet.java`

Although this is not a complete description of all of the code contained in the example, the next few paragraphs explain some of the key parts of the `HelloWorldServlet` class defined in the `HelloWorldServlet.java` file.

The following line in the code creates the bean to contain the user’s first and last name (`ss_userTitle`), which is to be used by a JSP:

```
private static final String PARAMETER_NAME_USER_TITLE = "ss_userTitle";
```

Toward the bottom of the file, the class defines the following method, which makes a Web services call, obtains the `user` object, and applies the `getTitle` method to the object, placing the user’s first and last name in a string:

```
private String getUserTitle(TeamingServiceV1SoapBindingStub stub, String
accessToken, Long userId)
      throws ServiceException, DocumentException, RemoteException {
    User user = stub.profile_getUser(accessToken, userId, false);
    return user.getTitle();
}
```

The Web services call was set up in the `doPost` method, which then calls the `getUserTitle` method shown in the last code example. Consider this code from the `doPost` method:

```
  private static final String TEAMING_SERVICE_ADDRESS = "http://
localhost:8080/ssr/token/ws/TeamingServiceV1";
      .
      .
      .
// Get ready for web services calls to the Teaming.
TeamingServiceV1SoapServiceLocator locator = new
TeamingServiceV1SoapServiceLocator();
      locator.setTeamingServiceV1EndpointAddress(TEAMING_SERVICE_ADDRESS);
TeamingServiceV1SoapBindingStub stub = (TeamingServiceV1SoapBindingStub)
locator.getTeamingServiceV1();

// Get the title of the user by making a web services call.
String userTitle = getUserTitle(stub, accessToken, Long.valueOf(userId));
```

When using Web services in the context of a remote application, you must use the `/ssr/token/ws/TeamingServiceV1` endpoint. See Section 1.4, “Server Endpoints,” on page 15, for more information about specifying server endpoints for Web service calls.

Finally, the `doPost` code specifies the location of a JSP, which is used to generate the response:

```
String jsp = "/WEB-INF/jsp/hello_world/view.jsp";
RequestDispatcher rd = req.getRequestDispatcher(jsp);
```

3.3.2 Reviewing the Servlet-Definition File

To review the XML used to define the hello-world servlet for Tomcat, locate this file:

```
/remoteapp/war/WEB-INF/web.xml
```

The XML file contains these lines:

```
  <servlet>
    <servlet-name>helloWorld</servlet-name>
    <servlet-
class>org.kablink.teaming.samples.remoteapp.web.HelloWorldServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>helloWorld</servlet-name>
    <url-pattern>/helloWorld/*</url-pattern>
  </servlet-mapping>
```

The `servlet` tag defines the class code to be executed when someone specifies `/helloWorld` in the URL. The servlet-mapping tag establishes `/helloWorld` portion of the URL. (See Section 3.4.1, “Registering a Remote Application,” on page 37, for information about how this definition maps to the URL you specify when registering a remote application with Vibe.)

3.3.3 Reviewing the JSP File

To review the JSP file used to generate the output for the application, locate the following file:

```
/remoteapp/war/WEB-INF/jsp/hello_world/view.jsp
```

The file has the following content:

```
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<c:set var="title" value="${ss_userTitle}"/>
<c:if test="${empty title}"><c:set var="title" value="world"/></c:if>

<strong>Hello ${title}!</strong><br>
```

The JSP tests to see if the `ss_userTitle` bean is empty, and, if it is, substitutes the string `Hello world!` for `Hello` *`userTitle`*`!`

Because the remote application is designed to provide a portion of an HTML page, the JSP file does not include HTML tags that structure the page, such as the `html`, `title`, `head`, and `body` tags. Vibe structures the page, and remote applications provide HTML for a segment within that page.

3.4 Related Sections

The following sections are for Kablink Vibe administrators and advanced Vibe users:

- Section 3.4.1, “Registering a Remote Application,” on page 37
- Section 3.4.2, “Configuring an Accessory to Show a Remote Application,” on page 37
- Section 3.4.3, “Controlling the Access of Remote Applications,” on page 37

3.4.1 Registering a Remote Application

After you create a remote application, the system administrator needs to register the application with the Kablink Vibe system before it can be applied to an accessory, a custom entry, a workflow state, or a tag.

For information on how to register a remote application, see “Using Remote Applications on Your Vibe Site” in the *Kablink Vibe OnPrem 3.1 Administration Guide*.

3.4.2 Configuring an Accessory to Show a Remote Application

Kablink Vibe enables you to view remote applications as accessories.

For information on how to create an accessory that displays a remote application, see “Setting Up a Remote Application As an Accessory” in the *Kablink Vibe OnPrem 3.1 Advanced User Guide*.

3.4.3 Controlling the Access of Remote Applications

Kablink Vibe enables you to set access controls on your remote applications.

For information on why it is important to set access controls on remote applications, see Section 3.2.3, “Setting Access Control for Remote Applications,” on page 33.

For information describing how users can set access controls on a remote application, see “Managing Access Controls for Remote Applications” in the *Kablink Vibe OnPrem 3.1 Advanced User Guide*.

Creating JavaServer Pages (JSPs) 4

Much of the information in this section references Teaming, which is the product name for the Vibe product for versions prior to Vibe 3. This information applies to Vibe 3 as well as earlier versions of the product.

- Section 4.1, “Overview of JSP Support,” on page 40
- Section 4.2, “Examples of Custom Entries,” on page 46
- Section 4.3, “Examples of Complex, HTML Data Types,” on page 65

JavaServer Pages (http://en.wikipedia.org/wiki/JavaServer_Pages)

JavaServer Pages technology (http://java.sun.com/products/jsp/)

NOTE: The method of specifying separate JSP files for the form, view, and mail was the primary method of applying JSP applications for versions of Kablink Vibe prior to 2.0, and this method is still supported in Teaming 2.0 and later. However, Teaming 2.0 and later added support for specifying a single JSP file for the form and then inheriting the JSP in the view. Also, Teaming 2.0 and later supports replacing standard items (such as the title or description) with JSP files.

This topic describes the application of JavaServer Page (JSP) customizations in Kablink Vibe.

Using form and view designers, you can add standard HTML or Vibe elements (for example, a text box or form elements to upload attachments) to standard entries. As another option, you can create new types of entries (for example, a paid-time-off-request entry, a resume-processing entry, a document-review entry, and so on). When you limit a folder to a custom task, then you created a dedicated application.

Using Vibe as an application-development platform is powerful. However, given the tools described in this guide so far, you may have noticed some limitations. For example, when you use the designers to create custom entries, you are allowed some control over the position of the custom elements on the page, but many formatting decisions are left to the Vibe software. If your application requires a level of formatting control that is difficult or impossible to achieve using the designers, you can use JSPs to enhance your customization.

This topic includes these sections:

- Section 4.1, “Overview of JSP Support,” on page 40
- Section 4.2, “Examples of Custom Entries,” on page 46
- Section 4.3, “Examples of Complex, HTML Data Types,” on page 65

NOTE: Although it includes examples of JSP tagging, it is beyond the scope of this topic to teach general tagging syntax and use cases for JavaServer Pages.

4.1 Overview of JSP Support

This section explains the relationship between the Kablink Vibe UI and the content of the JSP files, and provides other information to assist in your use of JSP customizations. If you prefer to learn by doing, you may want to skip this section and review the examples (see Section 4.2, “Examples of Custom Entries,” on page 46).

This section includes these subsections:

- Section 4.1.1, “Directory Structure,” on page 40
- Section 4.1.2, “Applicable Pages,” on page 41
- Section 4.1.3, “JSPs and the Vibe Designers,” on page 41
- Section 4.1.4, “Indexing Issues,” on page 44
- Section 4.1.5, “JSPs and Vibe Data Access,” on page 44
- Section 4.1.6, “Text Display in the HTML Editor,” on page 45
- Section 4.1.7, “Standard Styles,” on page 46

4.1.1 Directory Structure

In the Kablink Vibe implementation of JSP-based customizations, you specify the JSP files using the designers, which are located in the administration portlet.

The designer interface expects to find the JSP files relative to this location within the server directory structure:

```
/WEB-INF/jsp/custom_jsps
```

Vibe ships sample JSP customizations in this directory:

```
/WEB-INF/jsp/custom_jsps/samples
```

By default, Vibe includes in this directory three files that you can use to practice applying a small JSP customization to a single element within a page:

```
custom_jsp.html
custom_jsp_form.html
custom_jsp_view.html
custom_jsp_mail.html
```

A section that follows describes how to apply this sample customization and what it looks like in the Vibe user interface (UI). For more information, see Section 4.2.1, “A JSP That Defines Only One Data Element,” on page 46.

Because the `/custom_jsps` directory contains JSP files for all customizations in the installation, Novell strongly recommends that you create subdirectories for each customization. For more information, see “Enabling Custom JSPs to Be Used on Your Vibe Site” in the *Kablink Vibe OnPrem 3.1 Administration Guide*.

For example, a section that follows shows sample JSPs that produce almost the entire bodies of the form and view pages for an entry, as it implements a W-4-form application. (A W-4 form is a government form in the U.S.A. that is used to withhold federal taxes from an employee’s paycheck.) For example, the sample W-4 application places its JSP files in this subdirectory:

```
.../custom_jsps/samples/w4
```

See Section 4.2.2, “A JSP-Defined Entry (W-4 Form),” on page 51, for more information about the sample W-4 JSP customization.

4.1.2 Applicable Pages

You can apply JSP customizations to several types of Kablink Vibe pages:

- **Form:** This is the page used to create a workspace, folder, entry, or comment.
- **View:** This is the page that displays the content of the created workspace, folder, entry, or comment.
- **Landing Page:** This is the page that displays the summarized content of the workspace or folder.

  For information on how to reference a JSP file from a landing page, see “Adding a Custom JSP” in “Creating and Managing Landing Pages” in the *Kablink Vibe OnPrem 3.1 Advanced User Guide*.
- **Mail:** This is the e-mail message that Vibe sends as a result of the *Send mail* footer toolbar function.

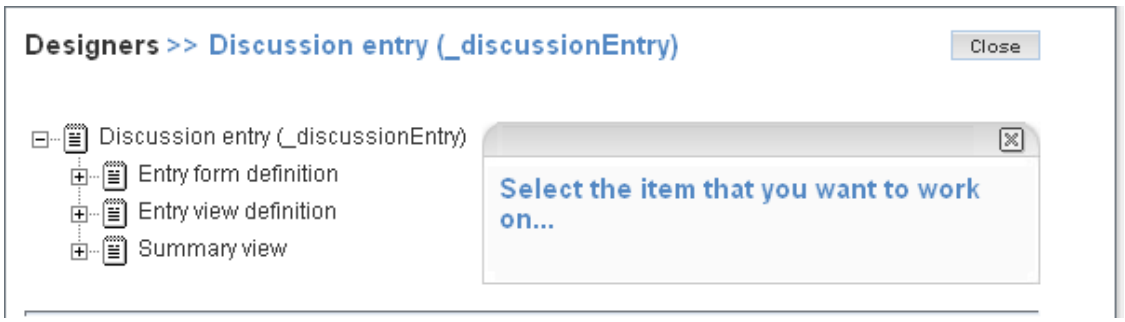
  For example, while viewing an entry, if you click *Send mail*, Vibe places this content into the body of the e-mail message to be sent.

4.1.3 JSPs and the Vibe Designers

Using the Vibe designers, you can remove or add elements of a binder (workspace or folder) or an entry. Using these tools, you can include one or more JSP files to customize content.

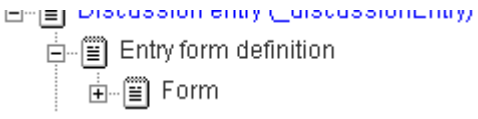
For example, after accessing the form and view designers in the administration portlet, you can edit the definition for a discussion entry:

Figure 4-1 *Using a Designer to Edit the Definition for a Discussion Entry*



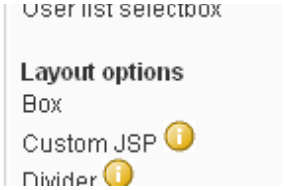
Expand the *Entry form definition* line so that you see the *Form* line:

Figure 4-2 *Expand the Form Definition*



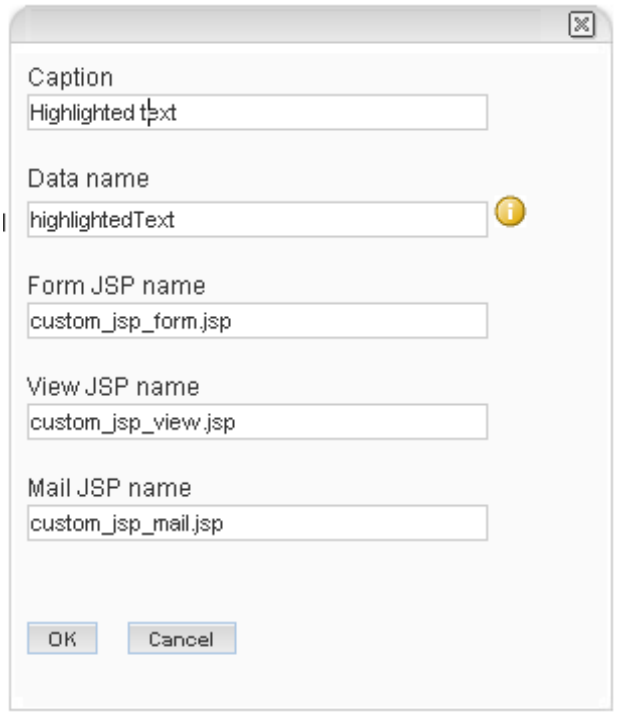
After clicking *Form*, click *Add* in the tools box on the right side of the designer, and notice the *Custom JSP* link in the *Layout options* section:

Figure 4-3 *Locate the Link that Adds the Custom JSP File*



Click *Custom JSP*, and, in the form, provide a caption (which is often used as the displayed title next to the page element), an internal-use name, and the JSP files that implement this customization. The following graphic shows the form used to specify JSP files, and provides the caption *Highlighted text*:

Figure 4-4 *Completed Form for a JSP Customization*

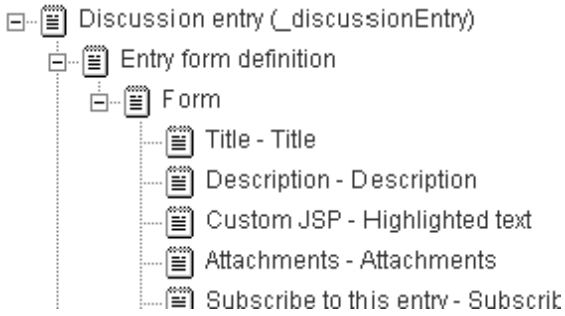


NOTE: Be sure to add the path to the filenames specified in the previous graphic. For example, the correct path for the form customization is `samples/custom_jsp_form.jsp`.

This example uses the sample JSP files provided in the `/WEB-INF/jsp/custom_jsps` directory.

After submitting the form, the JSP customization appears in the definition for the discussion-entry form, which allows you to position it, remove standard elements, and so on. Notice the *Custom JSP - Highlighted text* JSP element positioned just below the standard *Description* element:

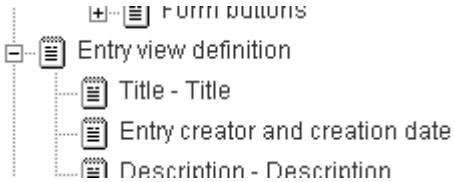
Figure 4-5 *The JSP Element Within the Definition*



Because you provided the file specifications for the form, view, and mail when you added the JSP form element, Vibe remembers this new element and you can locate it within the designer tools using its caption.

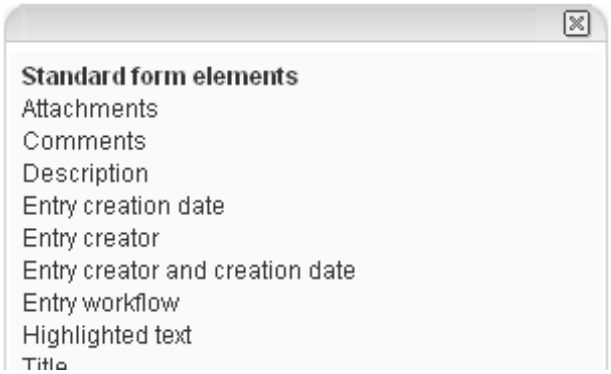
For example, after adding a custom form element, you usually want to add the corresponding custom view element. To do so, click the plus sign (+) next to the *Entry view definition* line to see the elements contained within the view page:

Figure 4-6 *Expand the View Definition*



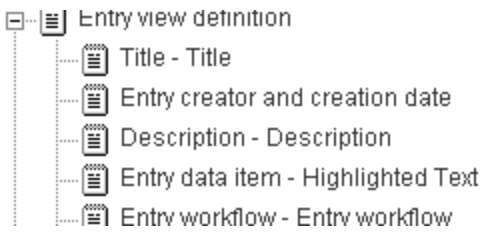
Then, click the *Entry view definition* text to display tools, and then click *Add* in the tools box located on the right side of the designer. Because Vibe recorded the creation of the *Highlighted text* JSP customization, *Highlighted text* now appears in the *Standard form elements* section of the tools box. For example, notice the *Highlighted text* line toward the bottom of the list in this graphic:

Figure 4-7 *The Custom Form Element is Available to the View Definition*



To add the `custom_jsp_view.html` file to the view definition, click *Highlighted text* in the *Standard form elements* section of the tools box, and then click *OK*. Vibe adds the custom element to the view definition, and then you can reposition it within the definition as you desire. The following graphic shows the *Entry data item - Highlighted text* element just under the *Description - Description* element:

Figure 4-8 *Positioning the Custom Element*



See Section 4.2.1, “A JSP That Defines Only One Data Element,” on page 46, for more information about how this customization appears in the UI.

4.1.4 Indexing Issues

When you use one JSP to define one page element, Vibe uses the internal-use data name that you assigned to the JSP in the designer to identify the data as well. In the example in the previous section (see Section 4.1.3, “JSPs and the Vibe Designers,” on page 41), the internal-use data name was *highlightedText*, and it applied to both the JSP file and the actual text collected by the custom text box.

When you use a JSP to define more than one page element, Vibe recognizes the JSPs to the extent that users can create and view a customized entry, but its tools do not recognize the multiple, distinct pieces of custom data within the JSP. As an additional step, you must use the designer to add form elements that match the unique elements found within the JSP file. After you complete this task, Vibe tools then recognize each distinct piece of custom data defined within the JSP. For example, users can now use the advanced search form to search based on exact values provided in those custom elements.

For more information about this additional task that must be performed when adding more than one custom element to the form using a JSP file, see “Identifying Multiple Data Items Defined in One JSP File” on page 62.

4.1.5 JSPs and Vibe Data Access

To enable all of the definition elements and building blocks for your custom JSP, include this line at the top of your JSP file:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>
```

Because JSP customizations within Kablink Vibe are applied to a section of larger web pages, do not specify the `html`, `head`, or `body` HTML tags in your JSP file.

When using custom JSPs, you can customize binders and entries. Binders include all types of workspaces and include folders. Entries include entries as well as comments. Because customization of entries is most common, the rest of this section discusses data accessible when customizing an entry.

To access entry data, you access the properties of the `ssDefinitionEntry` object. For example, this tag from a Vibe JSP file tests for the existence of the entry title:

```
<c:if test="${empty ssDefinitionEntry.title}">
```

Here is a list of some of the object attributes that you can access from your custom JSPs:

```
ssDefinitionEntry.title
ssDefinitionEntry.description
ssDefinitionEntry.fileAttachments
ssDefinitionEntry.creation.principal.title
ssDefinitionEntry.creation.date
ssDefinitionEntry.modification.principal.title
ssDefinitionEntry.modification.date
```

The identifier `principal` corresponds to the user who created or modified the entry.

Vibe provides Java beans that enable a JSP file to access the caption and the internal data name specified for the JSP file using designer tools:

```
property_name
property_caption
```

Use these beans when you are using one JSP file to add one custom element. In this scenario, the name and caption specifications for the JSP file are also applied to the custom data. (When using a JSP file to add more than one element, you specify the `name` and `id` HTML elements explicitly. For more information, see Section 4.2.2, “A JSP-Defined Entry (W-4 Form),” on page 51.)

When accessing the value of properties other than the name and caption, use the `customAttributes` method. For example, the following example shows a tag from a form JSP:

```
<input type="text" id="${property_name}" name="${property_name}"
value="${ssDefinitionEntry.customAttributes[property_name].value}"/>
```

The tag both provides an HTML-tag identifier (`id`) and name (`name`) for the custom form element, and uses the `customAttributes` method to retrieve the value of the custom element (if it exists). Examples in a subsequent section demonstrate the use of beans and the `customAttributes` method, when customizing part of a standard entry form, view, or mail message (see Section 4.2.1, “A JSP That Defines Only One Data Element,” on page 46).

The description of properties, beans, and methods in this section is not exhaustive. To learn about other types of data available to you while using JSP files to create customizations, review the Vibe JSP files, which are found here:

```
/WEB-INF/jsp/definition_elements
```

For example, when searching this folder for occurrences of the `ssDefinitionEntry` object, you can locate this code, which is found in the `popular_view.jsp` file:

```
<c:if test="${!empty ssDefinitionEntry.totalReplyCount}">
```

The `totalReplyCount` property provides an integer that tells you the number of comments for the current entry.

4.1.6 Text Display in the HTML Editor

Unlike most property data, description data almost always requires additional processing within the JSP file. Remember that users create the description text using an HTML editor and that users can place coded items within the text (for example, inline graphics or the double-bracket notation (`[[`*text*`]]`) often used in wiki entries to create a link to another entry).

Consider the following tagging from the `/WEB-INF/jsp/definition_elements/view_entry_data_description.jsp` file:

```
<span><ssf:markup type="view" entity="${ssDefinitionEntry}"><c:out
   value="${ssDefinitionEntry.description.text}" escapeXml="false"/></
ssf:markup></span>
```

The `ssf:markup` tag takes raw text, but processes the information so that it includes properly coded HTML for inline graphics and links to other entries. The `escapeXml` element of the `ssf:markup` tag provides an escape for HTML tags. In other words, the `false` setting for `escapeXml` in the last example indicates that the system should escape the angle bracket characters (`<tag-text>`). In this way, HTML tags are included and properly processed by the browser.

4.1.7 Standard Styles

When creating JSP customizations, you are free to style your page elements as needed. However, if you would like to use the standard styles used by the Vibe product, you can find their definitions here:

`/WEB-INF/jsp/common/ssf_css.jsp`

As you explore the standard JSPs that ship with the product in the `/WEB-INF/jsp/definition_elements` directory, note the name of the standard style, and view its CSS definition in the `ssf_css.jsp` file.

4.2 Examples of Custom Entries

This section provides examples of the two common approaches when using JSPs to customize an entry. Using the first approach, you use a JSP file to customize selected elements within the page segment; then, you can use the Kablink Vibe designers to customize the remaining, non-JSP elements. Using the second approach, you use JSP files to define almost the entire page segment; you can use the designers to include desirable Vibe tools (such as the ability to subscribe to the entry, the send-mail feature, workflow, attachments, and comments), if desired.

This section contains these subsections:

- Section 4.2.1, “A JSP That Defines Only One Data Element,” on page 46
- Section 4.2.2, “A JSP-Defined Entry (W-4 Form),” on page 51

4.2.1 A JSP That Defines Only One Data Element

Kablink Vibe provides sample JSP files for you to apply as a way of learning about JSP customizations. These files create a custom text box on the form, and, after a user enters text and submits the form, these files display the text on the view page and in the mail message (sent using the *Send mail* tool in the footer tool bar).

This section contains these subsections:

- “Understanding the Form Customization” on page 47
- “Understanding the View Customization” on page 49
- “Understanding the Mail Customization” on page 50

Understanding the Form Customization

Let's review the code in the `/WEB-INF/jsp/custom_jsps/custom_jsp_form.jsp` file. First, the file includes a tag that enables definition elements and building blocks for your JSP:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>
```

Because it is defining a page segment within a larger web page, the JSP does not include `html`, `head`,or `body` HTML tags.

Next, the sample JSP file for the form includes bolded text that makes the new element very noticeable on the form:

```
<div style="padding:10px 0px 10px 0px;">
<span class="ss_bold">This is a custom jsp form element</span>
<br/>
<br/>
```

If the caption property is not empty, then the code uses its contents as a title for the custom element:

```
<c:if test="${!empty property_caption}">
  <span class="ss_bold">${property_caption}</span>
  <br/>
</c:if>
```

Finally, the custom JSP file for the form displays the text box. It also displays the current value of the element (if it exists), which is applicable when a user modifies an existing entry. Consider the last portion of code in the custom JSP file for the form:

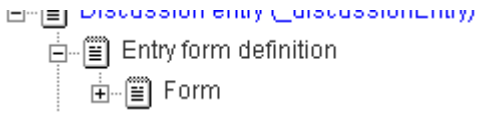
```
<input type="text" id="${property_name}" name="${property_name}"
  value="${ssDefinitionEntry.customAttributes[property_name].value}"/>
</div>
```

When the system performs a get for the `customAttribites` method, it uses the value of the `property_name` bean as a parameter. This action results in the specification of the appropriate custom attribute. Then, this code obtains the `value` property for that custom attribute, displaying it within the text box, if it exists. If it does not exist, the text box is empty.

To use this custom JSP in an entry, you use the designers in the administrative portlet. This section describes how you can add the custom-text customization to a standard discussion entry.

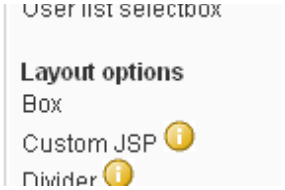
In the administration portlet, access the designer for entries, access the discussion entry, and click *Form*:

Figure 4-9 *Accessing the Form for a Discussion Entry*



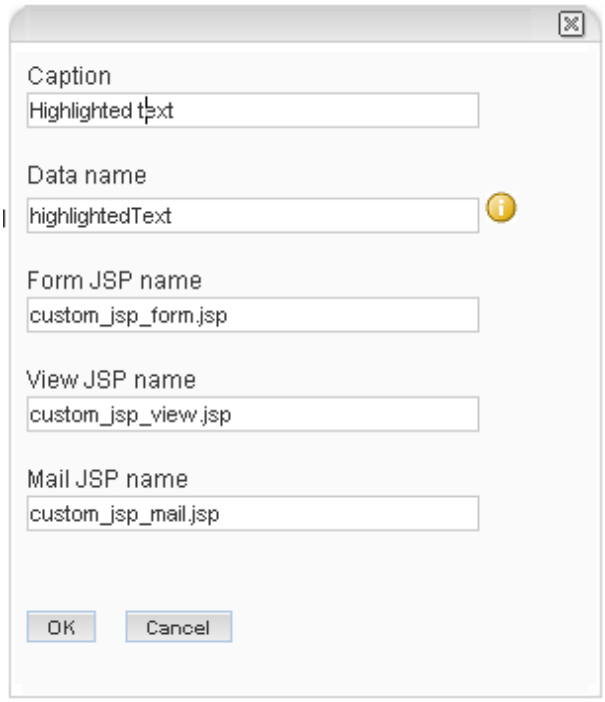
To add the custom JSP to the form, click *Add* in the tools presented in the tools box on the right side of the page, and then click *Custom JSP* in the *Layout options* section:

Figure 4-10 *Adding a Custom JSP to the Form*



Complete the form located in the tools box:

Figure 4-11 *Complete the Form for the Custom JSP*



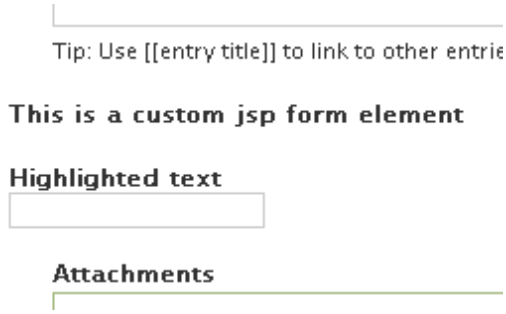
NOTE: Be sure to add the path to the filenames specified in the previous graphic. For example, the correct path for the form customization is `samples/custom_jsp_form.jsp`.

In the previous graphic, the caption for the custom text box is *Highlighted text*. The form in the previous graphic also specifies all three custom JSP files: three separate file specifications for the form, the view, and the mail message.

After you submit the form, position the custom element within the entry definition. The rest of this section assumes a position just below the description.

When a user adds a new entry for a discussion folder, the person sees this custom portion of the form:

Figure 4-12 *The Custom Element on the Form*



Assuming that the user entered *Here is some text* in the *Highlighted text* box, then submitting the form results in the user seeing this in the completed entry:

Figure 4-13 *The Custom Element in the View*

The next section shows the JSP tagging that displays the custom element shown in the previous graphic.

Understanding the View Customization

First, the view JSP includes a tag that enables definition elements and building blocks for your JSP:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>
```

Then, the view JSP file uses the same code that the form JSP used to display the caption for the custom element, if it exists:

```
<div style="padding:10px 0px 10px 0px;">
<c:if test="${!empty property_caption}">
  <span class="ss_bold">${property_caption}</span>
  <br/>
</c:if>
```

Finally, the view JSP uses this code to provide a cyan background color and to display the custom data, if it exists:

```
<div style="background-
color:cyan;">${ssDefinitionEntry.customAttributes[property_name].value}</div>
</div>
```

Understanding the Mail Customization

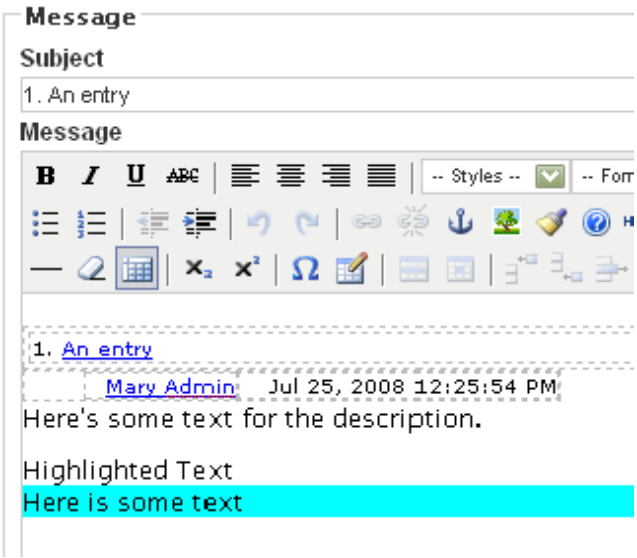
The mail JSP contains the same code as the view JSP:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>

<div style="padding:10px 0px 10px 0px;">
<c:if test="${!empty property_caption}">
  <span class="ss_bold">${property_caption}</span>
  <br/>
</c:if>

<div style="background-
color:cyan;">${ssDefinitionEntry.customAttributes[property_name].value}</div>
</div>
```

Because of space limitations or the desire to summarize entry data in mail messages, you can include or exclude data from the mail JSP, or you can format it differently.

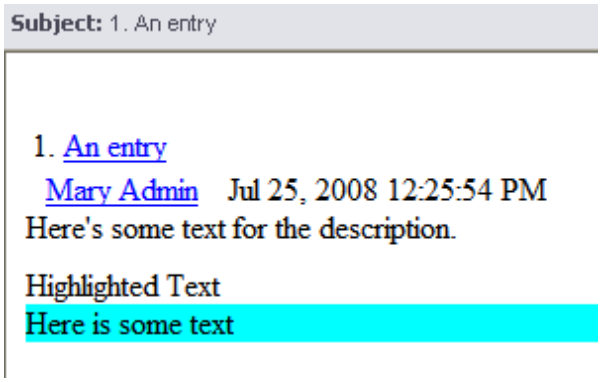
After viewing the entry with the custom element, click *Send mail* in the footer tool bar. The custom element appears on the *Send email* form as follows:

Figure 4-14 *The Custom Element on the Send-Email Form*



The person receiving the e-mail sees this message (different e-mail clients might display the message with slightly different formatting):

Figure 4-15 *The Custom Element in the Received E-mail Message*



4.2.2 A JSP-Defined Entry (W-4 Form)

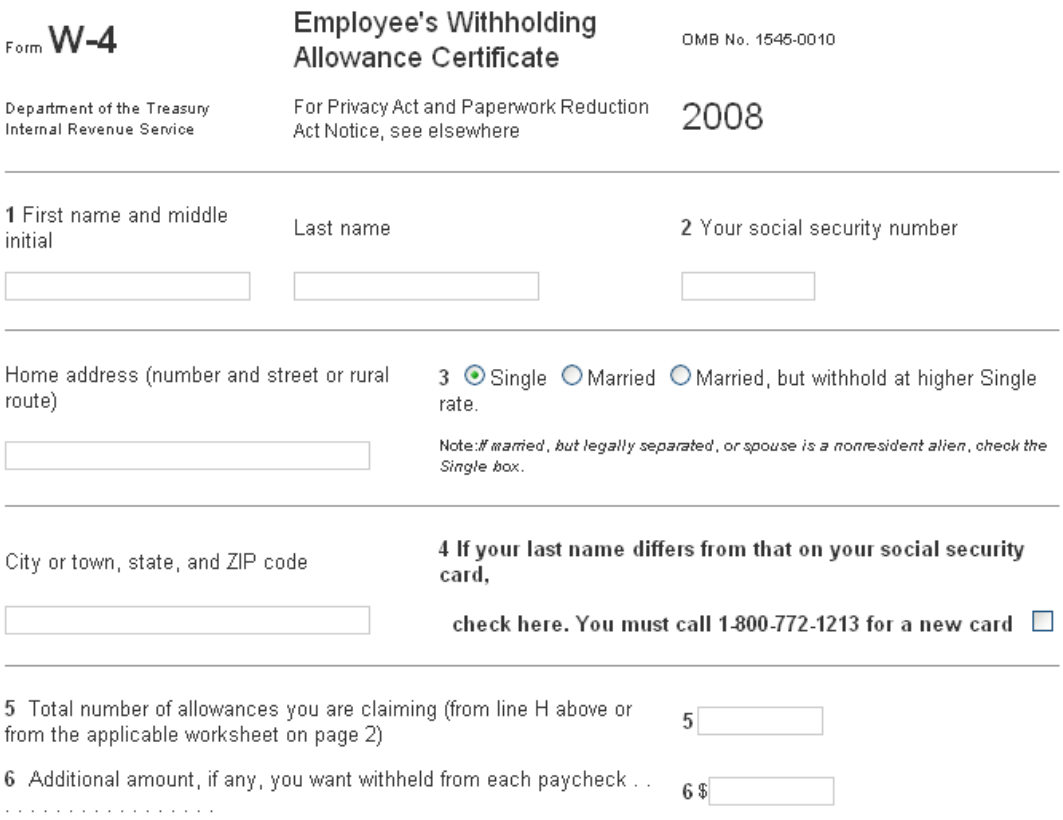
Some custom entries require formatting that is either difficult or impossible to achieve using a separate JSP file for every custom element. For example, there may be an HTML form that you are already using in your organization, and you want to use a nearly identical form within Kablink Vibe.

This section describes custom JSPs that implement a W-4-form application. (A W-4 form is a government form in the U.S.A. that is used to withhold federal taxes from an employee's pay.) This application requires that the entry resemble the paper form, but it allows for the use of standard tools within Vibe. For example, this application allows users to add comments to the W-4 entry, to subscribe to the entry, to send e-mail upon creation of the entry, and to apply a workflow process to the entry. Finally, the application changes the styles of the buttons used to cancel or submit the form.

NOTE: It is possible to replace the entire, standard Kablink Vibe form and to replicate coding for the Vibe tools (attachments, subscribing to the entry, and sending mail upon entry creation) using code in your JSP. However, doing so is significantly more difficult to code. The example in this section documents the best practice of using JSPs to generate most of the form while still enabling the Vibe tools for standard entries.

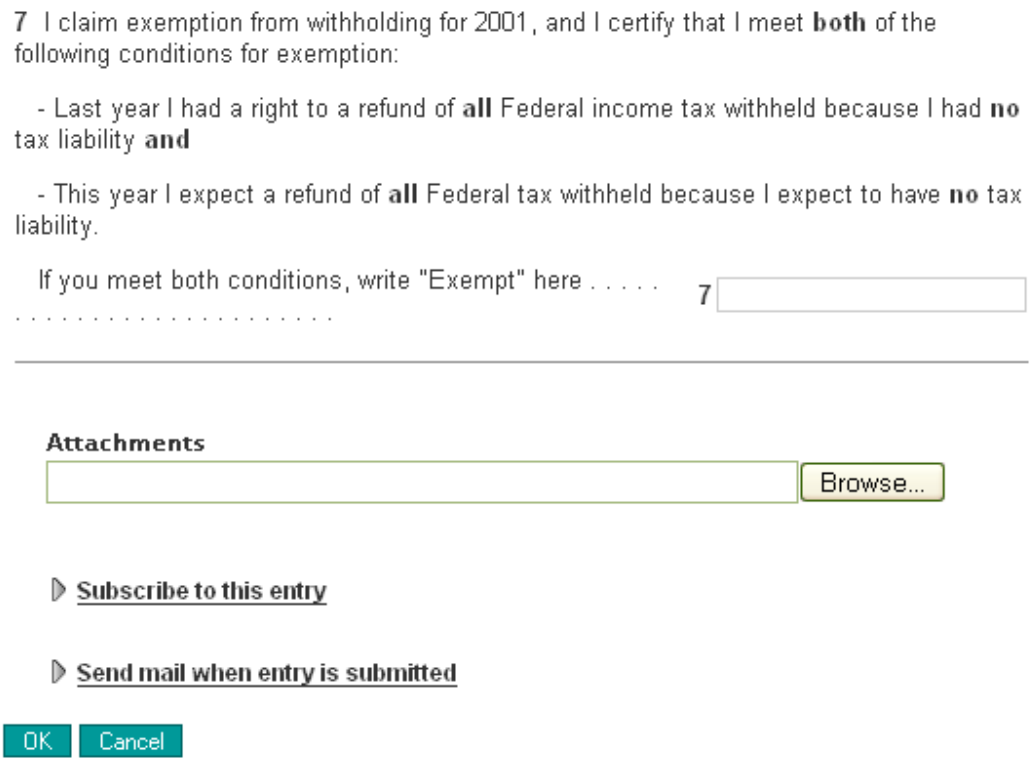
The graphic that follows shows the top portion of the W-4 form as it appears in a window sized to be narrow (note the lack of a "title" text box):

Figure 4-16 *The Top Portion of the W-4 Form*



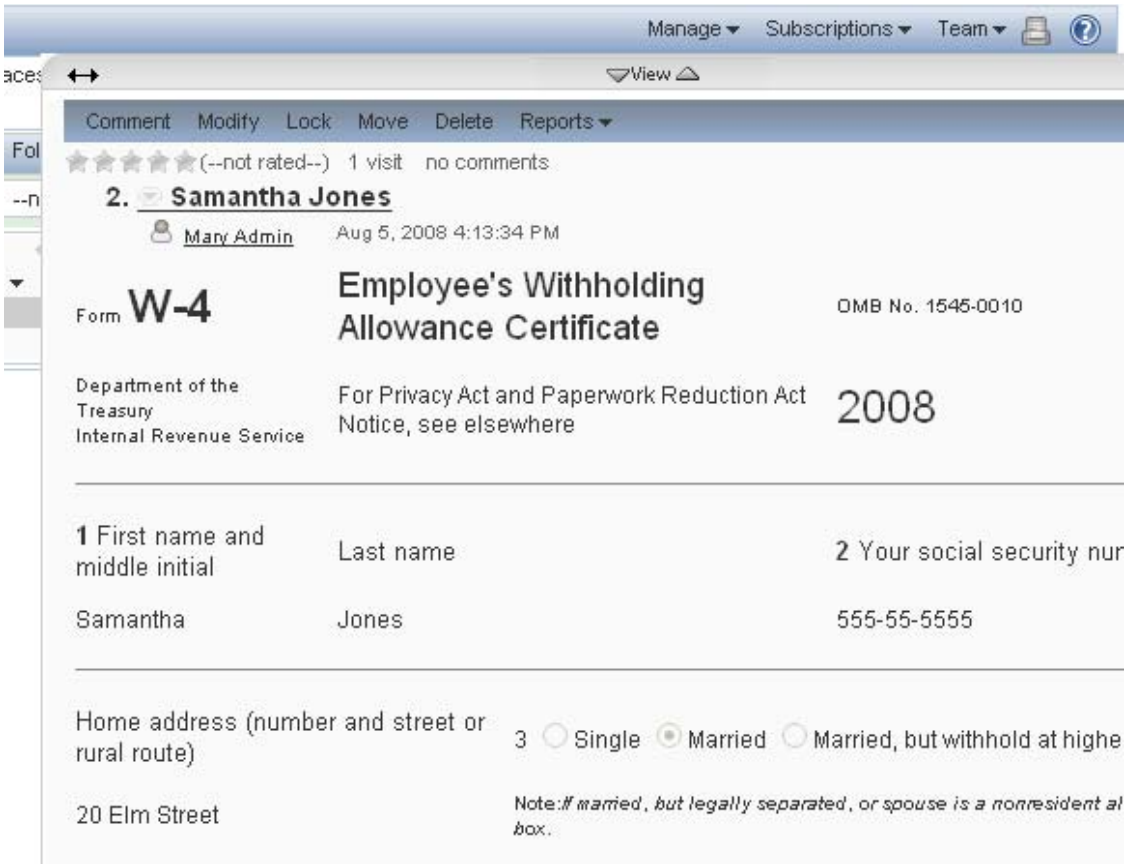
Here is the bottom portion of the W-4 form, which includes standard Vibe tools in between the W-4 content and the buttons:

Figure 4-17 *The Bottom Portion of the W-4 Form*



When a user creates an entry of this type, it appears as follows (note the title of the entry, comprised of two elements found on the form):

Figure 4-18 *Example of a Created Entry for the W-4 Customization*



This section contains these subsections:

- "Creating the Source Files" on page 54
- "Defining Custom Entries" on page 55
- "Coding the Form Files" on page 60
- "Coding the View File" on page 62
- "Identifying Multiple Data Items Defined in One JSP File" on page 62
- "Coding the Mail File" on page 64

Creating the Source Files

Because all JSP customizations are located in the `/custom_jsps` folders, it is recommended that you place application files in separate sub-folders. So, the W-4 application places source files here:

```
/WEB-INF/jsp/custom_jsps/samples/w4
```

These are the files required for the application:

```
w4_form_buttons.jsp
w4_form.jsp
w4_mail.jsp
w4_view.jsp
```

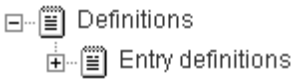
NOTE: When developing the W-4 example, it was helpful to begin with the form and view files containing only standard HTML tagging. For example, to begin, the `w4_form.jsp` file only contained the JSP tag that set definitions, and a standard HTML table containing `input` HTML tags. As the next step, use the instructions in the sections that follow to add one element at a time to both the form and view. In the UI, create or modify an entry of this custom type, provide a value for the custom element, submit the form, and check the view to be sure that Kablink Vibe properly captured the data for the custom element. Proceed with your customization one element at a time, until learning this customization process and debugging are no longer issues.

Defining Custom Entries

Although it is possible to use a JSP file to add a single element to a standard entry, the more likely application is the creation of a new definition for a custom entry. When creating a custom entry, it is common for an application to require that a JSP contain more than one custom element.

To create a new entry definition, access the entry designer in the form and view designers, which are all located in the administrative portlet. Then, click *Entry definitions*:

Figure 4-19 *Invoking Tools for Entry Definitions*

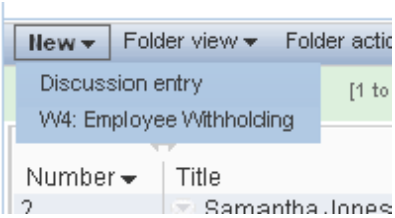


In the tools box on the right side of the designer, click *Add a new entry definition*:

Figure 4-20 *Add a New Entry Definition*

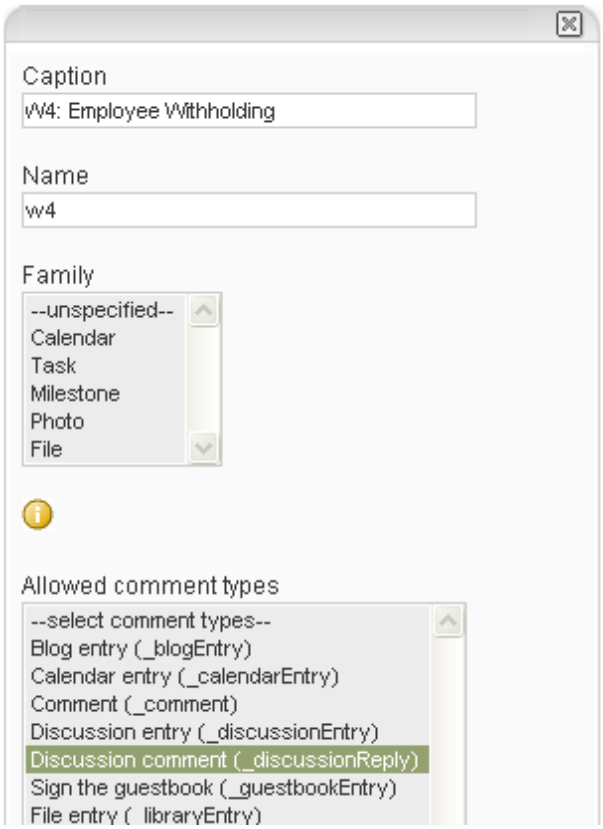
When providing a caption for your custom entry, remember that this text appears when the user clicks the *New* drop-down menu:

Figure 4-21 *The New Drop-Down Menu*



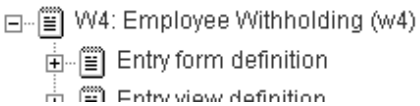
In the entry designer, complete the information for the custom-entry definition and click *OK* at the bottom of the form:

Figure 4-22 *Complete the Form for the Custom-Entry Definition*



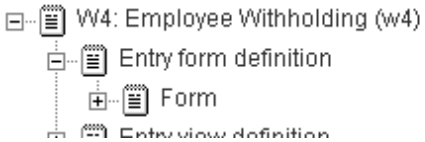
Next, you must provide the JSP file specifications for the form page, view page, and mail message. In the hierarchy to the left of the entry designer, click the plus sign (+) next to *Entry form definition* to expand it:

Figure 4-23 *Access the Form Definition*



Click *Form*:

Figure 4-24 *Access Tools for the Form Definition*

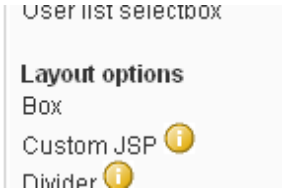


In the tools box on the right side of the designer, click *Add*:

Figure 4-25 *Add an Element to the Form*

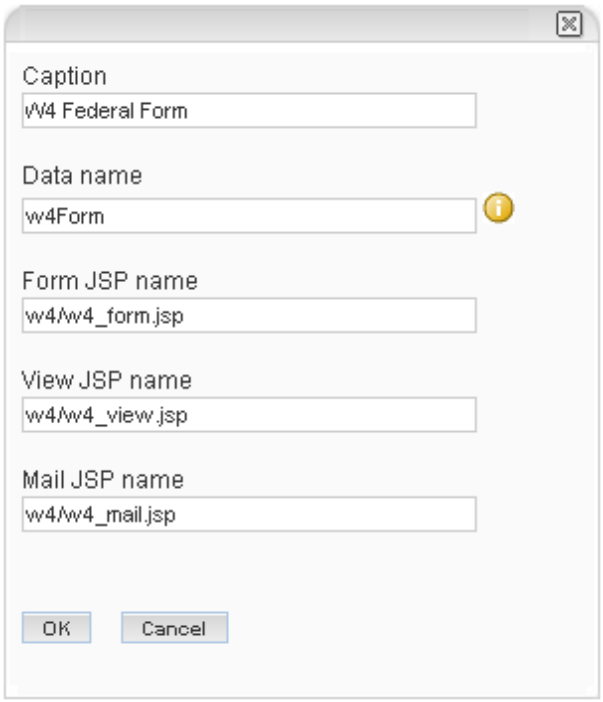
In the *Layout options* section of the tools box, click *Custom JSP*:

Figure 4-26 *Add the Custom JSP to the Form*



Complete the form, providing the file specifications for the custom JSPs:

Figure 4-27 *Providing File Specifications for the JSP Files*



NOTE: Be sure to add the path to the filenames specified in the previous graphic. For example, the correct path for the form customization is `samples/w4/w4_form.jsp`.

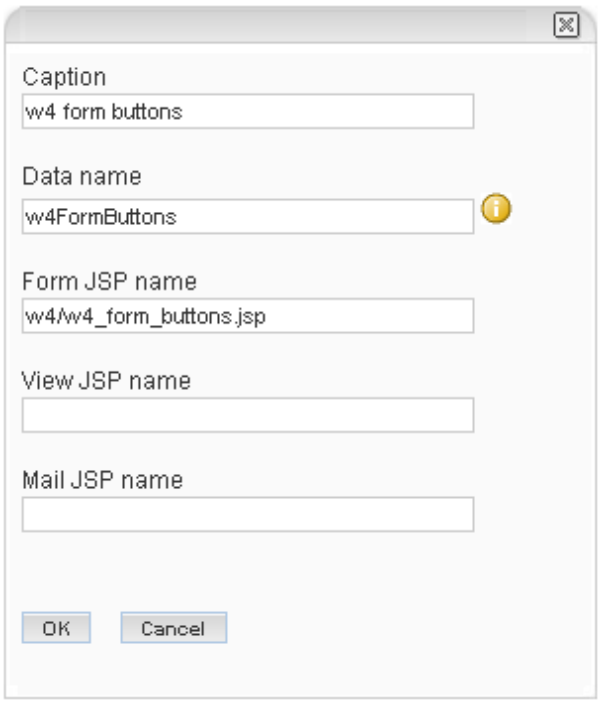
Because Vibe assumes that directories are relative to the `/WEB-INF/jsp/custom_jsps` directory, the file specifications in the previous graphic begin with the `samples/w4/` string, indicating that the files are located in the `/custom_jsps/samples/w4` directory.

Click *OK* to submit the form shown in the previous graphic.

Because the W-4 application customizes the buttons for this form, and because standard tools appear in between the custom form and the buttons (for example, these tools include the form elements that upload and manage attachments), you need to specify the JSP file for the buttons as a separate step in the process.

Click *Entry form definition* again, click *Add* in the tools box, and, in the *Layout options* section, click *Custom JSP*. Fill out the form:

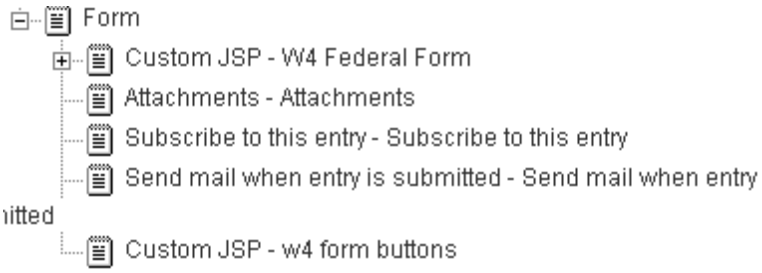
Figure 4-28 *Complete the Form for the Buttons JSP*



NOTE: Be sure to add the path to the filenames specified in the previous graphic. For example, the correct path for the buttons file is `samples/w4/w4_form_buttons.jsp`.

For the W-4 application, the standard title and buttons are not needed (the custom JSPs provide these elements). So, use the entry designer to delete the *Title - Title* and *Form buttons* elements from the form definition. Next, add the *Subscribe to this entry* and *Send mail when entry is submitted* elements. This is the appearance of the form definition when you are finished:

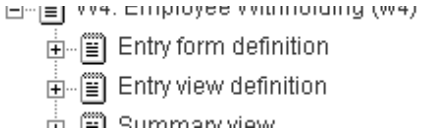
Figure 4-29 *Customized Form Including Both JSP Files*



NOTE: After you learn JSP tagging and debugging, you can continue using the form designer at this stage of the process to enable Kablink Vibe tools, such as indexing custom data and making custom elements available on the advanced search form (see "Identifying Multiple Data Items Defined in One JSP File" on page 62). However, when you are first learning to do these types of customizations, it is recommended that you code and debug each custom element one at a time, ensuring that users can successfully specify data for the custom elements before working further within the designer for the form definition.

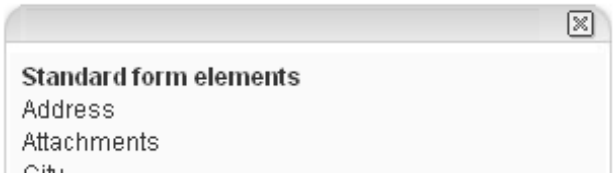
To begin the work needed to define the view definition, click *Entry view definition*:

Figure 4-30 *Access the View Definition*



In the tools box on the right side of the designer, click *Add*. Vibe displays a list of elements that you can add to the view:

Figure 4-31 *View Elements*

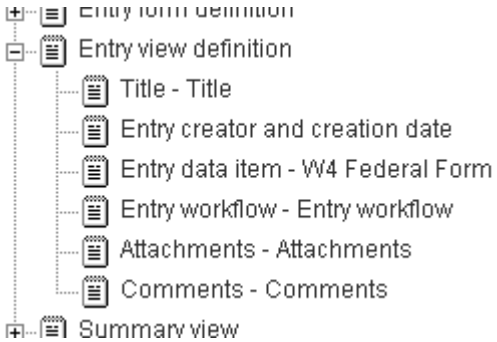


Deeper in the list, locate and click the caption you provided for the form definition (in this example, *W4 Federal Form*):

Figure 4-32 *Clicking the Caption for the Custom JSP*

Use the designer tools to reposition the JSP within the view as desired. Here is one order that you can specify for your view (note the *Entry data item - W4 Federal Form* line):

Figure 4-33 *Repositioning View Elements*



At this point in the process, you have a formatted and minimally functioning custom entry; a user can create an entry, and Vibe can display it. However, users cannot supply values for custom elements until you edit and debug the JSP files for the form and view so that they store and display custom data properly.

Use the information in the next section (see "Coding the Form Files" on page 60) to guide the coding of your JSP files, activating custom elements in your form and view one element at a time. Continue until a user can create a complete entry, providing values for all custom elements on the form and ensuring that the view page displays values for all of the custom elements.

When you are finished debugging, you must go back into the form designer to further identify all custom elements in the form so standard tools in Vibe can access them (see "Identifying Multiple Data Items Defined in One JSP File" on page 62).

Coding the Form Files

Generally, when you are first learning to implement JSP customizations with Kablink Vibe, you add one element, test it by creating an entry and reviewing the view, and then debug. For this reason, it can be helpful to have two windows open displaying Vibe pages: one in which to work within the designer and one in which to check a modified entry to see if the changes in the designer are taking effect.

Also, generally, many entries contain a title element. Vibe automatically uses the value of this element as the title for a created entry. For example, if you specify *Let's talk about the project plan* in the title element of a form for a new discussion topic, then *Let's talk about the project plan* appears as the title of the new entry.

However, the W-4 application does not include a title element on the form. It uses JavaScript and DOM coding to construct a title for the entry. For example, if the person filling out a W-4 form specified *Juanita* as a first name and *Suarez* as a last name, then the created entry combines the values of two form elements and uses *Juanita Suarez* as the entry title. This customization was included to illustrate possibilities for your JSP customizations.

So, in the W-4 customization, the first-name and last-name elements need to be added to the form before the view can properly display its entry title.

As mentioned, you include the JSP tag that establishes supporting definitions for Vibe customizations. Then, you specify HTML code needed for your specific customization. For the W-4 application, the form and view use a standard HTML table. Here are the first few lines of the W-4 form:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>


<table width="100%" cellpadding="6" cellspacing="0" summary="W4 form">
<tr>
<td><font size="-2">Form</font>
<b><font size="+2">W-4</font></b></td>
<td colspan="5"><font
size="+1">Employee's Withholding
Allowance Certificate</font></td>
<td><font size="-2">OMB No. 1545-0010</font></td>
</tr>
```

Because the JSP customization defines a page segment within a larger web page, it does not specify `html`, `head`, or `body` HTML tags.

The code that implements the first name and last name is the same syntactically as the code that implements single-element JSP files (see Section 4.2.1, “A JSP That Defines Only One Data Element,” on page 46). You provide identifiers for the custom elements, and you use the `ssDefinitionEntry` object, its attributes, and their values, to place existing data into the form element (if it exists). This is the code for the `firstName` and `lastName` attributes:

```
<tr>
<td colspan="2"><input type="text" name="firstName" id="firstName"
size="25" value="${ssDefinitionEntry.customAttributes['firstName'].value}" />
></td>
<td colspan="3"><input type="text" name="lastName" id="lastName"
size="25" value="${ssDefinitionEntry.customAttributes['lastName'].value}" />
></td>
```

Generally, the process of coding all other form elements is the same. However, coding radio buttons, check boxes, and select boxes (there are no select boxes in the W-4 example) require the knowledge of some additional details (see Section 4.3, “Examples of Complex, HTML Data Types,” on page 65).

The initial part of the JSP file for the buttons contains style information to be applied to the *OK* and *Cancel* buttons:

```
<style>
input.custom_submit {
  background-color: #009999;
  border: 1px solid #006666;
  color: #ffffff;
  font-size: 12px;
  padding: 0px 6px 0px 6px;
  cursor: pointer;
  white-space: nowrap;
}
</style>
```

The next section of code in the JSP file for the buttons contains the JavaScript that concatenates the first and last names, and assigns that string to the `title` form element:

```
<script language="JavaScript" type="text/javascript">
function mySubmit() {
    self.document.form1.title.value = self.document.form1.firstName.value + "
" + self.document.form1.lastName.value;
}
</script>
```

When Vibe generates a form that creates an entry, the standard identifier for the form is `form1`.

Because this application required the removal of the title element of the form using the designers, the internal mechanism for recognizing and storing the data for a title has been removed. The next line in the JSP file for the buttons uses a “hidden” tag to replace the mechanisms removed by the deletions in the designer:

```
<input type="hidden" name="title" value="" />
```

Execution of the JavaScript routine `mySubmit` provides a value for the `title` element.

The final segment of code in the JSP file for the buttons contains HTML for the *OK* and *Cancel* buttons:

```
<input type="submit" class="custom_submit" name="okBtn" value="OK"
onClick="ss_buttonSelect('okBtn');mySubmit();"/>
<input type="submit" class="custom_submit" name="cancelBtn" value="Cancel"
onClick="ss_buttonSelect('cancelBtn'); " />
```

The HTML for the *OK* button includes an `onClick` specification that executes the `mySubmit` JavaScript routine, creating the value of the entry title.

Coding the View File

For the W-4 application, the JSP file for the view is a table identical to the one included in the form, except that it only displays values for each of the custom elements (if they exist).

First, include the tag that establishes the JSP definitions:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>
```

Here is a sample of the code from the view file that displays the first name, last name, and social security number:

```
<tr>
<td colspan="2">${ssDefinitionEntry.customAttributes['firstName'].value}</td>
<td colspan="3">${ssDefinitionEntry.customAttributes['lastName'].value}</td>
<td colspan="2">${ssDefinitionEntry.customAttributes['ssn'].value}</td>
</tr>
```

The identifiers `firstName`, `lastName`, and `ssn` map to the `name` and `id` elements for the HTML `input` tags in the JSP for the form.

Complete the table so that it displays all of the custom elements.

Coding for most of the elements is virtually identical to the code displayed in the last example. However, displaying data from radio buttons, check boxes, and select boxes (there are no select boxes in the W-4 example) require the knowledge of some additional details (see Section 4.3, “Examples of Complex, HTML Data Types,” on page 65).

Identifying Multiple Data Items Defined in One JSP File

After adding and testing all custom elements, the custom entry for the W-4 application is complete, which means that someone can create and view an entry containing values for all custom elements. However, one of the desirable features ofKablink Vibe is the integration between features and tools. To enable this integration, you need to use the designers to report to the system that this entry includes custom data elements. As one example, after you report the existence of these elements, Vibe is able to index the custom data, which, in turn, allows users to perform advanced searches based on this custom data.

In summary, to report the custom elements to the system, use the form designer for the entry to add elements that use the same name and HTML data type (text box, radio button, and so on) as the element in the JSP files.

To report custom elements, access the definition of the custom entry (*W4: Employee Withholding (w4)*), and open the form definition by clicking the plus sign (+) next to both *Entry form definition* and *Form*. Then, click *Custom JSP - W4 Federal Form*. Finally, add elements as children to *Custom JSP - W4 Federal Form*.

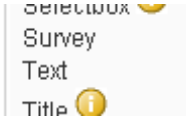
The instructions that follow show you how to add the first-name element.

As a reminder, here is the HTML from the JSP for the form that establishes the first-name element, whose form element is a text box:

```
<input type="text" name="firstName" id="firstName"
size="25" value="${ssDefinitionEntry.customAttributes['firstName'].value}" />
```

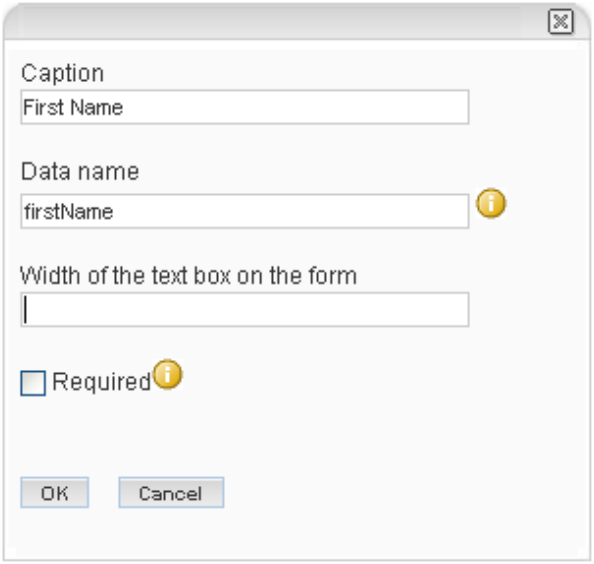
In the tools box on the right side of the designer, click *Text*:

Figure 4-34 *Adding a Text Element for Indexing Purposes*



In the form, specify the same data name (*firstName*) as the `name` and `id` elements used in the HTML `input` tag found in the JSP file, and provide the same caption:

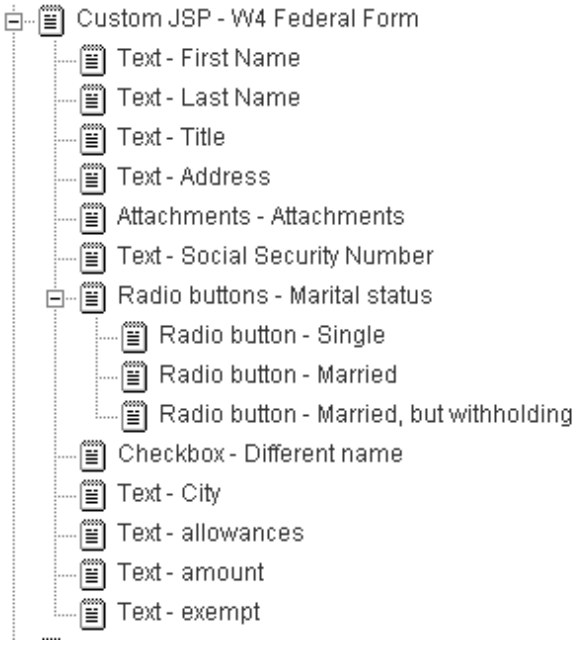
Figure 4-35 *Match the Name with the One Found in the JSP File*



None of the other form values have an effect. Click the *OK* button to submit the form.

Repeat this action for all elements defined in the JSP file, including the hidden title element. The order of the elements does not have to match the order of the elements in the JSP file. When you are finished, the definition includes all of the elements shown in this graphic:

Figure 4-36 *All Elements Specified for Indexing Purposes*



Notice that the elements appear as children to the custom JSP within the hierarchy (as opposed to peers).

Coding the Mail File

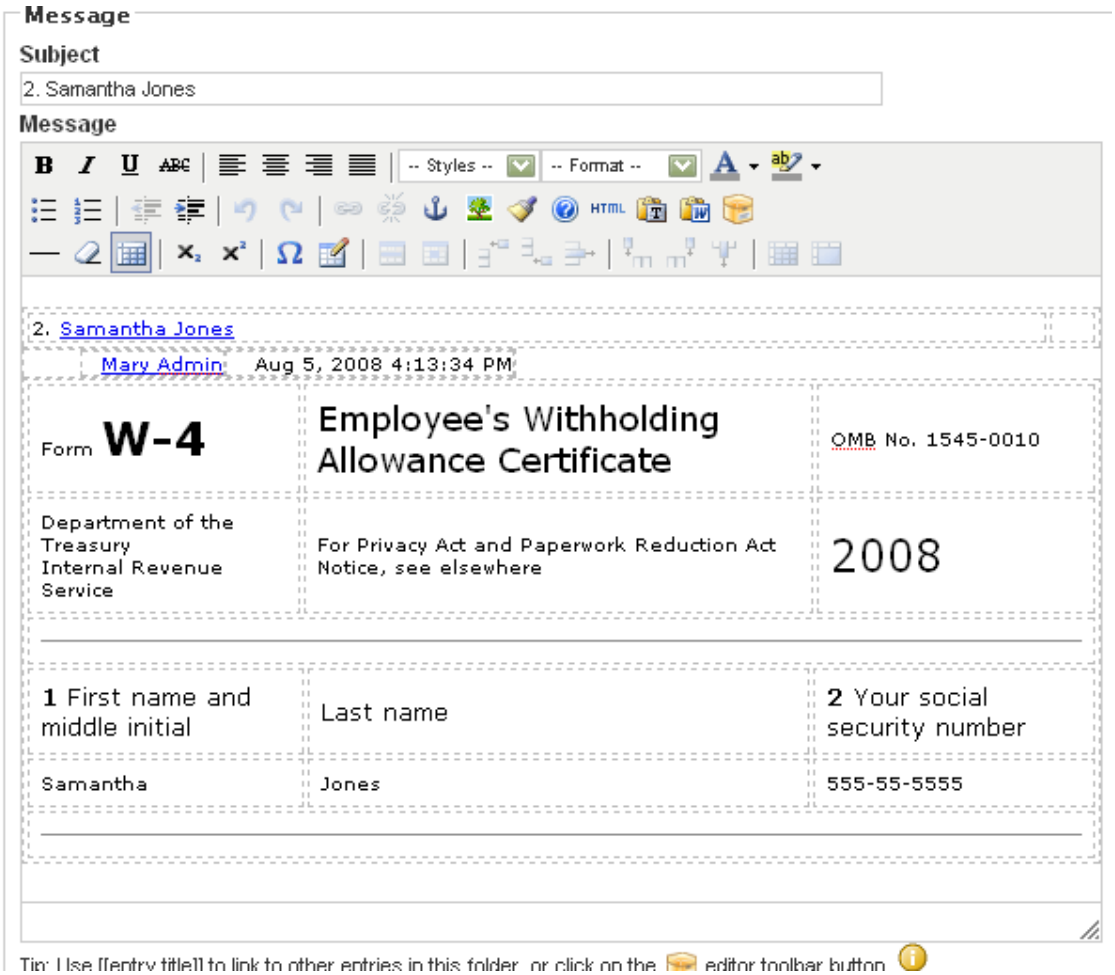
The W-4 application is a good example of a custom entry for which you might want to provide only a summary of the information in an e-mail message. For example, this is all of the code in the JSP for mail:

```
<%@ include file="/WEB-INF/jsp/definition_elements/init.jsp" %>

<table width="100%" class="border" cellpadding="6"
summary="W4 Information">
<tr>
<td><font size="-2">Form</font> <b><font size="+2">W-4</font></b></td>
<td colspan="5"><font size="+1">Employee's Withholding
Allowance Certificate</font></td>
<td><font size="-2">OMB No. 1545-0010</font></td>
</tr>
<tr>
<td><font size="-3">Department of the Treasury<br />Internal
Revenue Service</font></td>
<td colspan="5">For Privacy Act and Paperwork Reduction Act Notice, see
elsewhere</td>
<td><font size="+2">2008</font></td>
</tr>
<tr><td colspan="7"><hr size="1" noshade="noshade" /></td></tr>
<tr>
<td colspan="2"><font size="-1"><b>1</b> First name and
middle initial</font></td>
<td colspan="3"><font size="-1">Last name</font></td>
<td colspan="2"><font size="-1"><b>2</b> Your social security
number</font></td>
```

```
</tr>
<tr>
<td colspan="2">${ssDefinitionEntry.customAttributes['firstName'].value}</td>
<td colspan="3">${ssDefinitionEntry.customAttributes['lastName'].value}</td>
<td colspan="2">${ssDefinitionEntry.customAttributes['ssn'].value}</td>
</tr>
<tr><td colspan="7"><hr size="1" noshade="noshade" /></td></tr>

</table>
```

When a user views an entry of this type and clicks *Send mail*, that person sees:

Figure 4-37 *Showing Partial Information in Mail*



4.3 Examples of Complex, HTML Data Types

This section provides examples of capturing and displaying custom data for radio buttons, check boxes, and select boxes. This section contains these sections:

- Section 4.3.1, “Radio Buttons,” on page 66
- Section 4.3.2, “Check Boxes,” on page 66
- Section 4.3.3, “Select Boxes,” on page 67

4.3.1 Radio Buttons

This section describes the radio buttons used in the W-4 application described in a previous section (see Section 4.2.2, “A JSP-Defined Entry (W-4 Form),” on page 51). This is the code from the JSP for the W-4 form that implements the Single, Married, and “Married but...” radio buttons:

```
<td colspan="4">
<font size="-1"><b>3</b> <input type="radio"  name="status"
value="Single"

<c:if test="${ssDefinitionEntry.customAttributes['status'].value == 'Single'
|| ssDefinitionEntry.customAttributes['status'].value != 'Married' &&
ssDefinitionEntry.customAttributes['status'].value !=
'MarriedBut'}">checked="checked"</c:if> />Single

<input type="radio"  name="status" value="Married" <c:if
test="${ssDefinitionEntry.customAttributes['status'].value ==
'Married'}">checked="checked"</c:if> />Married

<input type="radio"  name="status" value="MarriedBut" <c:if
test="${ssDefinitionEntry.customAttributes['status'].value ==
'MarriedBut'}">checked="checked"</c:if> />Married, but withhold at higher
Single rate.
</font></td>
```

The JSP tagging code selects the *Single* radio button by default.

This is the code from the JSP for the view that displays all radio buttons, including the selected button, but displays them as being disabled (so users do not attempt to change the value of the element on the view):

```
<td colspan="4">3 <input type="radio"  name="status" value="Single"

<c:if test="${ssDefinitionEntry.customAttributes['status'].value == 'Single'
|| ssDefinitionEntry.customAttributes['status'].value != 'Married' &&
ssDefinitionEntry.customAttributes['status'].value !=
'MarriedBut'}">checked="checked"</c:if> DISABLED />Single

<input type="radio" name="status" value="Married" <c:if
test="${ssDefinitionEntry.customAttributes['status'].value ==
'Married'}">checked="checked"</c:if> DISABLED />Married

<input type="radio" name="status" value="MarriedBut" <c:if
test="${ssDefinitionEntry.customAttributes['status'].value ==
'MarriedBut'}">checked="checked"</c:if> DISABLED />Married, but withhold at
higher
Single rate.
</td>
```

4.3.2 Check Boxes

This section describes the check boxes used in the W-4 application described in a previous section (see Section 4.2.2, “A JSP-Defined Entry (W-4 Form),” on page 51). This is the code from the JSP for the W-4 form that implements the differing-names check box:

```
<td colspan="4">
<font size="-1"><b>  check
here. You must call 1-800-772-1213 for a new card </b>
</font>

<input type="checkbox" name="boxFour"
<c:if test="${ssDefinitionEntry.customAttributes['boxFour'].value ==
'true'}">checked="checked"</c:if>/>
</td>
```

Here is the code in the JSP for the view that displays the disabled check box (so users do not attempt to change the value of the element on the view):

```
<td colspan="4">
<font size="-1"><b>  check here. You must call
1-800-772-1213 for a new card </b></font>

<input type="checkbox" name="boxFour"
<c:if test="${ssDefinitionEntry.customAttributes['boxFour'].value ==
'true'}">checked="checked"</c:if> DISABLED />
</td>
```

4.3.3 Select Boxes

All other data types involve the storage of a single value. For both single-value and multiple-value select boxes,Kablink Vibe stores the selected value or values in a value set. You can access the attribute's value set using this notation:

`ssDefinitionEntry.customAttributes[`*`attributeName`*`].valueSet`

You can use the `foreach` tag to loop through the value set, using that information to present either form or view elements.

The W-4 application did not include an example of a select box, so this section presents code that implements a multiple-value select box that allows user to choose between the written numbers one through four:

Figure 4-38 *Example of a Select Box on a Form*

The previous graphic shows how the list looks by default, with no selections. Your code also needs to display currently selected items upon entry modification.

This code labels the form element and initializes flag variables:

```
<b>Test select box:</b><br/>

<c:set var="matchOne" value="0"/>
<c:set var="matchTwo" value="0"/>
<c:set var="matchThree" value="0"/>
<c:set var="matchFour" value="0"/>
```

Next, loop through the value set, and increment the flag that corresponds to a selected item:

```
<c:forEach var="selection"
items="${ssDefinitionEntry.customAttributes['testSelection'].valueSet}" >
<c:if test="${selection == 'one'}"><c:set var="matchOne" value="1"/></c:if>
<c:if test="${selection == 'two'}"><c:set var="matchTwo" value="1"/></c:if>
<c:if test="${selection == 'three'}"><c:set var="matchThree" value="1"/></
c:if>
<c:if test="${selection == 'four'}"><c:set var="matchFour" value="1"/></c:if>
</c:forEach>
```

Then, in the `option` HTML tags for the select list, use an `if` JSP tag to indicate that an option is selected:

```
<select name="testSelection" id="testSelection" multiple="multiple">
<option value="one" name="one" id="one"
<c:if test="${matchOne == 1}">selected="selected" </c:if>>One</option>
<option value="two" name="two" id="two"
<c:if test="${matchTwo == 1}">selected="selected" </c:if>>Two</option>
<option value="three" name="three" id="three"
<c:if test="${matchThree == 1}">selected="selected" </c:if>>Three</option>
<option value="four" name="four" id="four"
<c:if test="${matchFour == 1}">selected="selected" </c:if>>Four</option>
</select>
```

In the view, you can display the selections in many ways. Here is one example:

Figure 4-39 *Example of Displaying Selections on the View Page*

This code displays the bolded header, and loops through the value set, counting the selections:

```
<b>Test selection:</b>

<c:set var="numSelections" value="0"/>
<c:forEach var="selection"
items="${ssDefinitionEntry.customAttributes['testSelection'].valueSet}" >
<c:set var="numSelections" value="${numSelections + 1}"/>
</c:forEach>
```

Then, this code displays the written words that match the selections, separating them with commas, and does not display a comma after the last selection.

```
<c:set var="count" value="0"/>
<c:forEach var="selection"
items="${ssDefinitionEntry.customAttributes['testSelection'].valueSet}" >
<c:set var="count" value="${count + 1}"/>
<c:if test="${selection == 'one'}">One</c:if>
<c:if test="${selection == 'two'}">Two</c:if>
<c:if test="${selection == 'three'}">Three</c:if>
<c:if test="${selection == 'four'}">Four</c:if>
<c:if test="${count != numSelections}">, </c:if>
</c:forEach>
```

Web Services Operations

A

Much of the information in this section references Teaming, which is the product name for the Vibe product for versions prior to Vibe 3. This information applies to Vibe 3 as well as earlier versions of the product.

This section provides alphabetized reference pages for the Web services operations of Kablink Teaming 2.0 and later.

NOTE: All examples in this reference section use the Kablink Vibe client library. See Section 1.6, “Client Stubs,” on page 16, for more information about the client library and other ways to call Vibe Web services operations.

- “admin_destroyApplicationScopedToken” on page 72
- “admin_getApplicationScopedToken” on page 72
- “binder_addBinder” on page 73
- “binder_copyBinder” on page 74
- “binder_deleteBinder” on page 74
- “binder_deleteTag” on page 75
- “binder_getBinder” on page 76
- “binder_getBinderByPathName” on page 77
- “binder_getFileVersions” on page 77
- “binder_getFolders” on page 78
- “binder_getSubscription” on page 79
- “binder_getTags” on page 80
- “binder_getTeamMembers” on page 81
- “binder_getTrashEntries” on page 81
- “binder_indexBinder” on page 82
- “binder_indexTree” on page 83
- “binder_modifyBinder” on page 84
- “binder_moveBinder” on page 84
- “binder_preDeleteBinder” on page 85
- “binder_removeFile” on page 86
- “binder_restoreBinder” on page 86
- “binder_setDefinitions” on page 87
- “binder_setFunctionMembership” on page 88
- “binder_setFunctionMembershipInherited” on page 89
- “binder_setOwner” on page 89
- “binder_setSubscription” on page 90
- “binder_setTag” on page 91

- “binder_setTeamMembers” on page 92
- “binder_testAccess” on page 92
- “binder_uploadFile” on page 93
- “definition_getDefinitionAsXML” on page 94
- “definition_getDefinitionByName” on page 95
- “definition_getDefinitions” on page 95
- “definition_getLocalDefinitionByName” on page 96
- “definition_getLocalDefinitions” on page 97
- “folder_addEntry” on page 98
- “folder_addEntryWorkflow” on page 99
- “folder_addMicroBlog” on page 99
- “folder_addReply” on page 100
- “folder_copyEntry” on page 101
- “folder_deleteEntry” on page 101
- “folder_deleteEntryTag” on page 102
- “folder_deleteEntryWorkflow” on page 103
- “folder_getEntries” on page 103
- “folder_getEntry” on page 104
- “folder_getEntryByFileName” on page 105
- “folder_getEntryTags” on page 106
- “folder_getFileVersions” on page 106
- “folder_getSubscription” on page 107
- “folder_modifyEntry” on page 108
- “folder_modifyWorkflowState” on page 108
- “folder_moveEntry” on page 109
- “folder_preDeleteEntry” on page 110
- “folder_removeFile” on page 110
- “folder_reserveEntry” on page 111
- “folder_restoreEntry” on page 111
- “folder_setEntryTag” on page 112
- “folder_setRating” on page 112
- “folder_setSubscription” on page 113
- “folder_setWorkflowResponse” on page 114
- “folder_synchronizeMirroredFolder” on page 115
- “folder_unreserveEntry” on page 115
- “folder_uploadFile” on page 116
- “folder_uploadFileStaged” on page 117
- “ical_uploadCalendarEntriesWithXML” on page 118

- “ldap_synchAll” on page 119
- “ldap_synchUser” on page 119
- “license_getExternalUsers” on page 120
- “license_getRegisteredUsers” on page 121
- “license_updateLicense” on page 121
- “migration_addBinder” on page 122
- “migration_addBinderWithXML” on page 122
- “migration_addEntryWorkflow” on page 123
- “migration_addFolderEntry” on page 124
- “migration_addFolderEntryWithXML” on page 125
- “migration_addReply” on page 126
- “migration_addReplyWithXML” on page 127
- “migration_uploadFolderFile” on page 128
- “migration_uploadFolderFileStaged” on page 130
- “profile_addGroup” on page 131
- “profile_addGroupMember” on page 132
- “profile_addUser” on page 132
- “profile_addUserWorkspace” on page 133
- “profile_deletePrincipal” on page 134
- “profile_getFileVersions” on page 134
- “profile_getGroup” on page 135
- “profile_getGroupByName” on page 136
- “profile_getGroupMembers” on page 137
- “profile_getPrincipals” on page 137
- “profile_getUser” on page 138
- “profile_getUserByName” on page 139
- “profile_getUsers” on page 140
- “profile_getUserTeams” on page 141
- “profile_modifyGroup” on page 141
- “profile_modifyUser” on page 142
- “profile_removeFile” on page 142
- “profile_removeGroupMember” on page 143
- “profile_uploadFile” on page 144
- “search_getFolderEntries” on page 145
- “search_getTeams” on page 146
- “search_getWorkspaceTreeAsXML” on page 146
- “search_search” on page 147
- “template_addBinder” on page 148

- “template_getTemplates” on page 149
- “zone_addZone” on page 150
- “zone_deleteZone” on page 151
- “zone_modifyZone” on page 151

admin_destroyApplicationScopedToken

Destroys an application-scoped token.

Syntax

public void admin_destroyApplicationScopedToken(String *accessToken*, String *token*);

Description

The `admin_destroyApplicationScopedToken` operation destroys a previously acquired application scoped token.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

token

The string representation of the previously acquired application-scoped token that you want to destroy.

return_value

None.

admin_getApplicationScopedToken

Requests an application-scoped token on behalf of the user.

Syntax

public String admin_getApplicationScopedToken(String *accessToken*, long *applicationId*, long *userId*);

Description

The `admin_getApplicationScopedToken` operation requests the system to create and return an application-scoped token on behalf of the user.The token is subsequently utilized by the application.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

applicationId

The identifier of the application set up with the Vibe system.

userId

The identifier of the user on whose behalf you want the token to be created.

return_value

A string representation of the requested token.

binder_addBinder

Adds an unconfigured binder to the workspace tree hierarchy.

Syntax

public long binder_addBinder(String *accessToken*, Binder *binder*);

Description

The `binder_addBinder` operation adds either a workspace or folder to the hierarchy.

To add a fully configured binder, use the `template_addBinder` operation instead.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part implementing a remote application, or the null value.

binder

Data and methods for the Java `Binder` object, defined in the Vibe source code.

return_value

The identifier of the new binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- template_addBinder (page 148)

binder_copyBinder

Creates a new binder identical to an existing one.

Syntax

public long binder_copyBinder(String *accessToken*, long *sourceId*, long *destinationId*, boolean cascade);

Description

The `binder_copyBinder` operation copies an existing workspace or folder, and creates a new one.

Vibe automatically copies all non-binder content (entries and comments). As an option, you can replicate the source binder's sub-binders (sub-workspaces or sub-folders).

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

sourceId

The identifier of the binder you want to copy.

destinationId

The binder identifier of the parent for the new workspace or folder.

cascade

A Boolean value indicating whether you want to copy the source binder's sub-binders (sub-workspaces and sub-folders).

return_value

The identifier of the new binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)

binder_deleteBinder

Deletes a binder.

Syntax

public void binder_deleteBinder(String *accessToken*, long *binderId*, boolean *deleteMirroredSource*);

Description

The `binder_deleteBinder` operation deletes a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier of the workspace or folder you want to delete.

deleteMirroredSource

Deletes the source directory, if the folder being deleted is a mirrored folder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_deleteTag

Removes a tag from a binder.

Syntax

public void binder_deleteTag(String *accessToken*, long *binderId*, String *tagId*);

Description

The `binder_deleteTag` operation removes a tag from a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the binder that applies the tag you want to remove.

tagId

The tag you want to remove.

return_value

None.

Example

```
public static void checkTags(long binderId) throws Exception{ ...
Tag[] tags = setupTags(binderId);
for (int i=0; i<tags.length; ++i) {stub.binder_setTag(null, tags[i]); }
tags = stub.binder_getTags(null, binderId);
validateTags(tags);
stub.binder_deleteTag(null, binderId, tags[0].getId());
```

This code is taken from the source code for the `teamingservice-client-with-stub.bat` file.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getBinder

Accepts a binder identifier to get information about a binder.

Syntax

public Binder binder_getBinder(String *accessToken*, long *binderId*, boolean *includeAttachments*);

Description

The `binder_getBinder` operation gets information about a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder for which you want information.

includeAttachments

A Boolean value that indicates whether you want Vibe to return attached files.

By default, workspaces do not include attached files. However, users can use the designers to define workspaces that do include attached files.

return_value

A Binder Java object that contains data and methods for the requested binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getBinderByPathName

Accepts a directory specification to get information about a binder.

Syntax

public Binder binder_getBinderByPathName(String *accessToken*, String *pathName*, boolean *includeAttachments*);

Description

The `binder_getBinderByPathName` operation uses a workspace-hierarchy path name to get information about a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

pathName

The titles of the binder for which you want information, preceded by the titles of all of its parents, separated by slashes:

`Workspaces / Global workspaces / wsOrfolderTitle / ... / titleTargetWS`

includeAttachments

A Boolean value that indicates whether you want Vibe to return attached files.

By default, workspaces do not include attached files. However, users can use the designers to define workspaces that do include attached files.

return_value

A Binder Java object that contains data and methods for the requested binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getFileVersions

Returns information about the versions of a file.

Syntax

public void binder_getFileVersions(String *accessToken*, long *binderID*, String *fileName*);

Description

The `binder_getFileVersion` operation retrieves information about the versions of a file associated with a workspace or folder.

By default, workspaces and folders do not contain files, but users can alter definitions by using the designers in the user interface so that a custom workspace or folder can include one or more files.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderID

The binder identifier for the workspace or folder.

filename

The filename of the file you want to retrieve version information about.

return_value

A File Version Java object that contains information about the file versions.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getFolders

Returns a folder collection for a binder’s sub-folders.

Syntax

public FolderCollection binder_getFolders(String *accessToken*, long *binderId*,int *firstRecord*, int *maxRecords*);

Description

The `binder_getFolders` operation returns a folder collection, which contains information about the sub-folders of a specified binder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder for which you want information about its sub-folders.

firstRecord

The index of the first record whose folder information you want to obtain. The index is 0-based.

maxRecord

The maximum number of folders whose information should be returned. Specify -1 for unlimited.

return_value

A FolderCollection Java object containing information about the sub-folders.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getSubscription

Obtains subscription information about a binder.

Syntax

public Subscription binder_getSubscription(String *accessToken*, long *binderId*);

Description

The `binder_getSubscription` operation returns subscription information for a specified binder. When a user subscribes to a binder, that person receives e-mail notifications.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder for which you want subscription information.

return_value

A Subscription Java object containing subscription information.

Example

```
public static void checkBinderSubscriptions(long binderId) throws Exception {
...
Subscription subscription = setupSubscription(binderId);
stub.binder_setSubscription(null, binderId, subscription);
subscription = stub.binder_getSubscription(null, binderId);
validateSubscription(subscription);
}
```

This code is taken from the source code for the `teamingservice-client-with-stub.bat` file.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getTags

Obtains tags applied to a binder.

Syntax

public Tag[] binder_getTags(String *accessToken*, long *binderId*);

Description

The `binder_getTags` operation gets tag information for a specified binder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose tag information you want.

return_value

An array of Tag Java objects, each containing information about one of the tags applied to the binder.

Example

```
public static void checkTags(long binderId) throws Exception { ...
Tag[] tags = setupTags(binderId);
for (int i=0; i<tags.length; ++i) {stub.binder_setTag(null, tags[i]); }
tags = stub.binder_getTags(null, binderId);
validateTags(tags);
stub.binder_deleteTag(null, binderId, tags[0].getId());
```

This code is taken from the source code for the `teamingservice-client-with-stub.bat` file.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getTeamMembers

Obtains information about the members of a team assigned to a specified binder.

Syntax

public TeamMemberCollection binder_getTeamMembers(String *accessToken*, long *binderId*);

Description

The `binder_getTeamMembers` operation obtains information about the members of a team assigned to a specified binder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binder

The binder identifier for the workspace or folder for which you want information about team members.

return_value

A TeamMemberCollection Java object containing information about the members of a team assigned to the specified binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_getTrashEntries

Returns a trash collection for a binder.

Syntax

public TrashCollection binder_getTrashEntries(String *accessToken*, long *binderId*);

Description

The `binder_getTrashEntries` operation returns a trash collection, which contains information about the contents of the trash for a specified binder. The items in a binder's trash are all the sub-binders and entries that are in the trash.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder for which you want information about its trash contents.

firstRecord

The index of the first record whose trash information you want to obtain. The index is 0-based.

maxRecord

The maximum number of items whose information should be returned. Specify `-1` for unlimited.

return_value

A `TrashCollection` Java object containing information about the trash contents.

binder_indexBinder

Indexes a binder and its content.

Syntax

public void binder_indexBinder(String *accessToken*, long *binderId*);

Description

The `binder_indexBinder` operation indexes a workspace or folder (and its contents), optimizing the ability of Vibe to search its contents. This operation does not index sub-binders.

To index sub-binders, use the `binder_indexTree` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder that you want to index.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- binder_indexTree (page 83)

binder_indexTree

Indexes a binder’s sub-binders.

Syntax

public Long binder_indexTree(String *accessToken*, long *binderId*);

Description

The `binder_indexTree` operation indexes the specified workspace or folder, all sub-binders, and all content in all those binders.

If you want to index a binder and its contents without indexing sub-binders, use the `binder_indexBinder` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder that indicates the node where you want to begin indexing within the workspace hierarchy.

return_value

An array of integers, with each integer being the identifier of a binder successfully indexed.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- binder_indexBinder (page 82)

binder_modifyBinder

Modifies a binder.

Syntax

public void binder_modifyBinder(String *accessToken*, Binder *binder*);

Description

The `binder_modifyBinder` operation modifies a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binder

Data and methods for the Binder Java object, defined in the Vibe source code.

return_value

None.

Example

```
public static Binder modifyBinder(Binder binder) throws Exception{ ...
binder.setTitle(binder.getTitle() + " (Modified)");
binder.getDescription().setText(binder.getDescription().getText() + "
(Modified)");  stub.binder_modifyBinder(null, binder);
stub.binder_getBinder(null, binder.getId(), true);
System.out.println("ID of the modified binder: " + binder.getId());
return binder;
}
```

This code is taken from the source code for the `teamingservice-client-with-stub.bat` file.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_moveBinder

Moves a binder within the workspace tree hierarchy.

Syntax

public void binder_moveBinder(String *accessToken*, long *binderId*, long *newParentBinderId*);

Description

The `binder_moveBinder` operation moves either a workspace or folder within the workspace hierarchy.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder that you want to move.

newParentBinderId

The binder identifier of the binder under which you want *`binderId`* to appear as a sub-binder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_preDeleteBinder

Predeletes a binder by moving it to the trash.

Syntax

public void binder_preDeleteBinder(String *accessToken*, long *binderId*);

Description

The `binder_preDeleteBinder` operation moves a workspace or folder to the trash. All contained binders and entries are also moved to the trash.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder that you want to move to the trash.

return_value

None.

binder_removeFile

Removes a file from a binder.

Syntax

public void binder_removeFile(String *accessToken*, long *binderId*, String *fileName*);

Description

The `binder_removeFile` operation removes a file from a workspace or folder.

By default, workspaces do not contain files, but users can alter definitions by using the designers in the UI so that a custom workspace can include one or more files.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder from which you want to remove a file.

fileName

The file name of the file you want to remove from the binder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_restoreBinder

Undeletes a binder by removing it from the trash and restoring it to its previous location in the Vibe site.

Syntax

public void binder_restoreBinder(String *accessToken*, long *binderId*);

Description

The `binder_restoreBinder` operation undeletes a workspace or folder by removing it from the trash and restoring it to its previous location in the Vibe site.

Any containing binders that are in the trash are also undeleted.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder that you want to restore.

return_value

None.

binder_setDefinitions

Associates workflow definitions with entry definitions.

Syntax

public void binder_setDefinitions(String *accessToken*, long *binderId*, String[] *entryDefinitionIds*, String [] *workflowDefinitionIds*);

Description

The `binder_setDefinitions` operation associates entries within the specified binder with workflow processes. (Vibe associates identifiers in the first element of both arrays, the second element of both arrays, the third, and so on.)

When an entry is associated with a workflow process, creation of an entry of that type automatically places the entry into the initial state of the workflow process. By default, workspaces do not contain entries that can be associated with workflow processes. However, users can alter definitions by using the designers in the UI so that a custom workspace can include one or more files.

NOTE: This operation is an overwrite operation, setting all workflow associations for the folder; you cannot use repeated calls to this operation to set associations incrementally. Set all of the workflow associations for the folder with one call.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the folder or custom workspace within which you want entries associated with workflow processes.

entryDefinitionIds

An array of definition identfiers for each type of entry to which you want to associate a workflow process.

workflowDefinitionIds

An array of workflow identifiers in the order in which you want them applied to the entry-definition identifiers in *`entryDefinitionIds`*.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_setFunctionMembership

Applies access-control settings to a binder.

Syntax

public void binder_setFunctionMembership(String *accessToken*, long *binderId*, FunctionMembership[] *functionMemberships*);

Description

The `binder_setFunctionMembership` operation provides access-control settings for a folder or workspace. The term function is analogous to an access-control role in the UI.

NOTE: This operation is an overwrite operation, that sets all function memberships for the folder or workspace; you cannot use repeated calls to this operation to set memberships incrementally. Set all memberships for the workspace or folder with one call.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose access control you want to set.

functionMemberships

An array of FunctionMembership Java objects.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- binder_setFunctionMembershipInherited (page 89)

binder_setFunctionMembershipInherited

Establishes inheritance as the access-control mechanism for a folder or workspace.

Syntax

public void binder_setFunctionMembershipInherited(String *accessToken*, long *binderId*, boolean *inherit*);

Description

The `binder_setFunctionMembershipInherited` establishes whether a specified workspace or folder inherits access-control settings from its parent binder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder for which you want to establish the inheritance setting for access control.

inherit

A true or false value that establishes whether the binder inherits its access-control settings.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- binder_setFunctionMembership (page 88)

binder_setOwner

Establishes the owner of the binder.

Syntax

public long binder_setOwner(String *accessToken*, long *binderId*, long *userId*);

Description

The `binder_setOwner` operation establishes the specified user as the owner of a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose owner you want to establish.

userId

The identifier for the user whom you want to be the owner of the binder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_setSubscription

Establishes e-mail settings for a binder.

Syntax

public void binder_setSubscription(String *accessToken*, long *binderId*, Subscription *subscription*);

Description

The `binder_setSubscription` operation establishes subscription settings for a workspace or folder. When a user subscribes to a binder, that person receives e-mail notifications.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose subscription you want to set.

subscription

A Subscription Java object containing information used to establish e-mail notification settings for the specified binder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_setTag

Applies a tag for a binder.

Syntax

public void binder_setTag(String *accessToken*, Tag *tag*);

Description

The `binder_setTag` operation applies a tag to a workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

tag

A Tag Java object that contains information applying the tag to a workspace or folder.

return_value

None.

Example

```
public static void checkTags(long binderId) throws Exception{ ...
Tag[] tags = setupTags(binderId);
for (int i=0; i<tags.length; ++i) {stub.binder_setTag(null, tags[i]); }
tags = stub.binder_getTags(null, binderId);
validateTags(tags);
stub.binder_deleteTag(null, binderId, tags[0].getId());
```

This code is taken from the source code for the `teamingservice-client-with-stub.bat` file.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_setTeamMembers

Establishes members of a team for a binder.

Syntax

public void binder_setTeamMembers(String *accessToken*, long *binderId*, String[] *teamMembers*);

Description

The `binder_setTeamMembers` operation establishes members of the team for a specified workspace or folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binder

The binder identifier for the workspace or folder for which you want to establish team membership.

return_value

None.

teamMembers

Names of the team members.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

binder_testAccess

Tests if the calling user has the specified access right on each of the specified binders.

Syntax

public boolean[] binder_testAccess(String *accessToken*, String *workAreaOperationName*, long[] *binderIds*);

Description

The `binder_testAccess` operation tests if the calling user has the specified access right on each workspace or folder that is specified.

If a binder does not exist, the result for that specific binder is set to `<code>false</code>`. If the access right is an unknown value in Vibe, then the result for all binders is set to `<code>fasle</code>`.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

workAreaOperationName

The string name of a org.kablink.teaming.security.function.WorkAreaOperation instance. See the Java source file for the names.

binderIds

The ID of the binders against which to test the access.

return_value

An array of boolean values where each value represents whether the access test was successful or not for each binder.

binder_uploadFile

Uploads a file into a binder.

Syntax

public void binder_uploadFile(String *accessToken*, long *binder*Id,
String *formDataItemName*, String *fileName*);

Description

The `binder_uploadFile` operation performs an action equivalent to using the UI to upload a file to either a workspace or folder. You can attach only one file at a time; call this operation multiple times to attach more than one file to the binder.

By default, workspaces do not include attached files. However, users can use the designers to define workspaces that do include attached files.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder into which you want to upload a file.

formDataItemname

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined using the designers, you need to look up the `name` identifier for that form element.

If you are uploading to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

fileName

A string containing the filename of the file you want to upload to the binder.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

definition_getDefinitionAsXML

Obtains information about a definition.

Syntax

public String definition_getDefinitionAsXML(String *accessToken*, String *definitionId*);

Description

The `definition_getDefinitionAsXML` operation returns a string of XML containing information about a specified definition.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

definitionId

The definition identifier of the item about which you want information.

return_value

An XML string containing information about the definition. This XML is free form; it does not have a firm, established schema.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- definition_getDefinitionByName (page 95)
- definition_getDefinitions (page 95)
- definition_getDefinitionByName (page 95)
- definition_getLocalDefinitions (page 97)

definition_getDefinitionByName

Obtains information about a global definition.

Syntax

public DefinitionBrief definition_getDefinitionByName(String *accessToken*,
String *definitionName*);

Description

The `definition_getDefinitionByName` operation obtains information about a global definition by using the definition name. To get information about a local definition, use the `definition_getLocalDefinitionByName` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

definitionName

The descriptive word or phrase used to name the global definition.

return_value

A DefinitionBrief Java object containing information about the global definition.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- definition_getDefinitions (page 95)
- definition_getLocalDefinitionByName (page 96)

definition_getDefinitions

Obtains all global definitions in the installation.

Syntax

public DefinitionCollection definition_getDefinitions(String *accessToken*);

Description

The `definition_getDefinitions` operation obtains information about all global definitions in the installation. To get information about local definitions, use the `definition_getLocalDefinitions` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

A DefinitionCollection Java object containing information about all global definitions in the installation.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- definition_getLocalDefinitionByName (page 96)
- definition_getLocalDefinitions (page 97)

definition_getLocalDefinitionByName

Obtains information about a local definition.

Syntax

public DefinitionBrief definition_getLocalDefinitionByName(String *accessToken*, long *binderId*, String *name*, boolean *includeAncestors*);

Description

The `definition_getLocalDefinitionByname` operation obtains information about a local definition by using a name. To get information about a global definition, use the `definition_getDefinitionByname` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose local definition you want.

name

The word or phrase used to name the local definition.

includeAncestors

A Boolean value that indicates whether Vibe should check local definitions inherited from ancestor workspaces and folders, which are located higher in the hierarchy than the specified binder. If you specify false, Vibe checks only the local definitions created within the specified binder.

return_value

A DefinitionBrief Java object containing information about the definition that matches *name*.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- definition_getLocalDefinitionByName (page 96)
- definition_getLocalDefinitions (page 97)

definition_getLocalDefinitions

Obtains information about all local definitions.

Syntax

public DefinitionCollection definition_getLocalDefinitions(String *accessToken*, long *binderId*, boolean *includeAncestors*);

Description

The `definition_getLocalDefinitions` operation obtains information about the local definitions for a specified binder. If you want information about all global definitions in the installation, use the `definition_getDefinitions` operation.

If you want to add a fully configured binder, use `template_addBinder` instead.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the workspace or folder whose local definitions you want.

includeAncestors

A Boolean value that indicates whether Vibe should include local definitions inherited from ancestor workspaces and folders, which are located higher in the hierarchy than the specified binder. If you specify false, Vibe includes only the local definitions created within the specified binder.

return_value

A DefinitionCollection Java object that contains information about the binder's local definitions.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)
- definition_getDefinitions (page 95)
- definition_getLocalDefinitionByName (page 96)

folder_addEntry

Adds an entry to a folder.

Syntax

public long folder_addEntry(String *accessToken*, FolderEntry *entry*, String *attachedFileName*);

Description

The `folder_addEntry` operation adds an entry to a folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entry

A FolderEntry Java object containing information that Vibe uses to create the new entry.

attachedFileName

(Optional) A string containing the filename of a file to attach to the new entry. If you are not attaching a file, specify the null value for this argument.

return_value

The entry identifier of the newly created entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)

folder_addEntryWorkflow

Initiates a workflow process for a folder entry.

Syntax

public void folder_addEntryWorkflow(String *accessToken*, long *entryId*, String *workflowDefinitionId*);

Description

The `folder_addEntryWorkflow` operation initiates a workflow process for a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the folder entry with which you want to initiate a workflow process.

workflowDefinitionId

The definition identifier of the workflow process that you want to initiate for the specified folder entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_addMicroBlog

Adds a micro-blog entry to a folder.

Syntax

public long folder_addMicroBlog(String *accessToken*, string *text*);

Description

The `folder_addMicroBlog` operation adds a micro-blog entry to a folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

text

A string containing the text of the micro-blog to create.

return_value

The entry identifier of the newly created micro-blog entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_addReply

Adds a comment to a folder entry.

Syntax

public long folder_addReply(String *accessToken*, long *parentEntryId*, FolderEntry *reply*, String *attachedFileName*);

Description

The `folder_addReply` operation adds a comment to a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

parentEntryId

The entry identifier of the entry or comment that is to be the parent of the comment you are adding.

reply

A FolderEntry Java object containing information that yyyy uses to create the new comment.

attachedFileName

The filename of a file you are attaching to the comment. If you are not attaching a file, specify the null value for this argument.

return_value

The entry identifier of the newly created comment.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_copyEntry

Copies a folder entry.

Syntax

public long folder_copyEntry(String *accessToken*, long *entryId*, long *parentFolderId*);

Description

The `folder_copyEntry` operation copies a folder entry.

NOTE: This operation does not copy workflow information for an entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the folder entry you want to copy.

parentFolderId

The folder identifier of the folder you want to contain the copied entry.

return_value

The entry identifier of the new entry created by copying the existing entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_deleteEntry

Deletes a folder entry.

Syntax

public void folder_deleteEntry(String *accessToken*, long *entryId*);

Description

The `folder_deleteEntry` operation deletes a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the folder entry you want to delete.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_deleteEntryTag

Removes a tag from a folder entry.

Syntax

public void folder_deleteEntryTag(String *accessToken*, long *entryId*, String *tagId*);

Description

The `folder_deleteEntryTag` operation removes a tag from a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry from which you want to remove a tag.

tagId

A string identifying the tag you want to remove from the entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_deleteEntryWorkflow

Removes a workflow from an entry.

Syntax

public void folder_deleteEntryWorkflow(String *accessToken*, long *entryId*,
String *workflowDefinitionId*);

Description

The `folder_deleteEntryWorkflow` operation removes a workflow process from a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry for which you want to remove a workflow process.

workflowDefinitionId

A string containing the definition identifier for the workflow process you want to remove from the entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_getEntries

Obtains information about the entries within a specified folder.

Syntax

public FolderEntryCollection folder_getEntries(String *accessToken*, long *binderID*, int *firstRecord*,
int *maxRecords*);

Description

The `folder_getEntries` operation obtains information about the entries contained in a folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the folder containing the entries for which you want information.

firstRecord

The index of the first record whose information you want to obtain. The index is 0-based.

maxRecords

The maximum number of entries whose information should be returned. Specify -1 for unlimited.

return_value

A FolderEntryCollection Java object containing information about the entries contained within the folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_getEntry

Accepts an entry identifier to get information about an entry in a folder.

Syntax

public FolderEntry folder_getEntry(String *accessToken*, long *entryId*,
boolean *includeAttachments*);

Description

The `folder_getEntry` operation obtains information about an entry in a folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry about which you want information.

includeAttachments

A Boolean value that indicates whether you want Vibe to return the entry's attachments. The client program is responsible for placement of attachment files on its local system.

return_value

A FolderEntry Java object that contains information about the specified entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)

folder_getEntryByFileName

Accepts a filename to get information about an entry.

Syntax

public FolderEntry folder_getEntryByFileName(String *accessToken*, long *binderId*, String *fileName*, boolean *includeAttachments*);

Description

The `folder_getEntryByFileName` operation obtains information about an entry in a folder by using the entry's file name.

Although this operation is most useful for Files folders, it works for any folder that requires that all filenames within the folder to be unique.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The binder identifier for the folder containing the entry for which you want information.

fileName

The name of the file that corresponds with the entry for which you want information.

includeAttachment

A Boolean value that indicates whether you want Vibe to return the entry's attachments. The client program is responsible for placement of attachment files on its local system.

return_value

A FolderEntry Java object that contains information about the specified entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_getEntryTags

Obtains information about an entry’s tags.

Syntax

public Tag[] folder_getEntryTags(String *accessToken*, long *entryId*);

Description

The `folder_getEntryTags` operation gets information about each of the tags applied to a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry about whose tags you want information.

return_value

An array of Tag Java objects, where each object contains information about one tag applied to the entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_getFileVersions

Returns information about the versions of a file.

Syntax

public void folder_getFileVersions(String *accessToken*, long *entryId*, String *fileName*);

Description

The `folder_getFileVersions` operation retrieves information about the versions of a file associated with an entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry.

fileName

The filename of the file you want to retrieve version information about.

return_value

A File Versions Java object containing information about the file versions.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_getSubscription

Obtains subscription information for a specified folder.

Syntax

public Subscription folder_getSubscription(String *accessToken*, long *entryId*);

Description

The `folder_getSubscription` operation gets information about the e-mail notification settings for a specified folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier for the entry whose subscription information you want.

return_value

A Subscription Java object that contains information about e-mail notification settings for the specified folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_modifyEntry

Modifies an entry in a folder.

Syntax

public void folder_modifyEntry(String *accessToken*, FolderEntry *entry*);

Description

The `folder_modifyEntry` operation modifies the contents of a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entry

A FolderEntry Java object containing the information to apply to the existing folder entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_modifyWorkflowState

Changes the workflow state of an entry.

Syntax

public void folder_modifyWorkflowState(String *accessToken*, long *entryId*, long *StateId*, String *toState*);

Description

The `folder_modifyWorkflowState` operation changes the workflow state of a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry whose workflow state you want to change.

stateID

The token ID of the current workflow state from which you want the entry to transition to the new state.

toState

A string identifying your desired workflow state.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_moveEntry

Moves an entry within the folder-tree hierarchy.

Syntax

public void folder_moveEntry(String *accessToken*, long *entryId*, long *parentId*);

Description

The `folder_moveEntry` operation moves an entry to be under a new parent within the folder hierarchy.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry you want to move.

parentId

The identifier of the folder that is to be the new parent of the specified entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_preDeleteEntry

Predeletes an entry by moving it to the trash.

Syntax

public void folder_preDeleteEntry(String *accessToken*, long *entryId*);

Description

The `folder_preDeleteEntry` operation moves an entry to the trash.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry that you want to move to the trash.

return_value

None.

folder_removeFile

Removes a file attachment from an entry.

Syntax

public void folder_removeFile(String *accessToken*, long *entryId*, String *fileName*);

Description

The `folder_removeFile` operation removes a file attachment from an entry in a folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry that includes the attachment you want to remove.

fileName

A string containing the filename of the attachment you want to remove from the entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_reserveEntry

Reserves an entry.

Syntax

public void folder_reserveEntry(String *accessToken*, long *entryId*);

Description

The `folder_reserveEntry` operation reserves an entry in a folder, preventing others from modifying it.

Users reserve and release an entry in the UI using the *Reserve* and *Unreserve* menu items.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry you want to reserve.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_restoreEntry

Undeletes an entry by removing it from the trash and restoring it to its previous location in the Vibe site.

Syntax

public void folder_restoreEntry(String *accessToken*, long *entryId*);

Description

The `folder_restoreEntry` operation undeletes an entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The entry identifier of the entry that you want to restore.

return_value

None.

folder_setEntryTag

Applies a tag to a folder entry.

Syntax

public void folder_setEntryTag(String *accessToken*, Tag *tag*);

Description

The `folder_setEntryTag` operation applies a tag to a folder entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

tag

A Tag Java object containing information about the tag you want to apply.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_setRating

Sets a rating for a folder entry.

Syntax

public void folder_setRating(String *accessToken*, long *entryId*, long *value*);

Description

The `folder_setRating` operation applies a “star” rating to an entry.

In the UI, entries can have ratings that range from a lowest value of 1 star to the highest value of 5 stars.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry for which you want to apply a rating.

ratingValue

An integer indicating how many stars you want to set as the rating.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_setSubscription

Establishes subscription settings for an entry.

Syntax

public void folder_setSubscription(String *accessToken*, long *entryId*, Subscription *subscription*);

Description

The `folder_setSubscription` operation establishes settings for e-mail notifications for a specified entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry for which you want to set subscription information.

subscription

A Subscription Java object that contains subscription information to be applied to the specified entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_setWorkflowResponse

Applies an answer to a workflow question for a specified entry.

Syntax

public void folder_setWorkflowResponse(String *accessToken*, long *entryId*,
long *stateId*, String *question*, String *response*);

Description

The `folder_setWorkflowResponse` operation establishes an answer for a workflow question for a specified entry.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry that is currently in the workflow state within which you want to apply an answer to a question.

stateId

The token ID of the current workflow state that defines the question that you want to affect.

question

A string identifying the question that you are providing an answer to.

response

A string identifying the response you want to apply to the workflow question.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_synchronizeMirroredFolder

Synchronizes a mirrored folder with its source folder.

Syntax

public void folder_synchronizeMirroredFolder(String *accessToken*, long *binderId*);

Description

The `folder_synchronizeMirroredFolder` operation synchronizes a mirrored folder with its source folder.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the mirrored folder that you want to synchronize.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_unreserveEntry

Releases a locked entry.

Syntax

public void folder_unreserveEntry(String *accessToken*, long *entryId*);

Description

The `folder_unreserveEntry` operation releases a locked entry.

Users reserve and release an entry in the UI by using the *Reserve* and *Unreserve* menu items.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry that you want to release from its lock.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

folder_uploadFile

Uploads a file as an attachment to an entry.

Syntax

public void file_uploadFile(String *accessToken*, long *entryId*, String *formDataItemName*, String *fileName*);

Description

The `file_uploadFile` operation uploads a file as an attachment to an entry. You can attach only one file at a time; call this operation multiple times to attach more than one file to the entry.

Because transferring files across the Internet can be time-consuming, you can upload files that have already been moved to a staging area on the Vibe server by using the `folder_uploadFileStaged` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

The identifier of the entry that is to include the new attached file.

formDataItemName

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined by using the designers, you need to look up the `name` identifier for that form element.

If you are uploading to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

fileName

A string containing the filename of the file you want to attach to the entry.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- folder_uploadFileStaged (page 117)

folder_uploadFileStaged

Locates a locally stored file and attaches it to an entry.

Syntax

public void file_uploadFileStaged(String *accessToken*, long *entryId*, String *formDataItemName*, String *fileName*, String *stagedFileRelativePath*);

Description

As a way to streamline the transfer of files, the `file_uploadFileStaged` operation accesses a file that has been copied locally to the Vibe server, avoiding transferring them over the Internet. The operation then attaches the file to a folder entry in Vibe. In order for the Web services client to utilize this operation, the Vibe administrator must first configure the server to allow this operation by specifying `staging.upload.files.enable` and `staging.upload.files.rootpath` configuration settings in `ssf-ext.properties` file. Because it involves Vibe administrator access to the server environment, this operation is reserved only for major migration projects where individual file uploads through the HTTP protocol do not meet the performance requirements of the project.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entryId

An identifier for the entry to which you want to attach a file.

formDataItemName

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined by using the designers, you need to look up the `name` identifier for that form element.

If you are uploading to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

fileName

A string containing the filename of the file you want to attach to the entry.

stagedFileRelativePath

A pathname of the file relative to the staging area on the server side. On the Vibe server, the staging directory is designated by the value of the `staging.uploads.files.rootpath` configuration setting. This relative pathname is resolved against the staging directory of the Vibe server to identify the input file.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- folder_uploadFile (page 116)

ical_uploadCalendarEntriesWithXML

Adds a calendar entry to a folder.

Syntax

public void ical_uploadCalendarEntriesWithXML(String *accessToken*, long *folderId*, String *iCalDataAsXML*);

Description

The `ical_uploadCalendarEntriesWithXML` adds a calendar entry using iCal information in an XML string.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

folderId

The identifier of the folder where you want to add a calendar entry.

iCalDataAsXML

A string containing XML formatted calendar data (`<doc><entry>iCal data</entry>...</doc>`).

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

ldap_synchAll

Synchronizes all users with the current information that is in LDAP.

Syntax

public void ldap_synchAll(String *accessToken*);

Description

The `ldap_synchAll` operation synchronizes all users with the current information that is in LDAP.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

ldap_synchUser

Synchronizes one user with the latest information in LDAP for that person.

Syntax

public void ldap_synchUser(String *accessToken*, long *userId*);

Description

The `ldap_synchUser` operation synchronizes one user with the latest information in LDAP for that person.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

userId

The identifier of the user whose information you want synchronized with that person's LDAP data.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)

license_getExternalUsers

Obtains a count of external users.

Syntax

public long license_getExternalUsers(String *accessToken*);

Description

The `license_getExternalUsers` operation obtains a count of legal external users for the current license.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

An integer indicating the number of allowed external users.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)

license_getRegisteredUsers

Obtains a count of registered Vibe users.

Syntax

public long license_getRegisteredUsers(String *accessToken*);

Description

The `license_getRegisteredUsers` operation obtains a count of the current number of registered users on the system.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

An integer that is the count of users currently registered on the system.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

license_updateLicense

Updates the Vibe license.

Syntax

public void license_updateLicense(String *accessToken*);

Description

The `license_updateLicense` operation updates the Vibe license.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

migration_addBinder

Accepts a Java object to add a binder, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addBinder(String *accessToken*, Binder *binder*);

Description

The `migration_addBinder` operation adds either a workspace or folder to the hierarchy, allowing you to specify SiteScape Forum data (such as the person who created the workspace or folder in Forum, the Forum creation date, the user who last modified the workspace or folder in Forum, and the date of the last modification in Forum).

If you prefer to use XML to specify data, use the `migration_addBinderWithXML` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binder

Data and methods for the Java Binder object, defined in the Vibe source code. Edit the information in the Binder object to reflect the Forum values.

return_value

The identifier of the newly created binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addBinderWithXML (page 122)

migration_addBinderWithXML

Accepts XML to add a binder, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addBinderWithXML(String *accessToken*, long *parentId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*);

Description

The `migration_addBinderWithXML` operation adds either a workspace or folder to the hierarchy, allowing you to specify SiteScape Forum data (such as the person who created the workspace or folder in Forum, the Forum creation date, the user who last modified the workspace or folder in Forum, and the date of the last modification in Forum).

If you prefer to use a Java object to specify data, use the `migration_addBinder` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

parentId

The identifier of the workspace or folder that is to contain the new binder.

definitionID

A string that identifies the definition used to create the new binder.

inputDataAsXML

An XML string that provides the data needed to construct the workspace or folder.

creator

A string containing the username of the person who created the workspace or folder in Forum.

creationDate

A Calendar Java object that contains the creation date of the workspace or folder in Forum.

modifier

A string containing the username of the person who last modified the workspace or folder in Forum.

modificationDate

A Calendar Java object that contains the modification date of the workspace or folder in Forum.

return_value

The identifier of the newly created binder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addBinder (page 122)

migration_addEntryWorkflow

Associates an entry with a workflow process, allowing preservation of SiteScape Forum data.

Syntax

public void migration_addEntryWorkflow(String *accessToken*, long *binderId*, long *entryId*, String *definitionId*, String *startState*, String *modifier*, Calendar *modificationDate*);

Description

The `migration_addEntryWorkflow` operation associates an entry with a workflow process, while preserving values from a SiteScape Forum installation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the folder containing the entry with which you want to associate a workflow process.

entryId

The identifier of the entry with which you want to associate the workflow process.

definitionId

A string containing the definition identifier for the workflow process you want to associate with the entry.

startState

A string containing the name of the state of the entry as it was last set in the Forum installation.

modifier

A string containing the username of the person who last modified the workflow state in the Forum installation.

modificationDate

A Calender Java object that contains the date that the workflow state was last modified in the Forum installation.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

migration_addFolderEntry

Accepts a Java object to add an entry to a folder, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addFolderEntry(String *accessToken*, FolderEntry *entry*, boolean *subscribe*);

Description

The `migration_addFolderEntry` operation adds an entry to a folder.

If you prefer to use an XML string to create the new entry, use the `migration_addFolderEntryWithXML` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

entry

A FolderEntry Java object that contains information used to create the new entry, including information from the entry in the Forum installation.

subscribe

A Boolean value that implements the Forum *notify me when someone replies to this entry* feature by establishing a subscription for the entry owner.

return_value

The identifier of the newly created entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addFolderEntryWithXML (page 125)

migration_addFolderEntryWithXML

Accepts XML to add an entry to a folder, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addFolderEntryWithXML(String *accessToken*, long *binderId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*, boolean *subscribe*);

Description

The `migration_addFolderEntry` operation adds an entry to a folder, allowing you to preserve data from the entry as it last existed in an installation of SiteScape Forum.

If you prefer to create the entry by using a Java object, use the `migrate_addFolderEntry` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the folder that is to contain the new entry.

definitionId

A string containing the definition identifier for the new entry.

inputDataAsXML

A string containing the XML elements used to construct the new entry.

creator

A string containing the username of the person who created the entry in the Forum installation.

creationDate

A Calendar Java object containing the date the entry was created in the Forum installation.

modifier

A string containing the username of the person who last modified the entry in the Forum installation.

modificationDate

A Calendar Java object containing the date the entry was last modified in the Forum installation.

subscribe

A boolean value that implements the Forum feature “notify me when someone replies to this entry” by establishing a subscription for the entry owner.

return_value

The identifier of the binder for the newly created entry.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addFolderEntry (page 124)

migration_addReply

Accepts a Java object to add a comment, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addReply(String *accessToken*, long *parentEntryId*, FolderEntry *reply*);

Description

The `binder_addReply` operation adds a comment to an entry or a reply, and allows you to preserve data from the reply as it last appeared in a Forum installation.

If you prefer to add the comment by using XML, use the `migrate_addReplyWithXML` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

parentEntryId

The identifier of the entry or comment that is the parent of the comment you want to create.

reply

A FolderEntry Java object that contains information used to construct the new comment, including data reflecting the reply as it last appeared in the Forum installation.

return_value

The identifier of the newly created comment.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addReplyWithXML (page 127)

migration_addReplyWithXML

Accepts XML to add a comment, allowing preservation of SiteScape Forum data.

Syntax

public long migration_addReplyWithXML(String *accessToken*, long *binderId*, long *parentId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*);

Description

The `migration_addReplyWithXML` operation adds a comment to an entry or to another comment, allowing you to preserve data from the reply as it last appeared in the SiteScape Forum installation.

If you prefer to add the comment by using a Java object, use the `migration_addReply` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the folder that contains the entry to which you want to add the comment.

parentId

The identifier of the entry or comment that is to be the parent of the newly created comment.

definitionId

A string containing the definition identifier for the comment you want to create.

inputDataAsXML

An XML string whose elements are used to create the new comment.

creator

A string containing the username of the person who created the reply in the Forum installation.

creationDate

A Calendar Java object containing the date that the reply was created in the Forum installation.

modifier

A string containing the username of the person who last modified the reply in the Forum installation.

modificationDate

A Calendar Java object that contains the date that the reply was last modified in the Forum installation.

return_value

The identifier of the newly created comment.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_addReply (page 126)

migration_uploadFolderFile

Uploads an entry attachment, allowing preservation of SiteScape Forum data.

Syntax

public void migration_uploadFolderFile(String *accessToken*, long *binderId*, long *entryId*, String *formDataItemName*, String *fileName*, String *modifier*, Calendar *modificationDate*);

Description

The `migration_uploadFolderFile` operation attaches a file to an entry, allowing you to preserve data from the attachment as it last appeared in a SiteScape Forum installation. You can attach only one file at a time; call this operation multiple times to attach more than one file to the entry.

Because moving files across the Internet can be time-consuming, you can create attachments from Forum files that have already been copied to a staging area on the Vibe server by using the `migration_uploadFolderFileStaged` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the folder containing the entry to which you want to attach a file.

entryId

The identifier of the entry to which you want to attach a file.

formDataItemName

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined by using the designers, you need to look up the `name` identifier for that form element.

If you are uploading to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

fileName

A string containing the name of the file you want to upload.

modifier

A string containing the username of the last person in the Forum installation to modify the file.

modificationDate

A Calendar Java object containing the date that the file was last modified in the Forum installation.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_uploadFolderFileStaged (page 130)

migration_uploadFolderFileStaged

Uploads a local copy of an entry attachment, allowing preservation of SiteScape Forum data.

Syntax

public void migration_uploadFolderFileStaged(String *accessToken*, long *binderId*, long *entryId*, String *formDataItemName*, String *fileName*, String *stagedFileRelativePath*, String *modifier*, Calendar *modificationDate*);

Description

The `migration_uploadFolderFileStaged` operation accesses a file that has been copied locally to the Vibe server as a way to streamline the transfer of files, avoiding transferring them over the Internet. The operation then attaches the file to a folder entry in Vibe.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the binder containing the entry to which you want to attach a file.

entryId

The identifier of the entry to which you want to attach a file.

formDataItemName

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined by using the designers, you need to look up the `name` identifier for that form element.

If you are uploading to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

fileName

A string containing the name of the file you want to attach to an entry.

stagedFileRelativePath

A pathname of the file relative to the staging area on the server side. On the Vibe server, the staging directory is designated by the value of the `staging.upload.files.rootpath` configuration setting. This relative pathname is resolved against the staging directory of the Vibe server to identify the input file.

Although the files can be present in any folder structure within the staging area, one streamlined way to approach this task is to unzip the Forum hidden directory into the staging area. Then, use this parameter to specify the relative path through the hidden folder structure to the location of the file to be attached to the entry in Vibe.

modifier

A string containing the username of the person who last modified the file in the Forum installation.

modificationDate

A Calendar Java object that contains the date that the file was last modified in the Forum installation.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- migration_uploadFolderFile (page 128)

profile_addGroup

Adds a group.

Syntax

public long profile_addGroup(String *accessToken*, Group *group*);

Description

The `profile_addGroup` operation adds a new group to Vibe.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

group

A Group Java object containing information needed to create the new group in Vibe.

return_value

The identifier of the newly created group.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_addGroupMember

Adds a user to a group.

Syntax

public void profile_addGroupMember(String *accessToken*, String *groupName*, String *userName*);

Description

The `profile_addGroupMember` operation adds a user to a group.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

groupName

A string containing the name of the group.

userName

A string containing the name of the user to be added to the group.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_addUser

Adds a user profile.

Syntax

public long profile_addUser(String *accessToken*, User *user*);

Description

The `profile_addUser` operation adds a profile for a new Vibe user.

After you add a user profile, you can add a user workspace for the new user by using the `profile_addUserWorkspace` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

user

A User Java object containing the information needed to create a new user.

return_value

The identifier of the newly created user.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_addUserWorkspace (page 133)

profile_addUserWorkspace

Adds a user workspace for an existing user.

Syntax

public long profile_addUserWorkspace(String *accessToken*, long *userId*);

Description

The `profile_addUserWorkspace` operation adds a user workspace for an existing user.

To create a new user before using this operation, use the `profile_addUser` operation, which creates a profile for a new user.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

userId

The identifier of the user for whom you want to create a user workspace.

return_value

The binder identifier of the newly created user workspace.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_addUser (page 132)

profile_deletePrincipal

Removes a group or user.

Syntax

public void profile_deletePrincipal(String *accessToken*, long *principalId*,
boolean *deleteWorkspace*);

Description

The `profile_deletePrincipal` operation removes a group or user.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

principalId

The identifier of the group or user you want to delete.

deleteWorkspace

When you delete a user, this Boolean value indicates whether Vibe should delete the corresponding user workspace.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_getFileVersions

Returns information about the versions of a file.

Syntax

public void profile_getFileVersions(String *accessToken*, long *principalId*, string *fileName*);

Description

The `profile_getFileVersions` operation retrieves information about the versions of a file associated with a user or group.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

principalId

The identifier for the principal (a user or group).

fileName

The filename of the file you want to retrieve version information about.

return_value

A File Version Java object containing information about the file versions.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_getGroup

Accepts a group identifier to obtain the title and the description of the group.

Syntax

public Group profile_getGroup(String *accessToken*, long *groupId*, boolean *includeAttachments*);

Description

The `profile_getGroup` operation obtains the title and the description of the group.

If you want to get information about the members of a group, use the `profile_getGroupMembers` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

groupId

The identifier of the group about which you want information.

includeAttachments

A Boolean value that indicates whether you want files that are attached to the group.

By default, you cannot attach files to a group. However, a site administrator can use the designers in the UI to customize a group to be able to include files.

return_value

A Group Java object containing information about all of the group members.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_getGroupByName (page 136)
- profile_getGroupMembers (page 137)

profile_getGroupByName

Accepts a group name to obtain the title and the description of the group.

Syntax

public Group profile_getGroupByName(String *accessToken*, String *groupName*, boolean *includeAttachments*)

Description

The `profile_getGroupByName` operation obtains the title and the description of a group.

If you want to get information about the members of a group, use the `profile_getGroupMembers` operation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

groupName

A string containing the name of the group.

includeAttachments

A Boolean value that indicates whether you want files attached to the group.

By default, you cannot attach files to a group. However, a site administrator can use the designers in the UI to customize a group to be able to include files.

return_value

A Group Java object containing information about all of the group members.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_getGroup (page 135)
- profile_getGroupMembers (page 137)

profile_getGroupMembers

Obtains information about the members of a group.

Syntax

public PrincipalCollection profile_getGroupMembers(String *accessToken*, String *groupName*);

Description

The `profile_getGroupMembers` operation obtains information about members of a group.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

groupName

A string containing the name of the group whose members you want information about.

return_value

A PrincipalCollection Java object containing information about the members of the specified group.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_getPrincipals

Gets information for users and groups in the installation.

Syntax

public PrincipalCollection profile_getPrincipals(String *accessToken*, int *firstRecord*, int *maxRecords*);

Description

The `profile_getPrincipals` operation gets information for users and groups in the installation. Because the set of information is potentially very large, you can use successive calls to this operation to receive manageable subsets of information for each call.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

firstRecord

The number of the record (information about one user or group) to begin returning. Use this parameter to page the returned list of principals.

The number of the first record `0`.

maxRecords

The largest number of records you want returned in this call. For an unlimited number specify -1.

return_value

A PrincipalCollection Java objection containing information about the set of users and groups you requested.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_getUser

Accepts a user identifier to get information about a user.

Syntax

public User profile_getUser(String *accessToken*, long *userId*, boolean *includeAttachments*);

Description

The `profile_getUser` operation accepts a user identifier and returns information about a Vibe user.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

userId

The identifier of the user about whom you want information.

includeAttachments

A Boolean value that specifies whether Vibe should return attachments to the user's profile.

By default, the only attached files are the users' pictures. However, the site administrator can customize the profile to include other files by using the designer tools in the UI.

return_value

A `User` Java object that contains information about the requested user.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, "Working with Java Objects," on page 17)
- profile_getUserByName (page 139)

profile_getUserByName

Accepts a username to get information about a user.

Syntax

public User profile_getUserByName(String *accessToken*, String *userName*, boolean *includeAttachments*);

Description

The `profile_getUserByName` operation accepts a username as a parameter and returns information about a Vibe user.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

userName

A string containing the username of the user for whom you want information.

includeAttachments

A Boolean value that indicates whether Vibe should return attached files.

By default, the only attached files are the users' pictures. However, the site administrator can customize the profile to include other files by using the designer tools in the UI.

return_value

A User Java object containing information about the requested user.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_getUser (page 138)

profile_getUsers

Obtains information for users in the installation.

Syntax

public UserCollection profile_getUsers(String *accessToken*, boolean *captive*, int *firstRecord*, int *maxRecords*);

Description

The `profile_getUsers` operation gets information for users in the installation. Because the set of information is potentially very large, you can use successive calls to this operation to receive manageable subsets of information for each call.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

captive

Set this to `true` if you want the permalink URL returned for each user workspace to represent captive mode. When a user workspace is viewed in captive mode, the master heading and the sidebar are removed from the display, which allows the page to fit better in a small screen. The default is false.

firstRecord

The number of the record to begin returning. Use this parameter to page the returned list of users.

The number of the first record is 0.

maxRecord

The largest number of records you want to return in this call. Specify -1 for unlimited.

return_value

A UserCollection Java object that contains information about the entries contained within the folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- profile_getUserByName (page 139)

profile_getUserTeams

Obtains information about all teams that the specified user is a member of.

Syntax

public TeamCollection search_getUserTeams(String *accessToken*, long *userId*);

Description

The `search_getUsersTeams` operation obtains information about all teams that the user is a member of.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

userId

The identifier of the user about whom you want information.

return_value

A UserCollection Java object that contains information about the entries contained within the folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_modifyGroup

Modifies a group.

Syntax

public void profile_modifyGroup(String *accessToken*, Group *group*);

Description

The `profile_modifyGroup` operation modifies information associated with a group.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

group

A Group Java object containing modified information about a group.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_modifyUser

Modifies a user.

Syntax

public void profile_modifyUser(String *accessToken*, User *user*);

Description

The `profile_modifyUser` operation modifies information associated with a user.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

user

A User Java object containing modified information about a user.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_removeFile

Removes a file from the user profile.

Syntax

public void profile_removeFile(String *accessToken*, long *principalId*, String *fileName*);

Description

The `profile_removeFile` operation removes a file from a user profile.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

principalId

The identifier for the principal (by default, a user) from which you want to remove a file.

By default, only user profiles contain files. However, it is possible for site administrators to customize groups by using the designer tools in the UI.

fileName

A string containing the name of the file you want to remove.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_removeGroupMember

Removes a user from a group.

Syntax

public void profile_removeGroupMember(String *accessToken*, String *groupName*, String *userName*);

Description

The `profile_removeGroupMember` operation removes a user from membership in a group.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

groupName

A string containing the name of the group from which you want to remove a member.

userName

A string containing the name of the user you want to remove from the specified group.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

profile_uploadFile

Uploads a file as an attachment to a user or group.

Syntax

public void profile_uploadFile(String *accessToken*, long *principalID*, String *formDataItemName*, String *fileName*);

Description

The `profile_uploadFile` operation performs an action similar to using the user interface to upload a picture to user profiles. Files are attached one at a time; call this operation multiple times to attach more than one file to the binder.

By default, only user profiles contain files. However, it is possible for site administrators to customize groups by using the designer tools in the user interface,

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

principalId

The identifier for the user or group to which you want to attach a file.

formDataItemName

A string containing the internal identifier for the part of the principal entry that contains attached files. This identifier maps the name attribute of an `input` HTML tag on a form to data in the Vibe database; a `hidden` HTML tag communicates this file mapping to the server.

The name value for the standard entry element containing attached files is `ss_attachFile`. To upload a file into the custom forms element you defined by using the designer, you need to look up the name identifier for that form element.

To upload a picture for a user profile, specify `picture` as an argument to this parameter to make this attachment one of the pictures associated with the user profile.

fileName

A string containing the filename of the file you want to upload to the principal.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

search_getFolderEntries

Obtains information about the entries that match the specified search query.

Syntax

public String search_getFolderEntries(String *accessToken*, String *query*, int *offset*, int *maxResults*);

Description

The `search_getFolderEntries` operation obtains information about the entries matching the specified search query. Because the list of each result can be lengthy, this operation lets you make multiple calls, receiving a subset of the search results each time.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

query

A search query represented in XML.

offset

An integer indicating at which result you want to begin receiving information. The first result is numbered 0.

maxResults

An integer indicating the number of results you want returned. The value of -1 indicates unlimited.

return_value

A FolderEntryCollection Java object containing information about the entries contained within the folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

search_getTeams

Obtains information about the teams that the calling user is a member of.

Syntax

public TeamCollection search_getTeams(String *accessToken*);

Description

The `search_getTeams` operation obtains information about the teams that the calling member is a user of.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

A TeamCollection Java object that contains information about the teams that the calling user is a member of.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

search_getWorkspaceTreeAsXML

Obtains information needed to construct the Vibe workspace and folder tree.

Syntax

public String search_getWorkspaceTreeAsXML(String *accessToken*, long *binderId*, int *levels*, String *page*);

Description

The `search_getWorkspaceTreeAsXML` operation obtains information needed to construct the Vibe workspace and folder tree.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

binderId

The identifier of the binder whose descendants you want to include in the workspace and folder tree information.

The top workspace in the Vibe tree has a binder identifier of `1`.

levels

The number of hierarchical levels down from the node specified by *`binderId`* that you want to include in the returned information. The value `-1` indicates that you want all subsequent levels.

page

A parameter used to expand pages of binders. When you specify a valid page identifier, Vibe expands the page by the number of levels indicated in the `levels` parameter.

If you do not want to use this call expand pages, pass `null` as this parameter.

See Section 1.7.6, “Binder Pages and search_getWorkspaceTreeAsXML,” on page 20 for more detailed information about working with pages.

return_value

A string containing XML elements needed to construct each node within the requested levels of the workspace hierarchy.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

search_search

Returns XML for results of a search query.

Syntax

public String search_search(String *accessToken*, String *query*, int *offset*, int *maxResults*);

Description

The `search_search` operation returns XML for the results of a search query represented in XML. Because the list of each results can be lengthy, this operation is designed so that you can make multiple calls, receiving a subset of the search results each time.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

query

A search query represented in XML.

offset

An integer indicating at which result you want to begin receiving information. The first result is numbered `0`.

maxResults

An integer indicating the number of results you want returned.

return_value

A string of XML containing information about the search results that match your specified criteria.

Example

The following input query string in XML matches all users whose first name begins with the letter `J` or the last name is `Smith`.

```
<QUERY>
     <AND>
          <FIELD fieldname="_entityType" exactphrase="true">
               <TERMS>user</TERMS>
          </FIELD>
          <FIELD fieldname="_docType" exactphrase="true">
               <TERMS>entry</TERMS>
          </FIELD>
          <OR>
               <FIELD fieldname="lastName" exactphrase="true">
                    <TERMS>Smith</TERMS>
               </FIELD>
               <FIELD fieldname="firstName" exactphrase="false">
                    <TERMS>J*</TERMS>
               </FIELD>
          </OR>
     </AND>
</QUERY>
```

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

template_addBinder

Adds a fully configured workspace or folder to the workspace hierarchy.

Syntax

public long template_addBinder(String *accessToken*, long *parentId*, long *binderConfigId*, String *title*);

Description

The `template_addBinder` operation adds a fully configured workspace or folder to the workspace hierarchy.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

parentId

The identifier of the workspace or folder that is to contain the new binder.

binderConfigId

The identifier that maps to the default configuration for the folder you want to create.

You can use the `template_getTemplates` information to get a configuration identifier from a binder that has a configuration you want for your new binder. Or, you can get a binder configuration identifier from the Vibe user interface. See Section 1.7.2, “Adding Folders and the Binder Configuration Identifier,” on page 18, for information about getting a configuration identifier from the user interface.

title

A string containing the title of the new binder.

return_value

The binder identifier of the newly created workspace or folder.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)
- template_getTemplates (page 149)

template_getTemplates

Obtains information about all defined templates in the installation.

Syntax

public TemplateCollection template_getTemplates(String *accessToken*);

Description

The `template_getTemplates` operation obtains information about all defined templates in the installation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

return_value

A TemplateCollection Java object that contains information about all templates in the installation.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

zone_addZone

Adds a zone to the installation.

Syntax

public Long zone_addZone(String *accessToken*, String *zoneName*, String *virtualHost*, String *mailDomain*);

Description

The `zone_addZone` operation adds a zone to the installation.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

zoneName

A string containing the name of the new zone.

virtualHost

A string specifying the virtual host. (See the installation guide for more information.)

mailDomain

This parameter is not used.

return_value

The zone identifier, which is the binder identifier of the top workspace in the new zone.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

zone_deleteZone

Deletes a zone.

Syntax

public void zone_deleteZone(String *accessToken*, String *zoneName*);

Description

The `zone_deleteZone` operation deletes a zone.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

zoneName

A string containing the name of the zone you want to delete.

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

zone_modifyZone

Modifies a zone.

Syntax

public void zone_modifyZone(String *accessToken*, String *zoneName*, String *virtualHost*, String *mailDomain*);

Description

The `zone_modifyZone` operation changes a zone’s virtual host specification.

Parameters and Return Value

accessToken

Either the security token passed to your application by Vibe as part of implementing a remote application, or the null value.

zoneName

A string containing the name of the new zone.

virtualHost

A string specifying the virtual host. (See the installation guide for more information.)

mailDomain

This parameter is not used

return_value

None.

See Also

- Java objects in the Vibe sources (see Section 1.7.1, “Working with Java Objects,” on page 17)

Deprecated Web Services Operations

B

This topic provides alphabetized reference pages for deprecated Web services operations provided by Kablink Teaming.

NOTE: Novell recommends that you do not use these Web services operations in new applications. Instead, use the operations documented in Appendix A, "Web Services Operations," on page 69. Novell continues to support the operations in this appendix for backward compatibility for applications written to interoperate with Teaming 1.0.3 or earlier.

The following are conventions used in this reference section:

What you see	What it means
Click the *Add a team workspace* button. Click the *Getting Started* link.	Items that are clickable on the page, programming variables, or syntax parameters are presented in *italic* font.
Blog summary - Provides a.... **Note:** Remember that....	Defined terms in a list, note headers, section headers on a reference page, and list items on a reference page are presented in bold font.
Type `status`, then press Enter. Open the `ManagerGuide.pdf` file. Use the `open_db` routine with its `lock` parameter.	Text that you must type, file names, commands, command options, routines, Web services messages, and parameters are presented in `Courier` font when occurring in a body of text.
[page]	Optional syntax parameters are enclosed in brackets ([ ]).
..., paramSyntax1 | paramSyntax2,...	Required parameters that accept two or more optional syntaxes are separated by the vertical-line character.
(V1—V1.0.3)	The versions of Teaming that support the Web services operation ("all versions between Version 1.0 through Version 1.0.3")

NOTE: All examples in this reference section use Apache Axis run-time library methods that specify Web service operations and their argument lists. If you are not using Apache Axis, map the Apache methods to those you are using to implement your Web service calls.

The `search` operation is under development and subject to change or deletion at any time. Do not use this operation in your client applications.

Web service operations contained in this reference section are used by this Windows based client: `/ssf/samples/wsclient/facade-client.bat`. With the exception of `uploadCalendarEntries`, use the same parameters for the batch-file command that you use for the corresponding Web service message.

The following table maps the `facade-client.bat` command name to its corresponding, linked Web services message, which is documented in this reference section:

facade-client.bat command	Web services message
`addEntry`	addFolderEntry
`addFolder`	addFolder
`addReply`	addReply
[none]	addUserWorkspace
`indexBinder`	indexFolder
`listDefinitions`	getDefinitionListAsXML
`migrateBinder`	migrateBinder
`migrateEntry`	migrateFolderEntry
`migrateReply`	migrateReply
`migrateFile`	migrateFolderFile
`migrateFileStaged`	migrateFolderFileStaged
`migrateWorkflow`	migrateEntryWorkflow
`modifyEntry`	modifyFolderEntry
`printAllPrincipals`	getAllPrincipalsAsXML
`printDefinition`	getDefinitionAsXML
`printDefinitionConfig`	getDefinitionConfigAsXML
`printFolderEntry`	getFolderEntryAsXML
`printFolderEntries`	getFolderEntryAsXML
`printPrincipal`	getPrincipalAsXML
`printTeamMembers`	getTeamMembersAsXML
`printTeams`	getTeamsAsXML
`printWorkspaceTree`	getWorkspaceTreeAsXML
`setDefinitions`	setDefinitions
`setFunctionMembership`	setFunctionMembership
`setFunctionMembershipInherited`	setFunctionMembershipInherited
`setOwner`	setOwner
`setTeamMembers`	setTeamMembers
`synchronize`	synchronizeMirroredFolder
`uploadCalendar`	uploadCalendarEntries
`uploadFile`	uploadFolderFile

addFolder

Adds a folder to the workspace-tree hierarchy. (V1—V1.0.3)

Syntax

public long addFolder(long *parentBinderId*, long *binderConfigId*, String *title*);

Description

The `addFolder` operation adds a folder to the workspace and folder hierarchy.

Parameters and Return Value

parentBinderId

The identifier of the workspace or folder that is to contain the new folder.

binderConfigId

The identifier that maps to the default configuration for the folder you want to create.

title

A string providing a title for the new entry.

return_value

The binder identifier of the newly created folder.

Example

```
call.setOperationName(new QName("addFolder"));
Object result = call.invoke(new Object[] {new Long(21), new Long(146), new
String("My new folder")});
```

This code creates a new to the container whose binder identifier is `21`, gives the folder a configuration identifier of 146 (on our test installation, this corresponds to a discussion folder), and establishes the title of the new folder as *My new folder*. The container whose binder identifier is `21` can be either a workspace or folder.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section 1.7.1, "Working with Java Objects," on page 17

addFolderEntry

Adds an entry to a folder. (V1—V1.0.3)

Syntax

public long addFolderEntry(long *folderId*, String *definitionId*, String *inputDataAsXML*, String *attachedFileName* | null);

Description

The `addFolderEntry` operation adds an entry to a folder.

Parameters and Return Value

folderId

The binder identifier of the folder that is to contain the new entry.

definitionId

The 32-character, hexadecimal identifier that maps to the type of entry to be created (for example, some default entry types are topic, file, blog, wiki, and calendar).

The easiest way to work with definition identifiers for entries is to specify `null` for this value. When you specify `null`, Teaming automatically applies the definition identifier for the default entry type of the folder in which you are creating a new entry. For example, by default, you want to create an entry in a blog folder. If you pass `null` as the definition identifier, Teaming automatically applies the definition identifier for a blog entry.

As another option, you can use the `getDefinitionConfigAsXML` operation to get information about all definitions. Then, you can parse the XML string for the definition identifier of the type of entry you want.

inputDataAsXML

A string of XML containing the values needed to create an entry of your desired type.

Use the Teaming UI to create a complete entry of the type you want this Web services operation to create, note the entry identifier, and then use the `getFolderEntryAsXML` operation to return XML for the entry. Then, use the returned XML as a template for this parameter.

attachedFileName

The name of the file you wish to attach to the new entry. This is an optional parameter. The file must be located in the directory in which the client code executes.

return_value

The entry identifier for the newly created entry.

Examples

```
call.setOperationName(new QName("addFolderEntry"));
Object result = call.invoke(new Object[] {new Long(21), new
String("402883b90cc53079010cc539bf260002"), s, filename}, filename);
```

This code creates a new entry in the folder whose binder identifier is `21`; the specified entry-definition identifier maps to a discussion topic. The variable `s` contains XML elements needed by Teaming to create the entry. The new entry includes the attached file whose filename is specified by the value of the `filename` variable.

```
call.setOperationName(new QName("addFolderEntry"));
Object result = call.invoke(new Object[] {new Long(21), new
String("402883b90cc53079010cc539bf260002"), s, null});
```

This code produces the same effect as the last example, except that it does not attach a file.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- getFolderEntryAsXML (page 163)
- getDefinitionConfigAsXML (page 161)

addReply

Adds a new comment to an entry or comment. (V1.0.3)

Syntax

public long addReply(long *folderId*, long *parentEntryId*, String *definitionId*, String *inputDataAsXML*, String *attachedFileName* | null);

Description

The `addReply` operation adds a new comment to an entry or to an existing comment.

Parameters and Return Value

folderId

The binder identifier of the folder containing the entry or comment to which you want to apply the new comment.

parentEntryId

The entry identifier for the entry or comment to which you want to apply the comment.

definitionId

The 32-character, hexadecimal identifier that maps to the type of comment to be created.

You can use the `getDefinitionListAsXML` operation to get metadata for all definitions. Then, you can parse the XML string for the definition identifier of the type of comment you want.

inputDataAsXML

A string of XML containing the values needed to create a comment of your desired type.

Use the Teaming UI to create a complete comment of the type you want this Web services operation to create, note the entry identifier, and then use the `getFolderEntryAsXML` operation to return XML for the entry. Then, use the returned XML as a template for this parameter.

attachedFileName

The name of the file you wish to attach to the new comment. This is an optional parameter. The file must be located in the directory in which the client code executes.

return_value

The entry identifier of the newly created comment.

Example

```
call.setOperationName(new QName("addReply"));
Object result = call.invoke(new Object[] {new Long(21), new Long(45), null, s,
null});
```

This code creates a new comment in the folder whose binder identifier is `21`, and applies it to the entry or comment whose entry identifier is `45`. The first `null` value instructs Teaming to use the default comment type for the folder. The variable `s` contains XML elements needed by Teaming to create the comment. Because of the final `null` value, the new comment does not include an attached file.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- getFolderEntryAsXML (page 163)
- getDefinitionListAsXML (page 161)

addUserWorkspace

Adds a new personal workspace. (V1.0.3)

Syntax

public long addUserWorkspace (long *userId*);

Description

The `addUserWorkspace` operation adds a new personal workspace to the workspace hierarchy.

The primary purpose of this operation is to assist with migrating data from SiteScape Forum to Teaming. By default using Teaming, the creation of the personal workspace occurs when someone first uses the portal software to sign in with a username and password. If you want to migrate Forum information as sub-content to a personal workspace in Teaming, use this operation before creating the sub-content.

Parameters and Return Value

userId

The identifier for the user for whom you want to create the personal workspace

return_value

The binder identifier of the newly created personal workspace.

Example

```
call.setOperationName(new QName("addUserWorkspace"));
Object result = call.invoke(new Object[] {new Long(21)});
```

This code creates a new personal workspace.

See Also

- Section C.1, "Sequence of Migration Operations," on page 187
- Section C.3, "Migrating Users," on page 188

getAllPrincipalsAsXML

Returns summary information for users and groups. (V1—V1.0.3)

Syntax

public String getAllPrincipalsAsXML(int *firstRecord*, int *maxRecords*);

Description

The `getAllPrincipalsAsXML` operation returns XML elements that provide summary information about registered users and defined groups. You can use this operation to identify a particular user by name or other data, obtain an identifier for a particular user, and then use the `getPrincipalAsXML` operation to gather a finer level of information about that person.

Parameters and Return Value

firstRecord

The index of the first record whose user or group information you want to obtain. The index for the first principal in the system is `1`.

maxRecords

The maximum number of user and group records whose information should be returned.

You can use the previous parameter and this parameter in subsequent calls to `getAllPrincipalsAsXML` to process data for sets of users and groups at a time (for example, 50 at a time, or 100 at a time).

return_value

A string containing the XML elements providing information about the requested set of users and groups.

Example

```
call.setOperationName(new QName("getAllPrincipalsAsXML"));
Object result = call.invoke(new Object[] {new Integer(100), new Integer(50)});
```

This code requests information for users and groups starting with the record number 100 and including up to 50 records.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- getAllPrincipalsAsXML (page 159)

getDefinitionAsXML

Returns information about one definition. (V1—V1.0.3)

Syntax

public String getDefinitionAsXML(String *definitionId*);

Description

The `getDefinitionAsXML` operation returns an XML string containing information about one definition. You work with definitions using the designers in the administration UI.

For example, if you pass one of the definition identifiers for an entry type listed in the `addFolderEntry` reference page, Teaming returns information about the definition for that entry.

As an alternative, you can use the `getDefinitionConfigAsXML` operation to obtain all definitions in Teaming and then parse the larger string for the definition information you want.

Parameter and Return Value

definitionId

The identifier of the definition whose information you want. Definitions are maintained using the designers in the administration UI, and define the components of an object in Teaming.

return_value

A string of XML whose elements provide information about the components of an object in Teaming.

Example

```
call.setOperationName(new QName("getDefinitionAsXML"));
Object result = call.invoke(new Object[] {new
String("402883b9114739b301114754e8120008")});
```

This code requests XML-formatted information about the definition for a wiki entry.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- addFolderEntry (page 155)
- getDefinitionConfigAsXML (page 161)

getDefinitionConfigAsXML

Returns information about all configuration definitions. (V1—V1.0.3)

Syntax

public String getDefinitionConfigAsXML( );

Description

The `getDefinitionConfigAsXML` operation returns information about all configuration definitions. The configuration information does not include workflow or template definitions. You can uses the returned information to extract the definition identifier for a given entry type to use in a subsequent call to `addFolderEntry`.

Return Value

return_value

A string of XML whose elements describe all configuration definitions.

Example

```
call.setOperationName(new QName(“getDefinitionConfigAsXML”));
Object result = call.invoke();
```

This code obtains information about all configuration settings.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- addFolderEntry (page 155)

getDefinitionListAsXML

Returns metadata for all definitions in the installation. (V1.0.3)

Syntax

public String getDefinitionListAsXML ();

Description

The `getDefinitionListAsXML` operation returns metadata for all definitions in the installation. This metadata includes information such as the definition name and identifier.

When using other Web services operations that require a definition identifier, you can use this message, parse the XML for the name (discussion, blog, calendar, comment), and obtain the 32-character, hexadecimal identifier that maps to the desired object.

Return Value

return_value

A string of XML whose elements contain metadata for all definitions in the installation.

Example

```
call.setOperationName(new QName("getDefinitionListAsXML"));
Object result = call.invoke();
```

This code obtains metadata for all definitions in the installation.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)

getFolderEntriesAsXML

Returns a string containing XML providing summary information about entries in a folder. (V1—V1.0.3)

Syntax

public String getFolderEntriesAsXML(long *folderId*);

Description

The `getFolderEntriesAsXML` operation returns XML elements containing summary information about each entry in the specified folder.

Parameter and Return Value

folderId

The binder identifier of the folder containing the entries for which you want information.

return_value

A string containing XML elements containing summary information for each entry in the folder specified by `folderId`.

Example

```
call.setOperationName(new QName("getFolderEntriesAsXML"));
Object result = call.invoke(new Object[] {new Long(21)});
```

This code returns a string containing XML information for all of the entries in the folder whose binder identifier is `21`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)

getFolderEntryAsXML

Returns information about one entry in a folder. (V1—V1.0.3)

Syntax

public String getFolderEntryAsXML(long *folderId*, long *entryId*, boolean *includeAttachments*);

Description

The `getFolderEntryAsXML` operation returns XML whose elements provide information about one entry in a folder.

Parameters and Return Value

folderId

The binder identifier of the folder containing the entry whose information you want.

entryId

The identifier of the entry whose information you want.

includeAttachments

A boolean value that indicates whether you want Teaming to return the entry's attachments. The client program is responsible for placement of attachment files on its local system.

return_value

A string containing XML elements for the requested entry.

Example

```
call.setOperationName(new QName("getFolderEntryAsXML"));
Object result = call.invoke(new Object[] {new Long(21), new Long(34), new
Boolean.FALSE});
```

This code returns XML that includes information contained in entry number 34 in the folder whose identifier is 21. Because of the value of the last parameter, Teaming does not place the entry's file attachments in the client program's source directory.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section 1.7.4, “Fetching Attachments,” on page 20

getPrincipalAsXML

Returns information about one user or group. (V1—V1.0.3)

Syntax

public String getPrincipalAsXML(long *binderId*, long *principalId*);

Description

The `getPrincipalAsXML` operation returns XML whose elements provide information about one registered user or defined group.

Parameters and Return Value

binderId

The binder identifier of the principal’s parent workspace. The information returned by `getAllPrincipalsAsXML` includes the binder number of this containing workspace.

principalId

The identifier that maps to the user or group for which you want to gather information.

return_value

A string containing XML elements whose elements provide information about the specified user or group.

Example

```
call.setOperationName(new QName("getPrincipalAsXML"));
Object result = call.invoke(new Object[] {new Long(2), new Long(25)});
```

This code returns information about a user or group, whose parent workspace has a binder identifier of `2` and whose principal identifier is `25`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- getAllPrincipalsAsXML (page 159)

getTeamMembersAsXML

Returns information about all team members assigned within a workspace or folder. (V1—V1.0.3)

Syntax

public String getTeamMembersAsXML(long *binderId*);

Description

The `getTeamMembersAsXML` operation returns XML that names members of a team assigned within the specified workspace or folder.

Parameter and Return Value

binderId

The binder identifier of the workspace or folder for which you want information about team members. The `getTeamsAsXML` operation returns information about all workspaces and folders that have assigned teams.

return_value

A string containing XML elements describing team members for the specified place.

Example

```
call.setOperationName(new QName("getTeamMembersAsXML"));
Object result = call.invoke(new Object[] {new Long(23));
```

This code returns an XML string whose elements describe all of the team members assigned in the workspace or folder associated with the binder identifier of `23`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- getTeamsAsXML (page 165)

getTeamsAsXML

Returns information about all workspaces and folders that have assigned teams. (V1—V1.0.3)

Syntax

public String getTeamsAsXML( );

Description

The `getTeamsAsXML` operation returns an XML string providing information about all workspaces and folders that have assigned teams. You can use this operation to obtain the list of places that have assigned teams, note a binder number of a particular place, and then use the `getTeamMembersAsXML` operation to obtain the list of team members for that place.

Return Value

return_value

An XML string whose elements describe workspaces and folders that have assigned teams.

Example

```
call.setOperationName(new QName("getTeamsAsXML"));
Object result = call.invoke();
```

This code returns information about all places in the Teaming installation that have assigned teams.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- getTeamMembersAsXML (page 165)

getWorkspaceTreeAsXML

Returns information needed to construct the Teaming workspace and folder tree. (V1—V1.0.3)

Syntax

public String getWorkspaceTreeAsXML(long *binderId*, int *levels*, String *page*);

Description

The `getWorkspaceTreeAsXML` operation returns XML elements needed to construct the requested portion of the Teaming workspace tree.

Parameters and Return Value

binderId

The binder identifier of the starting node of the returned portion of the hierarchy. The top workspace in the Teaming tree has a binder identifier of `1`.

levels

The number of hierarchical levels down from the node specified by `binderId` that you want to include in the returned information. The value `-1` indicates that you want all subsequent levels.

page

A parameter used to expand pages of binders. When you specify a valid page identifier, Teaming expands the page by the levels indicated in the `levels` parameter.

If you do not want to expand pages using this call, pass `null` as this parameter.

The Web-services overview topic contains more detailed information about working with pages (Section 1.7.6, “Binder Pages and search_getWorkspaceTreeAsXML,” on page 20).

return_value

A string containing XML elements needed to construct each node within the requested levels of the workspace hierarchy.

Example

```
call.setOperationName(new QName("getWorkspaceTreeAsXML"));
Object result = call.invoke(new Object[] {new Long(1), new Integer(3), null});
```

This code returns a string containing XML information for the first three levels of the workspace hierarchy. The following depicts these levels using default workspace titles:

```
Level 1: Workspaces
     Level 2: Global, Personal, and Team workspaces
          Level 3: Children of Global, Personal, and Team
```

The children of *Global workspaces*, *Personal workspaces*, and *Team workspaces* can be either workspaces or folders.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section 1.7.6, “Binder Pages and search_getWorkspaceTreeAsXML,” on page 20

indexFolder

Indexes a folder. (V1.0.3)

Syntax

public void indexFolder(long *folderId*);

Description

The `indexFolder` operation indexes a folder.

The primary use of this operation is to index data after you migrate it from SiteScape Forum into Teaming. (The migration operations transfer the data but do not index it.)

Parameter

folderId

The binder identifier of the folder you want to index.

Example

```
call.setOperationName(new QName("indexFolder"));
Object result = call.invoke(new Object[] {new Long(21)});
```

This indexes the folder whose binder identifier is `21`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187

migrateBinder

Creates a new workspace or folder while preserving SiteScape Forum data. (V1.0.3)

Syntax

public long migrateBinder (long *parentId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*);

Description

The `migrateBinder` operation creates a workspace or folder in Teaming that preserves values from a SiteScape Forum installation (for example, the name of the person who created the item in Forum, the Forum creation date, the person who last modified the item in Forum, and the date of the last modification in Forum).

Parameters and Return Value

parentId

The binder identifier of the parent of the newly created workspace or folder.

definitionId

The 32-character, hexadecimal identifier that maps to the type of workspace or folder to be created.

You can use the `getDefinitionListAsXML` operation to get metadata for all definitions. Then, you can parse the XML string for the definition identifier of the type of workspace or folder you want to create.

inputDataAsXML

A string of XML supplying the elements and values needed to construct the workspace or folder you want to create.

creator

A string containing the username of the person who created the corresponding workspace or folder in the Forum installation.

creationDate

Calendar data specifying the date when the corresponding workspace or folder was created in Forum.

modifier

A string containing the username of the person who last modified the corresponding workspace or folder in Forum.

modificationDate

Calendar data specifying the date when the corresponding workspace or folder was modified in Forum.

return_value

The binder identifier of the newly created workspace or folder.

Example

```
call.setOperationName(new QName("migrateBinder"));
Object result = call.invoke(new Object[] {new Long(21), def, input, new
String("JSmith"), createcal, new String("JGarces"), modcal});
```

This code creates a new binder determined by the definition in the `def` variable (use the `getDefinitionListAsXML` operation to obtain the correct string for your binder type), and the binder will be a child of the binder whose identifier is `21`. The `input` variable contains an XML string, properly formatted for your binder type, which Teaming uses to create binder content. The remaining four parameters provide names (literals) and dates (the `createcal` and `modcal` variables) for the creation and last modification of the corresponding item in the Forum installation.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187
- getDefinitionListAsXML (page 161)

migrateEntryWorkflow

Associates an entry with a workflow process while preserving SiteScape Forum data. (V1.0.3)

Syntax

public void migrateEntryWorkflow (long *binderId*, long *entryId*, String *definitionId*, String *startState*, String *modifier*, Calendar *modificationDate*);

Description

The `migrateEntryWorkflow` operation associates a workflow process with an entry in Teaming, while preserving values from a SiteScape Forum installation (for example, the state to which the entry should be set, the person who last changed workflow state in Forum, and the date of the last state change in Forum).

Parameters and Return Value

binderId

The binder identifier of the folder that contains the entry to which you want to associate a workflow process.

entryId

The entry identifier of the entry to which you want to associate a workflow process.

definitionId

The 32-character, hexadecimal identifier that maps to the workflow-process definition.

Before using this message, you must replicate the Forum workflow processes in Teaming.

startState

The current state of the Teaming entry (which would reflect its last state in Forum).

modifier

A string containing the username of the person who last changed the workflow process in Forum.

modificationDate

Calendar data specifying the date when the workflow process was last changed in Forum.

Example

```
call.setOperationName(new QName("migrateEntryWorkflow"));
Object result = call.invoke(new Object[] {new Long(21), new Long(45),
String("ptoProcess"), String("PTO Request"), new String("JGarces"), modcal});
```

This code associates the `ptoProcess` workflow process with the entry whose identifier is `45` and which is located in a folder whose binder identifier is `21`. The entry should be placed in the `PTO Request` state. The operation also provides the name of the person who last changed the workflow state in Forum and the date when that state change occurred.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)

- Section C.1, “Sequence of Migration Operations,” on page 187
- Section C.5, “Migrating Custom Commands and Workflow,” on page 189

migrateFolderEntry

Creates a new folder entry while preserving SiteScape Forum data. (V1.0.3)

Syntax

public void migrateFolderEntry (long *binderId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*);

Description

The `migrateFolderEntry` operation creates a folder entry in Teaming that preserves values from a SiteScape Forum installation (for example, the name of the person who created the item in Forum, the Forum creation date, the person who last modified the item in Forum, and the date of the last modification in Forum).

When creating entries within a Files folder in Teaming, use this operation to create the entry, and then use either `migrateFolderFile` or `migrateFolderFileStaged` to attach the file to the entry.

Parameters and Return Value

binderId

The binder identifier of the folder to contain the new entry.

definitionId

The 32-character, hexadecimal identifier that maps to the type of entry to be created.

The easiest way to work with definition identifiers for entries is to specify `null` for this value. When you specify `null`, Teaming automatically applies the definition identifier for the default entry type of the folder in which you are creating a new entry. For example, by default, you want to create an entry in a blog folder. If you pass `null` as the definition identifier, Teaming automatically applies the definition identifier for a blog entry.

As another option, you can use the `getDefinitionListAsXML` operation to get metadata for all definitions. Then, you can parse the XML string for the definition identifier of the type of workspace or folder you want to create.

inputDataAsXML

A string of XML supplying the elements and values needed to construct the type of entry you want to create.

creator

A string containing the username of the person who created the corresponding entry in the Forum installation.

creationDate

Calendar data specifying the date when the corresponding entry was created in Forum.

modifier

A string containing the username of the person who last modified the corresponding entry in Forum.

modificationDate

Calendar data specifying the date when the corresponding entry was modified in Forum.

return_value

The entry identifier of the newly created entry.

Example

```
call.setOperationName(new QName("migrateFolderEntry"));
Object result = call.invoke(new Object[] {new Long(21), def, input, new
String("JSmith"), createcal, new String("JGarces"), modcal});
```

This code creates a new entry of the type determined by the definition in the `def` variable (use the `getDefinitionListAsXML` operation to obtain the correct string for your entry type), and the new entry is to be located in the binder whose identifier is `21`. The `input` variable contains an XML string, properly formatted for your entry type, which Teaming uses to create entry content. The remaining four parameters provide names (literals) and dates (the `createcal` and `modcal` variables) for the creation and last modification of the corresponding entry in the Forum installation.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187
- getDefinitionListAsXML (page 161)
- migrateFolderEntry (page 171)
- migrateFolderFileStaged (page 173)

migrateFolderFile

Attaches a file to an entry while preserving SiteScape Forum data. (V1.0.3)

Syntax

public void migrateFolderFile (long *binderId*, long *entryId*, String *fileUploadDataItemName*, String filename, String *modifier*, Calendar *modificationDate*);

Description

The `migrateFolderFile` operation attaches a file to a folder entry in Teaming that preserves values from a SiteScape Forum installation (for example, the person who last modified the item in Forum, and the date of the last modification in Forum).

Parameters and Return Value

binderId

The binder identifier of the folder that contains the entry to which you want to attach a file.

entryId

The entry identifier of the entry to which you want to attach the file.

fileUploadDataItemName

The internal-use name used by the database to identify the file as an element of an entry.

For example, a Forum custom command allowed for uploading different files into a single entry that served different functions, such as an expense report, a meeting presentation, and so on. These custom file uploads have associated internal-use names that are different than the reserved internal-use name applied to standard file entries or standard attachments.

If you are migrating to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

filename

The name of the file to be attached to the entry.

modifier

A string containing the username of the person who last modified the corresponding file in Forum.

modificationDate

Calendar data specifying the date when the corresponding file was modified in Forum.

Example

```
call.setOperationName(new QName("migrateFolderFile"));
Object result = call.invoke(new Object[] {new Long(21), new Long(45),
String("_budgetReport"), String("budget-report.xls"), new String("JGarces"),
modcal});
```

This code attaches the `budget-report.xls` file to the entry whose identifier is `45` and is located in a folder whose binder identifier is `21`. The internal-use name that maps to the file as an element in the entry is `_budgetReport`. The operation also provides the name of the person who modified the file in Forum and the date when that modification occurred.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187
- Section C.4, "Migrating Files," on page 188

migrateFolderFileStaged

Locates a locally stored file, and attaches it to an entry while preserving Forum data. (V1.0.3)

Syntax

public void migrateFolderFileStaged (long *binderId*, long *entryId*,
String *fileUploadDataItemName*, String filename, String *stagedFileRelativePath*, String *modifier*,
Calendar *modificationDate*);

Description

The `migrateFolderFileStaged` accesses a Forum file that has been copied locally on the Teaming server as a way to streamline the transfer of files, avoiding transferring them over the Internet. The operation then attaches the file to a folder entry in Teaming that preserves values from a SiteScape Forum installation (for example, the person who last modified the item in Forum, and the date of the last modification in Forum).

Parameters and Return Value

binderId

The binder identifier of the folder that contains the entry to which you want to attach a file.

entryId

The entry identifier of the entry to which you want to attach the file.

fileUploadDataItemName

The internal-use name used by the database to identify the file as an element of an entry.

For example, a Forum custom command allowed for uploading different files into a single entry that served different functions, such as an expense report, a meeting presentation, and so on. These custom file uploads have associated internal-use names that are different than the reserved internal-use name applied to standard file entries or standard attachments.

If you are migrating to a folder file, specify `upload` as an argument to this parameter to make this attachment the primary file for the entry.

filename

The name of the file to be attached to the entry.

stagedFileRelativePath

The relative path specification, beginning with the staging area designated in the `ssf.properties` and `ssf-ext.properties` files on the Teaming server. (See the installation guide for more information about these files.)

Although the files can be present in any folder structure within the staging area, one streamlined way to approach this task is to unzip the Forum hidden directory into the staging area. Then, use this parameter to specify the relative path through the hidden folder structure to the location of the file to be attached to the entry in Teaming.

modifier

A string containing the full name of the person who last modified the corresponding file in Forum.

modificationDate

Calendar data specifying the date when the corresponding file was modified in Forum.

Example

```
call.setOperationName(new QName("migrateFolderFileStaged"));
Object result = call.invoke(new Object[] {new Long(21), new Long(45),
String("_budgetReport"), String("budget-report.xls"), String("hidden/ssf/
myworkspace/myforum/4567849"), new String("JGarces"), modcal});
```

To locate the file, Teaming begins with the defined staging folder and then applies the relative path `hidden/ssf/myworkspace/myforum/456789`.This code attaches the `budget-report.xls` file to the entry whose identifier is `45` and is located in a folder whose binder identifier is `21`. The internal-use name that maps to the file as an element in the entry is `_budgetReport`. The operation also provides the name of the person who modified the file in Forum and the date when that modification occurred.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187
- Section C.4, "Migrating Files," on page 188

migrateReply

Creates a new comment while preserving SiteScape Forum data. (V1.0.3)

Syntax

public void migrateReply (long *binderId*, long *parentId*, String *definitionId*, String *inputDataAsXML*, String *creator*, Calendar *creationDate*, String *modifier*, Calendar *modificationDate*);

Description

The `migrateReply` operation creates a comment in Teaming that preserves values from a SiteScape Forum installation (for example, the name of the person who created the item in Forum, the Forum creation date, the person who last modified the item in Forum, and the date of the last modification in Forum).

Parameters and Return Value

binderId

The binder identifier of the folder that will contain the new comment.

parentId

The binder identifier of the entry or comment to which you want to apply the new comment.

definitionId

The 32-character, hexadecimal identifier that maps to the type of comment to be created.

You can use the `getDefinitionListAsXML` operation to get metadata for all definitions. Then, you can parse the XML string for the definition identifier of the type of comment you want to create.

inputDataAsXML

A string of XML supplying the elements and values needed to construct the type of comment you want to create.

creator

A string containing the username of the person who created the corresponding reply in the Forum installation.

creationDate

Calendar data specifying the date when the corresponding reply was created in Forum.

modifier

A string containing the username of the person who last modified the corresponding reply in Forum.

modificationDate

Calendar data specifying the date when the corresponding reply was modified in Forum.

return_value

The entry identifier of the newly created comment.

Example

```
call.setOperationName(new QName("migrateReply"));
Object result = call.invoke(new Object[] {new Long(21), new Long(45), def,
input, new String("JSmith"), createcal, new String("JGarces"), modcal});
```

This code creates a new comment of the type determined by the definition in the `def` variable (use the `getDefinitionListAsXML` operation to obtain the correct string for your comment type). The new comment is to be located in the binder whose identifier is 21, and applied to an entry or comment whose identifier is 45. The `input` variable contains an XML string, properly formatted for your comment type, that Teaming uses to create comment content. The remaining four parameters provide names (literals) and dates (the `createcal` and `modcal` variables) for the creation and last modification of the corresponding reply in the Forum installation.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section C.1, “Sequence of Migration Operations,” on page 187

modifyFolderEntry

Modifies a single entry. (V1—V1.0.3)

Syntax

public void modifyFolderEntry(long *folderId*, long *entryId*, String *inputDataAsXML*);

Description

The `modifyFolderEntry` operation modifies one entry in a folder.

Parameters and Return Value

folderId

The binder identifier of the folder that contains the entry to be modified.

entryId

The identifier of the entry to be modified.

inputDataAsXML

A string of XML containing the values needed to modify the entry.

return_value

None.

Example

```
call.setOperationName(new QName("modifyFolderEntry"));
Object result = call.invoke(new Object[] {new Long(21), new Long(43), s});
```

This code modifies entry 43 in the folder whose binder ID is `21`. The variable `s` contains XML elements needed by Teaming to modify the contents of the entry.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)

setDefinitions

Establishes workflow-entry associations for a folder. (V1.0.3)

Syntax

public void migrateEntryWorkflow (long *binderId*, String[] *definitionIds*, String[] *workflowAssociations*);

Description

The `setDefinitions` operation uses two arrays to associate workflow identifiers with entry identifiers for a folder. (Teaming associates identifiers in the first element of both arrays, the second element of both arrays, the third, and so on.)

When an entry is associated with a workflow process, creation of an entry of that type automatically places the entry into the initial state of the workflow process.

NOTE: This operation is an overwrite operation, setting all workflow associations for the folder; you cannot use repeated calls to this operation to set associations incrementally. So, set all of the workflow associations for the folder with one call.

Parameters and Return Value

binderId

The binder identifier of the folder in which you want to associate entry and workflow identifiers.

definitionIds

An array of entry identifiers.

workflowAssociations

An array of workflow identifiers.

Before using this message, you must replicate the Forum workflow processes in Teaming.

Example

```
call.setOperationName(new QName("setDefinitions"));
Object result = call.invoke(new Object[] {new Long(21), entries, workflows});
```

This code passes two array variables, `entries` and `workflows`. Teaming uses the corresponding elements in both arrays to create entry-workflow associations for the folder whose binder identifier is `21`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)

setFunctionMembership

Applies access-control settings to a folder or workspace. (V1—V1.0.3)

Syntax

public void setFunctionMembership(long *binderId*, String *inputDataAsXML*);

Description

The `setFunctionMembership` operation provides access-control settings for folder or a workspace. The term function is analogous to a role in the user interface (UI).

The primary use of this operation is to establish access-control settings when migrating workspaces and folders from Forum to Teaming. You must ensure that you have migrated Forum user and group names to Teaming that are required for your access-control settings.

NOTE: This operation is an overwrite operation, setting all function memberships for the folder or workspace; you cannot use repeated calls to this operation to set memberships incrementally. So, set all memberships for the workspace or folder with one call.

Parameters and Return Value

binderId

The binder identifier of the folder or workspace for which you want to set access control.

inputDataAsXML

A string of XML containing the values needed to set access control. Here is an example of XML that sets the visitor function:

```
<workAreaFunctionMemberships>
<workAreaFunctionMembership>
<property name="functionName">__role.visitor</property>
<property name="memberName">jGarces</property>
<property name="memberName">sChen</property>
<property name="members">1,2,3</property>
</workAreaFunctionMembership>
    .
    .
    .
</workAreaFunctionMemberships>
```

To obtain the `functionName` value:

1. Sign in as a site administrator for Teaming.
2. In the administration portlet, click *Configure role definitions*.
3. Click any item (for example, *Participant*).
4. Note or copy the identifier in the *Role Name* text box (for example, `__role.participant`). This identifier begins with a double underscore (_).

You can pass either user or group names (for example, `jGarces` or `sChen`) or user or group identifiers (for example, `1`, `2`, `3`). Teaming reserves the identifiers `-1` for the workspace or folder owner, and `-2` for a team member.

Example

```
call.setOperationName(new QName("setFunctionMembership"));
Object result = call.invoke(new Object[] {new Long(21), s});
```

This code uses the content of the XML string s to establish access-control settings for the folder or workspace whose binder identifier is 21.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section C.1, “Sequence of Migration Operations,” on page 187

setFunctionMembershipInherited

Establishes inheritance as the access-control mechanism for a folder or workspace. (V1.0.3)

Syntax

public void setFunctionMembershipInherited(long *binderId*, boolean *inherit*);

Description

The `setFunctionMembershipInherited` operation allows you to establish that a folder or workspace is to inherit its access-control settings from the parent binder. The primary purpose of this operation is to set inheritance for folders and workspaces that you migrate from Forum.

Parameters and Return Value

binderId

The binder identifier of the folder or workspace for which you want to establish inheritance for its access-control settings.

inherit

A boolean value that determines whether the folder or workspace uses inheritance to establish its access settings.

Example

```
call.setOperationName(new QName("setFunctionMembershipInherited"));
Object result = call.invoke(new Object[] {new Long(21), new Boolean.TRUE});
```

This code establishes inheritance as the access-control mechanism for the folder or workspace whose binder identifier is 21.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section C.1, “Sequence of Migration Operations,” on page 187

setOwner

Establishes the owner of a folder or workspace. (V1.0.3)

Syntax

public void setOwner(long *binderId*, long *userId*);

Description

The `setOwner` operation allows you to establish an owner for a folder or workspace. The primary purpose of this operation is to mirror Forum ownership as you migrate folders and workspaces.

Parameters and Return Value

binderId

The binder identifier of the folder or workspace for which you want to establish ownership.

userId

The user identifier of the person whom you want to be the owner of a folder or workspace.

Example

```
call.setOperationName(new QName("setOwner"));
Object result = call.invoke(new Object[] {new Long(21), new Long(345)});
```

This code establishes the user whose identifier is `345` as the owner of the folder or workspace whose binder identifier is 21.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187

setTeamMembers

Establishes the membership of a team for a folder or workspace. (V1.0.3)

Syntax

public void setTeamMembers(long *binderId*, String[] *memberNames*);

Description

The `setTeamMembers` operation establishes the members of a team for a folder or workspace.

Parameters and Return Value

binderId

The binder identifier of the folder or workspace for which you want to establish team membership.

memberNames

An array containing the names of all team members for the folder or workspace.

Example

```
call.setOperationName(new QName("setTeamMembers"));
Object result = call.invoke(new Object[] {new Long(21), users});
```

This code establishes each username in the array `users` as team members for the folder or workspace whose binder identifier is 21.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187

synchronizeMirroredFolder

Synchronizes the mirrored folder with the folder on the external drive. (V1.0.3)

Syntax

public void synchronizeMirroredFolder(long *binderId*);

Description

The `synchronizeMirroredFolder` operation synchronizes a mirrored folder with the corresponding file on the external drive. A new mirrored folder does not synchronize with its external drive until a synchronization occurs manually in the user interface (UI) or using this message.

Parameters and Return Value

binderId

The binder identifier of the mirrored file that you want to synchronize with its external drive.

Example

```
call.setOperationName(new QName("synchronizedMirroredFolder"));
Object result = call.invoke(new Object[] {new Long(21)});
```

This code synchronizes with its external drive the mirrored folder whose binder identifier is 21.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section C.1, "Sequence of Migration Operations," on page 187

uploadCalendarEntries

Creates new calendar entries from a file. (V1—V1.0.3)

Syntax

public void uploadCalendarEntries(long *folderId*, String XMLCalendarData);

Description

The `uploadCalendarEntries` operation uses iCal information in an XML string or in an attachment to add entries to a calendar folder.

NOTE: The `uploadCalendar` command in the `facade-client.bat` batch file accepts two required parameters and an optional third parameter. The second parameter is a file containing XML that specifies iCal data. The third, optional parameter is an iCal formatted file. Both files must be located in the same directory as `facade-client.bat`. Again, if you want the iCal file to be the only source of data for newly created entries, place an empty XML document in the file specified as the second command parameter.

Parameters and Return Value

folderId

The binder identifier of the calendar folder that is to contain the new entries.

XMLCalendarData

A string containing XML formatted calendar data (`<doc><entry>iCal data</entry>...</doc>`). If you wish to specify all of your calendar data in an iCal file attached to the message, pass an empty document for this string (`<doc></doc>`).

return_value

None.

Example

```
call.setOperationName(new QName("uploadCalendarEntries"));
Object result = call.invoke(new Object[] {new Long(21), s});
```

This code creates new entries in the calendar folder whose binder ID is `21`. Teaming uses XML-formatted iCal information contained in the `s` variable to create the new calendar entries.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, “Deprecated Web Services Operations,” on page 153)
- Section 1.7.5, “Adding Calendar Entries,” on page 20

uploadFolderFile

Attaches a file to an entry to a folder. (V1—V1.0.3)

Syntax

public void uploadFolderFile(long *folderId*, String *entryId*, String *fileUploadDataItemName*, String *attachedfileName*);

Description

The `uploadFolderFile` operation attaches a file to an entry in a folder. You can attach only one file at a time; call this operation multiple times to attach more than one file to the entry. Files to be attached must be located in the same directory as the executing client.

Parameters and Return Value

folderId

The binder identifier of the folder that contains the entry to which you want to attach a file.

entryId

The identifier of the entry to which you want to attach a file.

fileUploadDataItemName

A string containing the internal identifier for the part of the entry that contains attached files. This identifier maps the `name` attribute of an `input` HTML tag on a form to the Teaming database; a `hidden` HTML tag communicates this mapping to the server.

The `name` value for the standard entry element containing attached files is `ss_attachFile`. If you want to upload a file into a custom form element you defined using the designers, you need to look up the `name` identifier for that form element (see also `getDefinitionConfigAsXML` or `getFolderEntryAsXML`).

attachedFileName

The name of the file you wish to attach to the new entry. This client is responsible for locating on its local system the file to be used as an attachment.

return_value

None.

Example

```
call.setOperationName(new QName("uploadFolderFile"));
Object result = call.invoke(new Object[] {new Long(21), new Long(43), new
String("ss_attachFile"), filename}, filename);
```

This code attaches a file to entry 43 in the folder whose binder ID is `21`. The name of the file to be attached to the entry is contained in the variable `filename`.

See Also

- The operation table for the Windows based `facade-client.bat` program (Appendix B, "Deprecated Web Services Operations," on page 153)
- Section 1.7.3, "Attaching Files," on page 19
- getDefinitionConfigAsXML (page 161)
- getFolderEntryAsXML (page 163)

Migrating from Forum to Kablink Teaming

C

Kablink Teaming is the ongoing path from the legacy SiteScape Forum product. To assist with migrating data from SiteScape Forum to an installation of Kablink Teaming, Novell developed a set of Web services.

Although this section provides guidance about migrating, the task is complex and requires the active assistance of the Kablink Teaming support team. This is especially true for workflow migration. For more information, please contact the support team and arrange to receive consultation as you perform this task.

- Section C.1, “Sequence of Migration Operations,” on page 187
- Section C.2, “Migration Overwrite Operations,” on page 188
- Section C.3, “Migrating Users,” on page 188
- Section C.4, “Migrating Files,” on page 188
- Section C.5, “Migrating Custom Commands and Workflow,” on page 189

C.1 Sequence of Migration Operations

Some operations require the previous execution of other operations. For example, migrating an entry requires that you have already migrated the folder. As another example, a workflow process requires that you have already migrated user and group names, so that these names can be applied to its access control.

Here are notes regarding the sequence of operations:

- Migrate users and groups, and create personal workspaces early in the process.

  Use LDAP to establish the Forum users in Kablink Teaming.

  You need the existence of personal workspaces to be able to migrate sub-workspaces and child folders. Also, migrating workflow and some types of custom commands requires that your users be established in Kablink Teaming first.

  The Forum term “custom command” maps to “custom view and form” in Kablink Teaming.
- Generally, migrate parents before children you wish to create.

  Examples include migrating parent workspaces before its child folders, and migrating entries before migrating attached files.

  Using SiteScape Forum, a “forum” maps to a “folder” in Kablink Teaming, a “reply” maps to a “comment” in Kablink Teaming, and the process of “attaching a file” maps to the Web services phrase “adding a folder file.”
- Migrate binders before setting their ownership, team members, and access control.

  The Forum items “workspaces and folders” map to the Kablink Teaming Web services term of “binders.” The Forum term “access control” maps to “membership.” Also, the Web services term “function” is equivalent to the term “roles” in the UI for Kablink Teaming.

- Migrate custom commands before creating entries.

  The custom command migration process cannot be done using only Web services (see Section C.5, “Migrating Custom Commands and Workflow,” on page 189, for more information).

- Migrate workflow processes before migrating entries.

  First, the workflow-migration process cannot be done using only Web services (see Section C.5, “Migrating Custom Commands and Workflow,” on page 189, for more information). Second, any entry that is currently in a workflow state requires the presence of the workflow definition in Kablink Teaming.

- After migrating custom commands and workflow, you can migrate workflow associations for specific folders.

- Finalizing operations include indexing folders and synchronizing any mirrored folders that you created.

  Remember that migrated entries do not appear in the UI until you index the folders containing these entries.

  The Kablink Teaming UI does not begin to mirror the files on the drive until someone manually synchronizes them. The Web services call is equivalent to a manual synchronization in the UI.

C.2 Migration Overwrite Operations

Two operations require that you perform the operation for all items using one call to the message; they do not allow you to perform the operation incrementally on subsets of items, using multiple calls to the message. If you call these operations sequentially for subsets of the information, each successive call erases the established data from the previous call.

The operations that require you to perform the operation for all items using only one call are:

- **`binder_setDefinitions`:** Associates entry types with workflow processes (see setDefinitions (page 177)).
- **`binder_setFunctionMembership`:** Sets access control for a workspace or folder (see setFunctionMembership (page 178)).

C.3 Migrating Users

Migrating users requires two steps:

1. Use LDAP to add your Forum users to Kablink Teaming.
2. Use the `migration_addBinder` operation to add personal workspaces for the new Kablink Teaming users.

Migrating custom commands and workflow involve additional work in regard to users. See Section C.5, “Migrating Custom Commands and Workflow,” on page 189, for more information.

C.4 Migrating Files

If you have a small number of files to migrate to Kablink Teaming, you can use the `migration_uploadFolderFile` operation.

However, most Forum installations include a significant number of files, and those files might be large. To improve performance, you should strongly consider using the `migration_uploadFolderFileStaged` message.

Staging involves moving all of the files from SiteScape Forum to the server running the Kablink Teaming installation. Although the files can be located using any folder hierarchy on the server, a convenient way to migrate files is to unzip the Forum hidden directory onto the Kablink Teaming server machine and to work within that existing folder hierarchy from Forum. After placing the files on the Kablink Teaming server, the `migration_uploadFolderFileStaged` operation takes files from the staging area and migrates them into the Kablink Teaming installation.

Here are the steps needed to migrate files:

1. Establish a directory on the Kablink Teaming server machine where you want to place the Forum files.
2. Make the three required changes to the `ssf.properties` and `ssf-ext.properties` files. This action indicates the location of the staging directory. (See the installation guide for more information about the these files.)

   Multiple Forum file versions are separate files in the staged area. Call the `migration_uploadFolderFileStaged` operation once for each version of the file, using the same filename for each call but specifying a different path. This method creates versioned files in Kablink Teaming.
3. Copy the Forum files onto the Kablink Teaming server, using the specified staging directory as your top directory.
4. Use the `migration_uploadFolderFileStaged` operation to migrate the files into the Kablink Teaming installation.

   This command attaches files to an existing entry. Also, it accepts as one of its arguments a relative path, which traverses the s beneath the designated staging directory.

See migrateFolderFileStaged (page 173), for more information.

C.5 Migrating Custom Commands and Workflow

Migrating custom commands and workflow require tasks beyond the scope of using only Web services calls. It is highly recommended that you work closely with the Kablink Teaming support team while completing these tasks.

These are the general steps needed to migrate custom commands and workflow processes:

1. Migrate your Forum users to Kablink Teaming.
2. Use the `profile_getPrincipals` operation to get a list of the user identifiers for the newly created Kablink Teaming users.
3. Create a mapping file that maps Kablink Teaming user identifiers to Forum usernames.
4. Run a Tcl script—which uses the mapping file—to generate an XML file of workflow information.
5. Import the workflow XML file into Kablink Teaming.
6. Create another mapping file, which maps workflow identifiers in Kablink Teaming to Forum workflow names.

7. Run a Tcl script—which uses the second mapping file—to generate an XML file of custom command information.

   Some custom commands are associated with workflow processes. Because of this, the mapping file of workflow information is necessary.

8. Import the custom command XML into Kablink Teaming.

NOTE: This process migrates custom commands created using Forum's UI. It does not migrate template-based custom commands. To migrate template-based custom commands, use the Kablink Teaming entry designer and any necessary JSPs to recreate the command.
