

## Oracle Enterprise Repository

# Repository Extensibility Framework Guide (REX)

---

- **A Note on Product Rebranding**

- As a result of Oracle's acquisition of BEA/Flashline, Inc., the product **Flashline Registry** has been rebranded as **Oracle Enterprise Repository**. The bulk of the REX documentation has been revised to reflect this change. However, in order to avoid confusion, certain instances of the words "flashline" and "registry" are unchanged in this documentation, particularly where they appear in the java package structure and in the REX class names.
- 

## Overview

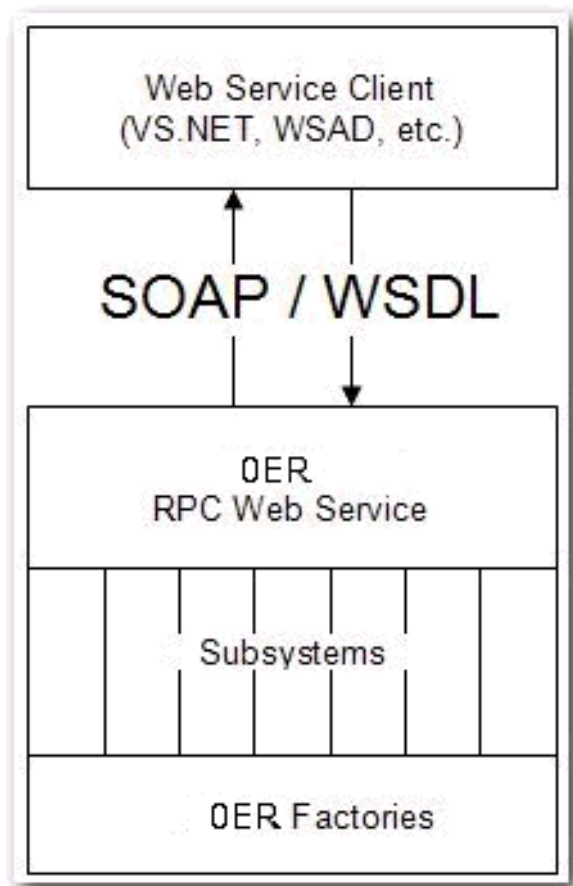
REX is a Web Services API for programmatic integration into Oracle Enterprise Repository. It is based on accepted industry standards, and designed with a focus on interoperability and platform independence. REX uses Remote Procedure Call (RPC) Web Services described by the Web Services Description Language (WSDL v1.1). This allows clients to interact with Oracle Enterprise Repository using any platform and any implementation language that supports Web Services. For example, while Oracle Enterprise Repository is a J2EE application, REX allows programmatic interaction with a .NET client.

If your Oracle Enterprise Repository is or will be configured to be secured by Siteminder, you will need to configure the policy server to ignore (or unprotect) the following URL to allow OpenAPI integration to function properly

- `http://appserver.example.com/oer/services/`

**Note:** Every example provided in the documentation is provided for illustrative purposes and will not necessarily compile due to package structure changes between versions of the Oracle Enterprise Repository WSDL. The user is expected to change the package structure to appropriately target their version of Oracle Enterprise Repository. Full sample code for your version of Oracle Enterprise Repository is provided as part of the install media.

## REX Architecture



The high-level architecture of Oracle Enterprise Repository and REX is designed with several high-level goals in mind:

- **Flexibility**

- Any client platform that conforms to accepted industry standards, such as SOAP, WSDL, and HTTP, can interact with Oracle Enterprise Repository through the REX interface. Proper functioning of the API with the most common client platforms has been validated.

- **Extensibility**

- Oracle Enterprise Repository's layered architecture simplifies the process of adding subsystems to provide access to new features as they are added to Oracle Enterprise Repository. For more information see the section on [Versioning Considerations for REX](#), below.

- **Simplicity**

- End users will find it easy to take advantage of the extensive feature set available in REX.

### ***Subsystems Overview***

Oracle Enterprise Repository's REX provides access to a variety of subsystems. These subsystems loosely group system functionality into logical categories roughly equivalent to the type of entity on which they operate.

Much of this document is organized into sections related to these subsystems.

REX methods are named using a scheme based on the various subsystems. See the section on the [CRUD-Q Naming Convention](#) for a description of the algorithm used in this process. The subsystems defined in REX include:

- acceptableValue
- asset
- assetType
- authToken
- categorization
- categorizationType
- department
- extraction
- import/export
- project
- relationship
- role
- user
- vendor

### ***CRUD-Q Naming Convention***

The scheme used in naming the Open API methods is based on the CRUD-Q mnemonic. CRUD-Q represents five operations:

- **C** - Create
- **R** - Read
- **U** - Update
- **D** - Delete
- **Q** - Query

Each method starts with the name of the subsystem to which it belongs, followed by a description of the operation to be performed within that subsystem, as in the following example:

<subsystem><Operation>

For example, the method to perform a create operation in the asset subsystem would be:

```
assetCreate(...)
```

This naming convention would also produce:

```
assetRead(...)
assetUpdate(...)
assetDelete(...)
assetQuery(...)
```

Subsystems are likely to have operations beyond the CRUD-Q set, and may not include all of CRUD-Q. For example, since it is impossible to delete a user, there is no `userDelete` method. There is, however, a `userDeactivate` method. The following chart provides greater detail.

	Create	Read	Update	Delete	Query	Other Features
<b>Acceptable Value List</b>	Yes	Yes	Yes	Yes	Yes	Accept, Activate, Assign, Deactivate, Register, Retire, Submit, Unaccept, Unassign, Unregister, Unsubmit, Modify Custom Access Settings
<b>Asset</b>	Yes	Yes	Yes	Yes	Yes	
<b>Asset Type</b>	Yes	Yes	Yes	Yes	Yes	
<b>Categorization Type</b>	Yes	Yes	Yes	Yes	Yes	
<b>Department</b>	Yes	Yes	Yes	No	Yes	
<b>Extraction</b>	Yes	Yes	Yes	No	Yes	
<b>Project</b>	Yes	Yes	Yes	Yes	Yes	Close, Open, Reassign extractions, Remove user
<b>Relationship</b>	Yes	Yes	Yes	No	Yes	
<b>Role</b>	Yes	Yes	Yes	Yes	Yes	
<b>User</b>	Yes	Yes	Yes	No	Yes	Activate, Deactivate, Lockout, Unapprove
<b>Vendor</b>	Yes	Yes	Yes	Yes	Yes	
<b>Contact</b>	Yes	Yes	Yes	Yes	Yes	

## Atomicity of Method Calls

Unless otherwise noted, every call to REX is atomic. That is, each call either succeeds completely, or fails completely.

For example, one version of the `categorizationUpdate` method takes as an argument an array of categorization updates. In this case, if one categorization update fails, all categorization updates fail.

## No Inter-call Transaction Support

REX does not currently support inter-call transactions. For example, in the event of an error it is impossible to roll back operations associated with a series of REX calls.

## Technical Details

### Fundamental WSDL Data Types

REX uses the following fundamental WSDL data types, in addition to the complex types defined in the WSDL. Arrays of any of these types may be returned:

- `xsd:int`
- `xsd:long`
- `xsd:string`
- `xsd:boolean`
- `xsd:dateTime`
- `xsd:base64Binary`

Users can dynamically generate API Stubs by consuming the REX WSDL by pointing their IDEs or Web services toolkits at the following URL:

<http://appserver/oer/services/FlashlineRegistry?WSDL>

For example, Java stubs for the Oracle Enterprise Repository REX WSDL may be created using the AXIS WSDL2java utility:

```
java -cp .;axis.jar;xerces.jar;commons-discovery.jar;commons-logging.jar;jaxen-full.jar;jaxrpc.jar;saaj.jar;wsdl4j.jar;xalan.jar org.apache.axis.wsdl.WSDL2Java http://appserver/oer/services/FlashlineRegistry?WSDL
```

The JAR files required to complete this conversion process are:

- `axis.jar`
- `xerces.jar`
- `commons-discovery.jar`
- `commons-logging.jar`
- `jaxen-full.jar`
- `jaxrpc.jar`
- `saaj.jar`
- `wsdl4j.jar`
- `xalan.jar`

- **Note**

- Replace "appserver" in the URL with the name of the server on which Oracle Enterprise Repository is installed.

## Versioning Considerations for the Oracle Enterprise Repository REX

Naturally, the evolution of the Oracle Enterprise Repository REX will parallel the evolution of Oracle Enterprise Repository. However, as a result of this process, incompatibilities may emerge between older and newer versions of REX. While full version compatibility is our goal, backwards-compatibility is subject to unpredictable and therefore potentially unavoidable limitations. In future, Oracle Enterprise Repository releases REX will include the following backwards-compatible enhancements:

- Addition of new methods to the Oracle Enterprise Repository Web service
- Definition of new complex types in the WSDL

With regard to these backwards-compatible changes, the regeneration of client proxies will be necessary only when the need arises to take advantage of new features and functionality.

The namespace of the service will change only when incompatible changes are unavoidable. Examples of such a change would include the modification of an existing complex type, or a change in the signature of a method in the service. In this event, client proxy regeneration will be necessary, as will minimal code changes. Client proxies generated from prior versions of REX will be unable to connect to the new service.

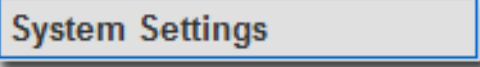
*The namespace of complex types will never change.*

## Basic Concepts

### Getting Started - Enabling the OpenAPI within the Oracle Enterprise Repository

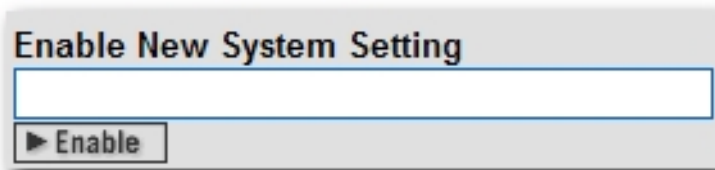
The procedure is performed on the Oracle Enterprise Repository **Admin** screen.

1. Click **System Settings**.



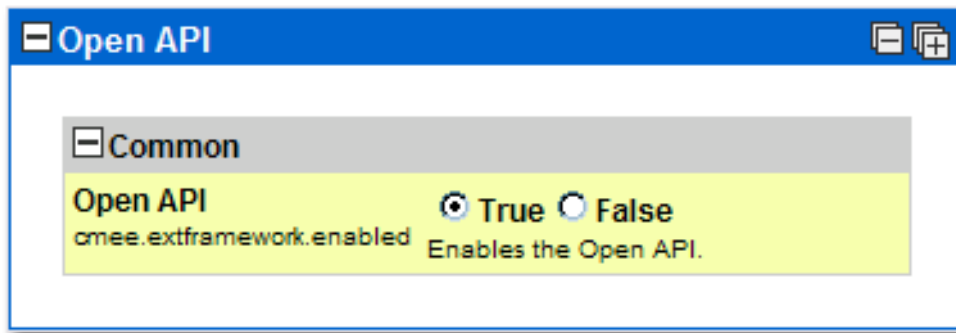
System Settings

2. Enter the property `cmee.extframework.enabled` in the **Enable New System Setting** text box.



3. Click **Enable**.

The **Open API** section opens.



4. Make sure the `cmee.extframework.enabled` property is set to **True**.
5. Click **Save**.

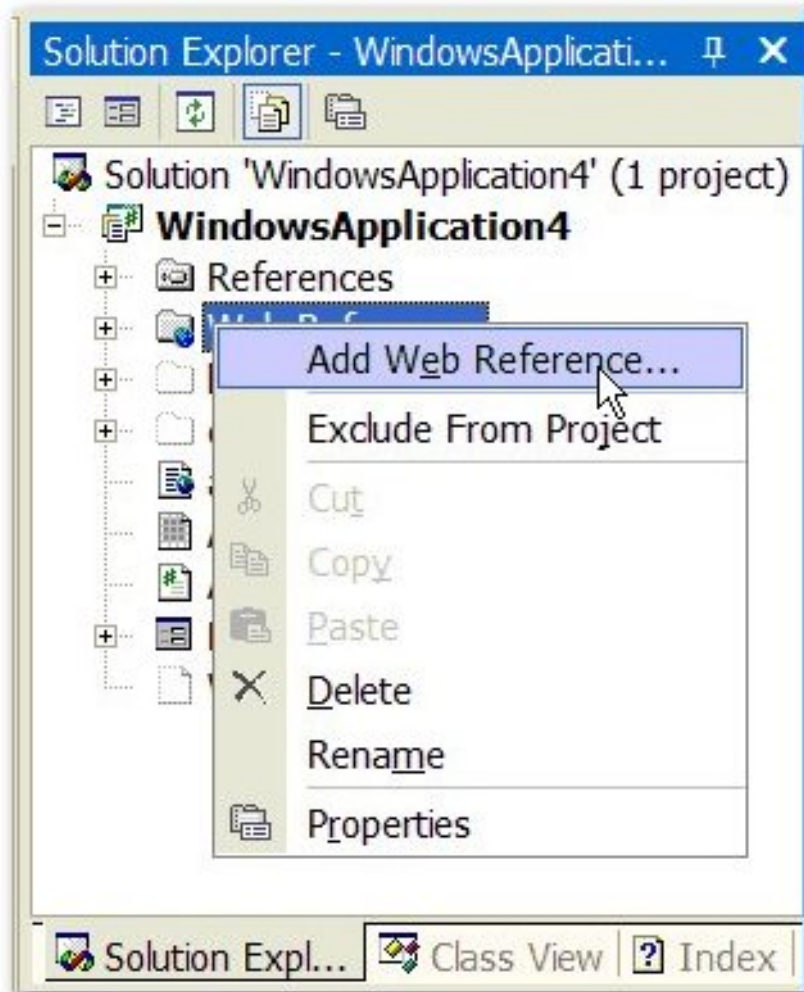
REX is now enabled within your instance of Oracle Enterprise Repository.

## Getting Started - Consuming the WSDL

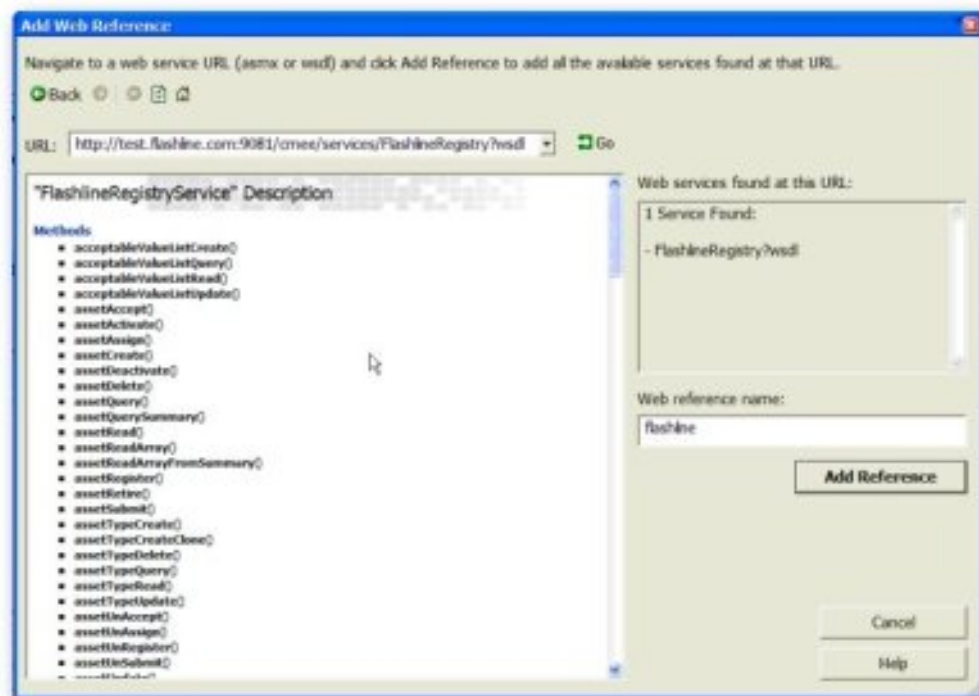
The first step in using REX is to generate the client-side stubs necessary to communicate with the Oracle Enterprise Repository server. This is generally accomplished using the automated tools provided by the specific Web services toolkit in use. This section describes how to generate client stubs using a variety of integrated development environments and toolkits.

### Visual Studio .NET

1. Right-click on the **Web References** node in the Solution Explorer tree.
2. Select **Add Web Reference**



3. Specify the name of the Web reference.





You may load the Oracle Enterprise Repository WSDL file directly from the application server (if it is running) by specifying the proper URL, or from a static file local to your Eclipse project. If you wish to use a URL, it can be found in the following location (replace "yourserver" and "appname" with the appropriate values):

```
http://yourserver:port/appname/services/FlashlineRegistry?wsdl
```

4. Connect to the service endpoint. Once the Web reference has been created, your application can use it by establishing an instance of the client service proxy:

```
flashline.FlashlineRegistryService registry = new flashline.FlashlineRegistryService();
```

5. If the default URL of the service (as contained in the WSDL file used to establish the Web reference) is not the actual address of the Web service, the endpoint address can be changed as follows:

```
registry.Url = "http://appserver/oeo/services/FlashlineRegistry";
```

Your application is ready to interact with Oracle Enterprise Repository via REX.

## Eclipse - Lomboz plugin

The Lomboz plugin will work with Eclipse but any type of Eclipse plugin that provides support for Web Services should work. Most of these tools/plugins will usually ask for:

- The location of the WSDL file
- The source directory from your project in which generated code should live
- The WSDL version with which the WSDL file complies.

The Oracle Enterprise Repository WSDL file can be loaded directly from the application server (if it is running) by specifying the proper URL, or from a static file local to your Eclipse project. The WSDL URL can be found in the following location (replace "www.example.com" and "appname" with the appropriate values):

```
http://www.example.com/appname/services/FlashlineRegistry?wsdl
```

Choose a source directory on your project's build path as the target of the generated client proxy classes.

Oracle Enterprise Repository WSDL conforms to version 1.1 of the WSDL standard.

## Authentication and Authorization

### *Authentication*

The first step in using REX is authenticating with the server. Authentication is performed using the `authTokenCreate` method. This method takes a user ID and password as arguments to be used in authenticating with Oracle Enterprise Repository. If the ID and password are successfully authenticated, an authentication token is returned. This token must be used in every subsequent call to REX.

If a valid `AuthToken` is not included for every REX method, an `OpenAPIException` will be thrown. The applies to all methods except `authTokenCreate` and `authTokenDelete`.

The following example shows how to retrieve an `AuthToken` and use it in subsequent REX calls.

```
package com.example.flashlineclient;

//The imports below are assumed for any of the included examples
import javax.xml.rpc.ServiceException;
import java.net.MalformedURLException;
import java.net.URL;
import java.rmi.RemoteException;
import com.flashline.registry.openapi.service.v300.FlashlineRegistry;
import com.flashline.registry.openapi.service.v300.FlashlineRegistryServiceLocator;
import com.flashline.registry.openapi.base.OpenAPIException;
import com.flashline.registry.openapi.entity.AuthToken;
import com.flashline.registry.openapi.entity.Asset;

public class FlexTest {

    public FlexTest () {

    }

    public static void main(String[] pArgs)throws OpenAPIException, RemoteException, ServiceException {
        try {
            FlashlineRegistry IRegistry = null;
            AuthToken IAuthToken = null;

            URL IURL = null;
            IURL = new URL("http://www.example.com/appname/services/FlashlineRegistry");
            //"www.example.com" should be your server address
            //"appname" is the application name of the location that the Registry is running on
            //These two things must be changed to the proper values in every example

            IRegistry = new FlashlineRegistryServiceLocator().getFlashlineRegistry(IURL);
            IAuthToken = IRegistry.authTokenCreate("username", "password");
            System.out.println(IAuthToken.getToken());
        }
    }
}
```

```

//displaying the authToken as a string to the screen
Asset lAsset = lRegistry.assetRead(lAuthToken, 559);
//reading asset number 559
System.out.println(lAsset.getName());
//displaying the name of asset 559 to the screen

} catch (OpenAPIException lEx) {
    System.out.println("ServerCode = "+ lEx.getServerErrorCode());
    System.out.println("Message   = "+ lEx.getMessage());
    System.out.println("StackTrace:");
    lEx.printStackTrace();
} catch (RemoteException lEx) {
    lEx.printStackTrace();
} catch (ServiceException lEx) {
    lEx.printStackTrace();
} catch (MalformedURLException lEx) {
    lEx.printStackTrace();
}

System.out.println("execution completed");

System.exit(0);
}

}

```

## *Authorization*

REX enforces the same authorization rules as the Oracle Enterprise Repository application. The user ID and password used to authenticate will determine the privileges available to the user via REX. For example, if the authenticated user does not have EDIT privileges assigned for projects, and attempts to create a project using the `projectCreate` REX method, an `OpenAPIException` will be thrown.

## **Exception Handling**

Open API communicates server errors to the client via a SOAP Fault. The manner in which SOAP Faults are handled varies according to the language and SOAP toolkit in use.

This section suggests ways to detect and deal with exceptions generated by the Open API within client code, using the most common platform/toolkit combinations.

### *Java and AXIS*

Exceptions thrown by the Open API are transferred as SOAP Faults, and then de-erialized by the AXIS client toolkit as Java Exceptions. That is, AXIS makes an attempt to map the SOAP Fault to a corresponding client-

side `OpenAPIException` class. Server-side errors are represented to the client as `com.flashline.registry.openapi.OpenAPIException` instances. Consequently, client code can catch exceptions with the code listed below which is from the code above:

```
try {

    IAsset = IRegistry.assetCreate(..);

} catch (OpenAPIException IEx) {
    System.out.println("ServerCode = "+ IEx.getServerErrorCode());
    System.out.println("Message   = "+ IEx.getMessage());
    System.out.println("StackTrace:");
    IEx.printStackTrace();
} catch (RemoteException IEx) {
    IEx.printStackTrace();
} catch (ServiceException IEx) {
    IEx.printStackTrace();
} catch (MalformedURLException IEx) {
    IEx.printStackTrace();
}
```

## *.NET*

Using a consumed Web service in .NET is a bit more complicated. All service exceptions are caught on the client side as exceptions of type `System.Web.Services.Protocols.SoapException`. This makes it somewhat tricky to retrieve the extended information available in the `OpenAPIException` thrown by the Open API.

The .NET `SoapException` property represents the SOAP Fault message. However, the additional fields provided by the `OpenAPIException`, beyond what is explicitly mapped to the standard SOAP Fault, must be obtained by manually parsing the XML Detail property of the .NET `SoapException`. For example, code similar to the following could be used to view the server-side error code and stack trace returned with an `OpenAPIException`:

```
try
{
    registry.testException();
}
catch (SoapException exc)
{
    XmlNode INode = null;

    INode = exc.Detail.SelectSingleNode("*/serverErrorCode");
    if (INode != null)
        Console.Out.WriteLine("Error Code: "+INode.InnerText);
}
```

```

lNode = exc.Detail.SelectSingleNode("*/serverStackTrace");
if(lNode != null)
    Console.Out.WriteLine("Server Stack Trace: \n"+lNode.InnerText);

}

```

It's a good idea to use a more explicit XPath expression than `*/serverErrorCode` in order to eliminate the chance that the returned SOAP Fault includes more than one XML Element with the name `serverErrorCode`.

The following SOAP response illustrates an `OpenAPIException` represented as a SOAP Fault message:

```

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server.userException</faultcode>
      <faultstring>Error [100], Severity [SEVERE]:An unkown server-side error occured.
Please record stack trace (if available) and contact technical support.</faultstring>
      <detail>
        <com.flashline.cmee.openapi.OpenAPIException xsi:type="ns1:OpenAPIException"
  xmlns:ns1="http://base.openapi.registry.flashline.com">
          <message xsi:type="xsd:string">Error [100],
Severity [SEVERE]:An unkown server-side error occured.
Please record stack trace (if available) and contact technical support.</message>
          <serverErrorCode xsi:type="xsd:int">100</serverErrorCode>
          <serverStackTrace xsi:type="xsd:string">java.lang.NullPointerException&#xd;
            at java.util.HashMap.<init>(HashMap.java:214)&#xd;
            ...
            at java.lang.Thread.run(Thread.java:534)&#xd;
        </serverStackTrace>
          <severity xsi:type="xsd:string">SEVERE</severity>
        </com.flashline.cmee.openapi.OpenAPIException>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>

```

## Validation

When attempting to save an entity in Oracle Enterprise Repository, the system will attempt to validate the input. Any missing or invalid data will cause the server to throw an `OpenAPIException` containing a list of

fields and their respective errors. See documentation above regarding exception handling.

## Query Considerations in REX

The **criteria** object model is currently moving to a more flexible representation of terms and grouping. As they occur, these changes will affect the availability of certain API features when executing a query using a criteria object. The subsystems only directly evaluate their corresponding criteria objects, and do not make use of the extended capabilities of the underlying SearchTermGroup, unless otherwise noted in this documentation.

## Sending Binary Data (Attachments)

Various REX methods require sending or receiving potentially large sets of binary data between the client and server. For example, the import/export subsystem provides methods for sending a payload from which to import, and methods for retrieving a payload representing a set of exported assets.

Typically, binary data is transferred via Web services RPC invocations through Dynamic Internet Message Exchange (DIME); SOAP with Attachments (SwA); or Base-64 Encoding. Each has its advantages and disadvantages, but few client toolkits directly support all three.

The Oracle Enterprise Repository OpenAPI supports all three mechanisms for transferring binary data. Details are provided in the following sections.

Any method that provides for the binary transfer of data will have three versions, each one supporting a different transfer mechanism. For example, to retrieve the results of an export, a user can select any one of the following methods:

- `importGetResultsB64`
  - Retrieve results of export in base-64 encoded format. This is the lowest common denominator, and can be used on any platform, provided that the client can encode/decode base-64 data.
- `importGetResultsDIME`
  - Retrieve export results as an attached file, using the DIME protocol. This is the preferred option for most .NET clients.
- `importGetResultsSwA`
  - Retrieve export results as an attached file, using the SOAP with Attachments (SwA) protocol (MIME-based)

### *Using DIME attachments with .NET and the Microsoft Web Services Enhancement (WSE) Kit*

Microsoft provides an extension to the standard .Net Web service toolkit. The Microsoft Web Services Enhancement (WSE) kit provides advanced functionality, such as sending and receiving attachments via Web

services using the Dynamic Internet Messaging Exchange (DIME) protocol.

The following code snippet gives an example of sending data via a DIME attachment:

```
// relax the requirement for the server to understand ALL headers. This MUST be
// done, or the call with the attachment will fail. After the call, if you wish,
// you can set this back to "true"
registry.RequestSoapContext.Path.EncodedMustUnderstand= "false";

// clear the attachments queue
registry.RequestSoapContext.Attachments.Clear();
registry.RequestSoapContext.Attachments.Add(
    new Microsoft.Web.Services.Dime.DimeAttachment("0", "application/zip",
    Microsoft.Web.Services.Dime.TypeFormatEnum.MediaType, "c:\\tmp\\import.zip"));

// start an import running on the server
registry.ImportExecute(lAuthToken, "flashline", null, "FEA Flashpack Import",
    null);

// do some polling (calls to ImportStatus) to monitor the import progress,
// if you wish
```

The following code snippet gives an example of receiving data via a DIME attachment:

```
// relax the requirement for the server to understand ALL headers. This MUST be
// done, or the call with the attachment will fail. After the call, if you wish,
// you can set this back to "true"
registry.RequestSoapContext.Path.EncodedMustUnderstand= "false";

// clear the attachments queue
registry.RequestSoapContext.Attachments.Clear();

// start an export
flashline.ImpExpJob lJob =
registry.ExportExecute(lAuthToken, "flashline", null, "Complete Export", "flashline",
"<entitytypes>
<entitytype type=\\\"acceptableValueList\\\">
<entities>
<entity id=\\\"100\\\"/>
</entities>
</entitytype>
</entitytypes>");

// do some polling (calls to ExportStatus) to watch the progress of the
// export, if you wish...
```

```
// this call will block until either the method returns (or an exception is thrown),
// or the call times out.
registry.exportGetResultsDIME(lAuthToken, lJob);

// check to see if the call resulted in attachments being returned...
if(registry.ResponseSoapContext.Attachments.Count > 0)
{
    Stream lStream = registry.ResponseSoapContext.Attachments[0].Stream;

    // write the data out somewhere...
}
```

### *Using SOAP with Attachments and Java AXIS clients*

The Axis client provides functions to handle SOAP attachments in Java. For more information see: <http://www-106.ibm.com/developerworks/webservices/library/ws-soapatt/>

The following code snippet gives an example of receiving data:

```
byte[] lResults = null;

    ImpExpJob lExportJob =
mFlashlineRegistrySrc.exportExecute(mAuthTokenSrc,"flashline",null,
"Export Assets","default", createAssetQuery().toString());

    lExportJob =
mFlashlineRegistrySrc.exportStatus(mAuthTokenSrc, lExportJob);
    lResults =
mFlashlineRegistrySrc.exportGetResultsB64(mAuthTokenSrc, lExportJob);

// write the results out to disk in a temp file
File lFile = null;

    String lTempDirectory =
System.getProperty("java.io.tmpdir");
    lFile = new File(lTempDirectory + File.separator + "impexp.zip");

    FileOutputStream lOS = new FileOutputStream(lFile);
    BufferedOutputStream lBOS = new BufferedOutputStream(lOS);

    lBOS.write(lResults);
    lBOS.flush();
    lBOS.close();
    lOS.close();
```

The following code snippet gives an example of sending data via a DIME attachment:



```
// open file and attach as data source
InputStream IIS = new FileInputStream(lFile);
    ((Stub)mFlashlineRegistryDest)._setProperty
(Call.ATTACHMENT_ENCAPSULATION_FORMAT, Call.ATTACHMENT_ENCAPSULATION_FORMAT_DIME);
ByteArrayDataSource lDataSource = new ByteArrayDataSource(IIS, "application/x-zip-compressed");
DataHandler lDH = new DataHandler(lDataSource);

// add the attachment
((Stub)mFlashlineRegistryDest).addAttachment(lDH);
ImpExpJob lJob =
mFlashlineRegistryDest.importExecute(mAuthTokenDest, "flashline", null, "Import Assets Test", null);
```