Oracle® Fusion Middleware

Integrating Enterprise Data Quality with External Systems

14*c* (14.1.2.0.0)

F96506-01

December 2024

Oracle Fusion Middleware Integrating Enterprise Data Quality with External Systems, 14*c* (14.1.2.0.0)

F96506-01

Primary Author: Oracle Corporation

Contents

ORACLE

Preface

Describes how to integrate Enterprise Data Quality with external systems and applications.

Audience

This document is intended for advanced users of EDQ and administrators responsible for integrating EDQ with third-party applications.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Accessible Access to Oracle Support

Oracle customers who have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the Oracle Enterprise Data Quality documentation set.

Find the latest version of the EDQ guides and all of the Oracle product documentation at

`https://docs.oracle.com`

Online Help

Online help is provided for all EDQ user applications. It is accessed in each application by pressing the **F1** key or by clicking the Help icons. The main nodes in the Director project browser have integrated links to help pages. To access them, either select a node and then press **F1**, or right-click on an object in the Project Browser and then select **Help**. The EDQ processors in the Director Tool Palette have integrated help topics, as well. To access them, right-click on a processor on the canvas and then select **Processor Help**, or left-click on a processor on the canvas or tool palette and then press **F1**.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Integrating with Subversion

This chapter describes how to integrate and use EDQ with the Subversion version control system.
The following sections are included:

1.1 Software Requirements

EDQ supports integration with all current versions of Subversion. For more information about Subversion, see the Apache Subversion website found at `http://subversion.apache.org/`.

The Subversion server with which EDQ is being integrated must meet these prerequisites:

- Support Hypertext Transfer Protocol (HTTP) and Distributed Authoring and Versioning (DAV) access.
- Require authentication on commit.
- Not require authentication on checkout or update.

When Subversion is integrated with EDQ as a store of configuration information, the following restrictions and limitations apply. Consider the following points before deciding to configure integrated version control using Subversion.

- You cannot update or revert an item that is open in the Director interface or the Subversion server.
- You cannot rename a project once the project is under version control. This is critical in avoiding duplication of reference processor names in a project.
- Deleting a project does not remove it from the Subversion repository.
- Case insensitive name matching is used.

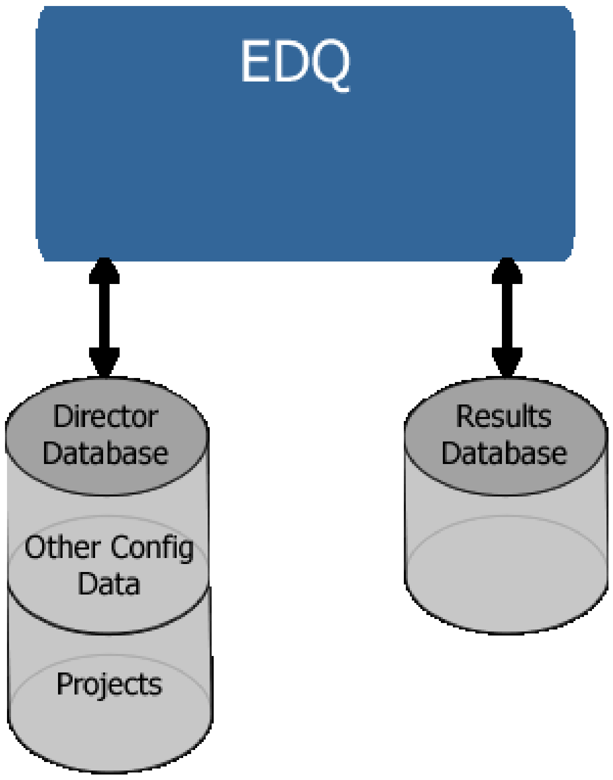
1.2 Understanding the Integration Architecture

The EDQ server can be configured to be aware of a Subversion server as a store of configuration information.

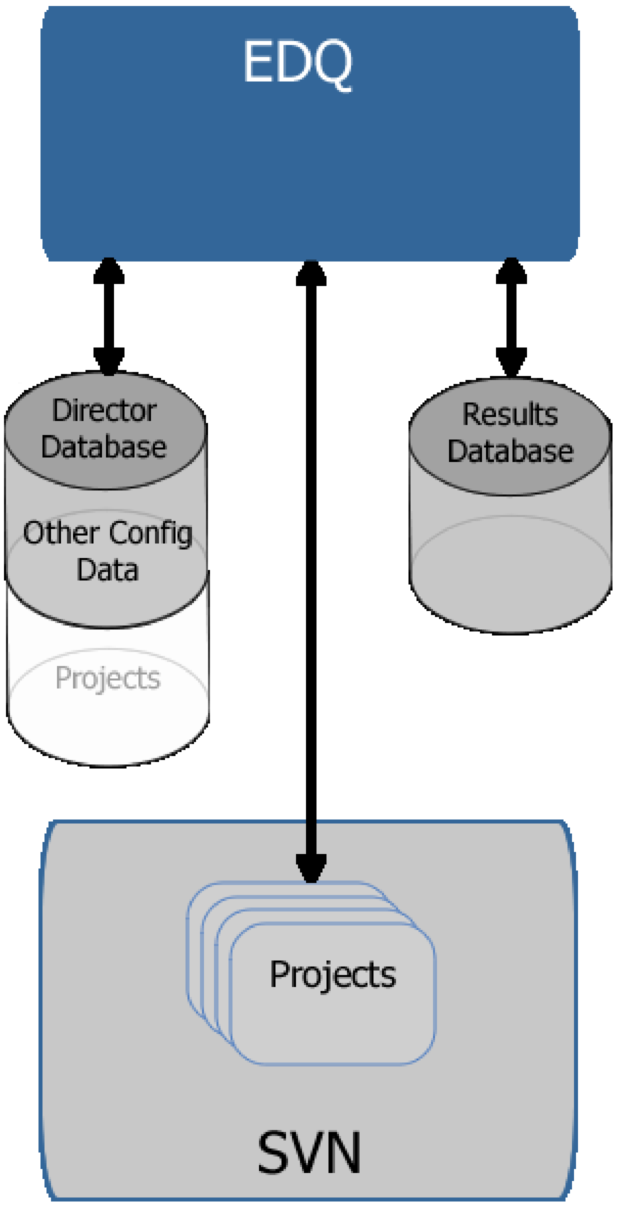
Note:

In this instance, configuration information means information that is managed using the Director UI; for example, projects and system-level data.

In a standard EDQ instance, configuration information, including project information, is stored in the Director database:



The following figure shows an EDQ instance integrated with Subversion:



Note:

The Director database is still required because it contains data derived from the file-mastered configuration that has been normalized to allow querying by the applications.

With EDQ configuration files mastered and stored in a Subversion repository, a Subversion client can be used to commit or otherwise access them. Because EDQ includes an embedded Subversion client, Subversion client operations to control configuration changes can be performed directly in Director once the EDQ integration with Subversion has been enabled.

1.3 Setting Up a Repository

The first stage of configuration is to create a workspace directory where the checked out data is stored:

1. Create a directory on the disk where desired (for example, `C:\MyRepository`) and then add it and commit it to Subversion.
2. Inside the newly created directory, set the following Subversion property:

   ```
   svn propset edq:systemversion 12.1.3:base .
   ```

   > **Note:**
   >
   > Set Subversion property to "12.1.3:base" and not to the current version of EDQ.

3. Commit these changes into Subversion. Your workspace now displays these properties:

   ```
   svn proplist -v .
   Properties on '.':
     edq:systemversion
       12.1.3:base
   ```

4. Create the following subdirectories in the newly created directory:
   - Data Stores
   - Hidden Reference Data
   - Images
   - Projects
   - Published Processors
   - Reference Data
5. Add and commit these directories. The repository is now set up correctly for EDQ.

The preceding steps only need to be performed once per repository. All remaining changes can be made using EDQ.

1.4 Configuring EDQ with Subversion

Subversion must be integrated with a fresh installation of EDQ.

> **Caution:**
>
> When an EDQ instance is integrated with Subversion, all pre-existing and other configuration information is lost. To retain this information, you must package and export it first. For further details, see Retaining Existing Configuration Information.

> **Note:**
>
> Oracle recommends that a single workspace be assigned to each instance of EDQ because it is difficult to move between workspaces in a single EDQ instance.

1.4.1 Configuring a New EDQ Installation

To configure a new EDQ installation:

1. Shut down the application server.
2. Check out the workspace from Subversion. It is not necessary to checkout the whole tree; just the workspace directory itself is required.
3. Edit the `director.properties` file in the *`ORACLE_HOME`*`/user_projects\domains\domains\edq_domain\edq\oedq.local.home` directory.
4. Add the following line replacing the directory path with that of the absolute path to your root workspace directory. For example:

   ```
   sccs.workspace = C\:\\MyRepository
   ```

   > **Note:**
   >
   > This example demonstrates the need to escape colon (:) and backslash (\) characters in the path with a backslash. You must also escape space characters in the path with a backslash.

5. Start your EDQ server, and then start Director.
6. Check the top of the `Main0.log` file for an `INFO` message listing the name of the SCCS workspace you added. For example:

   ```
   INFO: 02-Sep-2013 10:05:21: SCCS workspace is C:\MyRepository
   ```

7. If no errors follow this message, EDQ is configured to use Subversion. If there are errors, see Troubleshooting Errors, for possible solutions.

1.4.2 Retaining Existing Configuration Information

As previously stated, Subversion must be integrated with a fresh installation of EDQ. Therefore, any pre-existing projects and other configuration items in an EDQ installation must be packaged before integration begins and then imported to the new installation afterwards:

1. Package all configuration items in the current EDQ instance into DXI files.
2. Install a new instance of EDQ with the Subversion integration enabled.
3. Import the DXI files into the new instance, and commit the files to the Subversion workspace.
4. Check that the configuration items are all valid and working correctly.

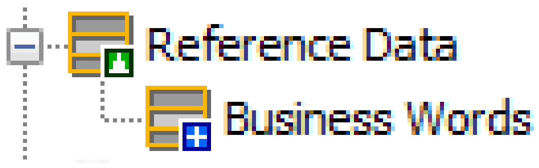
   Note that all passwords for Data Stores must be re-entered after a configuration import.
5. Decommission the previous instance.

1.5 Understanding the Integration Elements

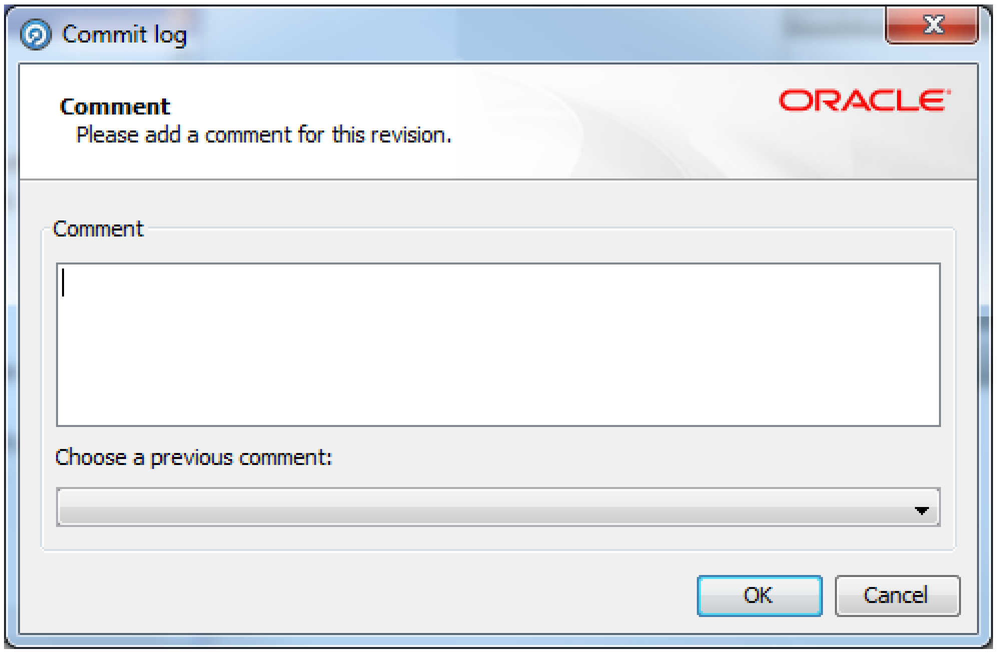
Once EDQ is integrated with Subversion enabled, the following interface elements become visible within the Director application:

- Subversion status icon overlays in Project Browser - There are two icons used to indicate the three possible Subversion statuses of nodes in the Project Browser:
  - No icon - The node (and its sub-nodes) are all up to date.
  - Green icon- This node (and its sub-nodes) have modifications.
  - Blue icon - This node (and its sub-nodes) is new/currently not under Version Control.

  For example, the following image shows both icons in use. The Reference Data node is modified (green icon) as one of its sub-nodes has changed. A new piece of Reference Data, Business Words, has been added, and is marked with the blue icon:



- Version Control tab - The Properties dialog (displayed by right-clicking on an item in the Project Browser and selecting **Properties**) now contains a Version Control tab that describes the state of the item, when it was last updated, its Subversion revision, and whether it is current.
- New context menu for Version Control - The Project Browser right-click menu now contains a Version Control option. When selected, this displays a sub-menu with Subversion options to update, commit, revert, compare or view the log for the item. These options are recursive. For example, if you perform View Log on a single process then you will see the log for this process only, but if you perform View Log on the Processes node you will see changes for all processes.
- Comment and credentials dialogs on commit - When you commit changes to the repository, Director displays the Commit log dialog:



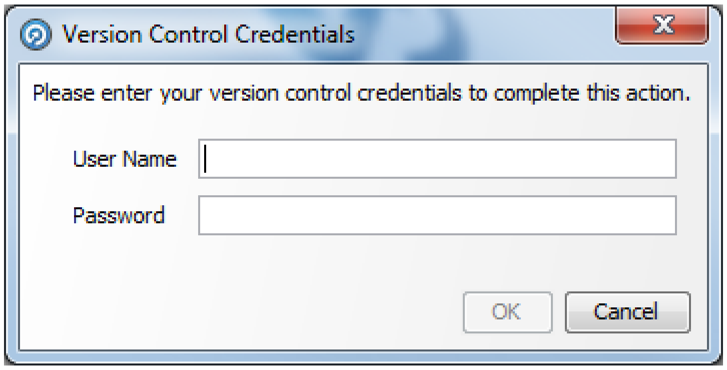
Comment
Please add a comment for this revision.

Comment

Choose a previous comment:

In this dialog you can enter a comment describing the change in the Comment field. Alternatively, you can automatically populate the field by choosing a comment from the list of comments previously entered in the current session.

After you click **OK** in the Commit log dialog, Director displays the Version Control Credentials dialog if you have not already provided your credentials in the current session:



Please enter your version control credentials to complete this action.

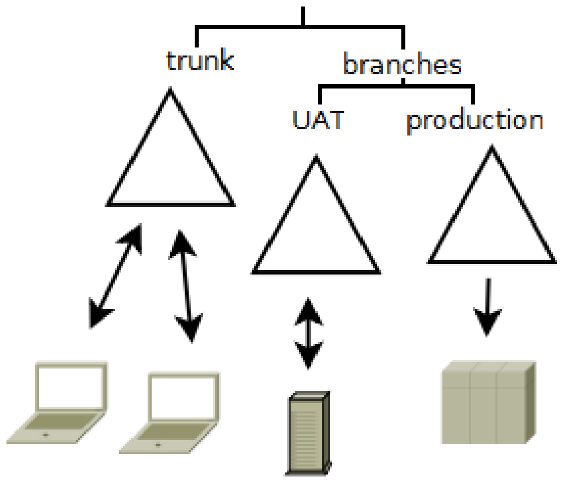
User Name

Password

In this dialog you enter your user name and password for the Subversion repository and then click OK.

1.6 Reviewing a Deployment Example

An example deployment is presented here. In this illustration, there is a single Subversion server that holds three copies of the configuration for four EDQ installations:



The copies of the configuration are:

- **trunk** - the traditional location that all development work is performed on. New features of the configuration are developed and saved here.
- **branches and UAT** - this branch represents the copy of the configuration under UAT testing.
- **branches and production -** this branch represents the production copy of the configuration.

The four EDQ installations using the Subversion server for storing their configuration are:

- Two development laptops where design work and maintenance of existing projects are carried out.
- A UAT server for User Acceptance Testing changes.
- A production server for production runs.

In this example deployment, the laptop users develop configuration for individual projects on their own laptops and then commit changes back to the subversion repository on "trunk". Where the developers are co-operating on developing a project they will periodically update their local installation to pick up changes from the other developer.

At some point development reaches a point where it needs to be released to UAT for testing. A release manager then copies the necessary projects from "trunk" to "UAT" on the subversion server.

For example, the following Subversion command may be used:

```
svn cp -m"Release Project X to UAT" http://svn/repos/config/trunk/ProjectX http://svn/
repos/config/branches/UAT
```

The test manager then updates the UAT server's projects to load the new configuration into the EDQ server. Over a period of time testing continues. As issues are found they are fixed in the UAT environment and committed back to the subversion repository.

Once UAT environment has achieved an acceptable test level it is promoted to release. This achieved in much the same way as the release from development to UAT. The necessary projects are copied across in the version control repository and then the production server is updated to use this configuration.

1.7 Troubleshooting Errors

You may encounter the following errors for which the cause and solution is provided.

Error	Cause and Solution
`Configuration database is not compatible with workspace`	The database has been used with a different workspace. This error usually arises occurs when operations have been performed in EDQ before Subversion version control is enabled.There are two solutions: drop and recreate the Director database or reinstall EDQ.
`Unable to open an ra_local session to URL`	This may occur when trying to commit files to an invalid repository. The EDQ integration is not compatible with file-based repositories (those repositories beginning with `file:///` or `C:\example`). A fully declared http:// path to the repository must be made.

2

Integrating with Git

This chapter describes how to integrate and use EDQ with the Git version control system. Consider the following points before deciding to configure integrated version control using Git.

- Project and global objects are stored in the file systems, but Software Configuration Management (SCM) operations such as commits, pushes and pull requests are performed manually outside of EDQ.
- The EDQ system must be freshly initialized. The Git configuration must be completed before you start the system for the first time.

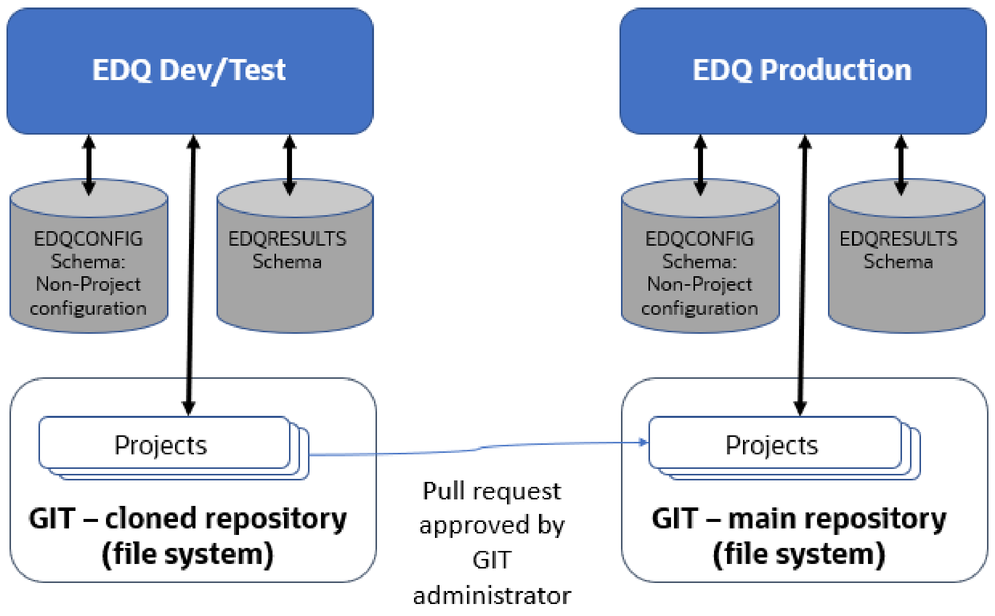
> **Note:**
>
> If you have an existing EDQ system with configuration that you want to convert to using Git, you should package all projects on this system and back up its local home. You can then import the projects and any configuration extensions or other needed files in the local home onto a Git-integrated system.

The following sections are included:

2.1 Understanding the Integration Architecture

The EDQ server can be configured to be aware of a Git server as a store of configuration information.

The following figure illustrates a typical setup of two EDQ instances that are integrated with Git. Here the Dev/Test instance is used to make and test changes. The tested changes are then promoted onto a main branch used by a Production instance by a pull request that is approved by a Git administrator. Note that the Dev and Test instances could equally be separate. For example, they may both work on the same fork, but with changes only moving to Test once committed and pushed from Dev.


EDQ Dev/Test
EDQ Production
EDQCONFIG Schema: Non-Project configuration
EDQRESULTS Schema
EDQCONFIG Schema: Non-Project configuration
EDQRESULTS Schema
Projects
GIT – cloned repository (file system)
Pull request approved by GIT administrator
Projects
GIT – main repository (file system)

2.2 Preparing the Git Workspace

The first stage of configuration is to create a workspace directory where the checked out data is stored:

1. Create a new empty directory in a Git workspace and create a `.wsprops` file that includes this single line:

   ```
   systemversion=12.1.3:base
   ```

   > **Note:**
   >
   > Set the property to "12.1.3:base" and not to the current version of EDQ. This is the correct value for all versions of EDQ 12.2.*x*.

2. Create the following subdirectories in the newly created directory:
   - Data Stores
   - Hidden Reference Data
   - Images
   - Projects
   - Published Processors
   - Reference Data

3. In each directory, create an empty `.gitignore` file to ensure that the directories can be committed to Git successfully. You need to do this because you cannot commit empty directories to Git. To create and populate the directories, use the following script:

   ```
   rootdirs="Data_Stores Hidden_Reference_Data Images Projects Published_Processors
             Reference_Data"
   # Create root dirs
   ```

```
for i in $rootdirs
do x=$(echo $i | tr '_' ' ')
   mkdir "$x"
   echo > "$x/.gitignore"
   echo created $x
done
cat > .wsprops <<EOF
systemversion=12.1.3:base
EOF
```

4. Add and commit these directories to Git. The repository is now set up correctly for EDQ.

2.3 Configuring EDQ with Git

Git must be integrated with a fresh installation of EDQ. Before you start the EDQ server for the first time, edit `director.properties` and add these lines:

```
sccs.workspace = file system path to root directory
sccs.vcs.type  = null
```

Here, the root directory in the first line is the directory where you created the repository in Preparing the Git Workspace. The second line disables the integrated Subversion support.

After you have edited `director.properties`, start the EDQ server.

The built-in reference data objects will be created in the root Reference Data folder. Commit these objects to Git and push them to the origin server.

2.4 Using EDQ

After the Git configuration is complete, you can use the EDQ Director client to create and edit global and project objects as normal. The objects are stored in the file system location set by the `sccs.workspace` property in Configuring EDQ with Git.

To make changes visible to other users of the Git repository, commit and push the changes using the standard Git command line tools. For example:

```
$ cd /opt/git/repo/dev
$ git add Projects/test1
$ git commit -m 'Committing a project'
$ git push
```

After the changes have been pushed to the origin server, other users can clone or pull from the repository to work on the same objects.

To update the local workspace with changes created by other users, use a `git pull` command:

```
git pull
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 13 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (13/13), done.
...
```

If the local EDQ server is running, rescan the workspace for changes. To do this, use the `jshell.jar` utility to run the following script:

```
$ java -jar jshell.jar scripts/sccs/scan.groovy \
          -server host:8090 -user username -pw password
```

The Project Browser of running EDQ clients reflect the changes that are picked up by the script.

Note that the `scan.groovy` script may update process and job objects that are open in EDQ clients. However, the Canvas does not reflect these changes. You must close and reload the objects to see the changes. A good practice is to ensure that no clients or jobs are running when a scan is run. After you execute any Git operation that changes the workspace contents, such as switching branches, make sure that you run the `scan.groovy` script or restart the EDQ server.

3

Using the Command Line Interface

This chapter describes how to use the command line interface.
This chapter includes the following sections:

- Running the Command Line Interface
- Understanding the Commands and Arguments
- Reviewing Examples

The command line interface, `jmxtools.jar`, provides access to a number of facilities.

3.1 Running the Command Line Interface

The command line interface is distributed as a self contained `.jar` file in the tools directory, and is executed by the following command line invocation:

```
java -jar jmxtools.jar commandname arguments
```

The commands and arguments are described in the following section.

3.2 Understanding the Commands and Arguments

The command line interface can run a number of commands and provides functionality including:

- Running jobs
- Listing and dropping orphaned results tables
- Showing user session logs
- Shutting down real-time jobs
- Checking the EDQ version number

The following sections provide a full guide to the commands, arguments and options available.

EDQ also provides support for jobs through the REST-based EDQ Configuration API interfaces. For details on using these interfaces for performing various jobs, see "REST Interfaces for Jobs" in the "*Integrating Enterprise Data Quality With External Systems*" guide.

3.2.1 `runjob`

The `runjob` command runs a named job in the same way as if running the job using the Director UI. The `runjob` command takes the following arguments:

Argument	Use
`-job` *`job_name`*	Specifies the name of the job to run.
`-project` *`project_name`*	Specifies the name of the project that contains the job .

Argument	Use
-u *user_name*	Specifies the user name to use to connect to the EDQ server. The user must have permission to run jobs and must have permission to the project containing the job.
-p *password*	Specifies the connecting user's password. If the -p option is not set, EDQ will prompt the user for the password.
-nolockwait	Indicates that if any of the resources used by the job are locked, the job should not wait for them to become available. Instead, it should terminate with a failure code and return control to the command line. The `-nolockwait` argument takes no extra values.
-nowait	Indicates that the command line should not wait for the job to complete. The `-nowait` argument takes no extra values.
server:port	Specifies the server and port of the JMX (management) interface.

3.2.2 runopsjob

The `runopsjob` command runs a named job in the same way as if running the job using the Server Console user interface. This provides additional functionality to the `runjob` command, specifically the use of Run Labels and Run Profiles. Run Labels may be used to store results separately from other runs of the same job. Run Profiles may be used to override externalized configuration settings at runtime.

The `runopsjob` command takes the following arguments:

Argument	Use
-job *job_name*	Specifies the name of the job to run.
-project *project_name*	Specifies the name of the project that contains the job
-u *user_name*	Specifies the user name to use to connect to the EDQ server. The user must have permission to run jobs and must have permission to the project containing the job.
-p *password*	Specifies the connecting user's password. If the `-p` option is not set, EDQ will prompt the user for the password.
-nolockwait	Indicates that if any of the resources used by the job are locked, the job should not wait for them to become available. Instead, it should terminate with a failure code and return control to the command line. The `–nolockwait` argument takes no extra values.
-nowait	Indicates that the command line should not wait for the job to complete. The `-nowait` argument takes no extra values.
-runlabel *run_label_name*	Specifies the name of the run label under which you wish to store staged output results. Note that this will override any run label that is specified in a run profile or by `-D runlabel` = *run_label_name*.
-props *run_profile_name*	Specifies the full path to a run profile properties file containing override settings for externalized configuration options in the job.
-D *externalized_option=value*	Allows you to override specific externalized options for the job individually. The syntax for the externalized options and values is the same as used in run profile properties files. Note that characters is interpreted by the command line, so some characters will need to be escaped according to the shell conventions of your environment. Also note that any individually specified externalized option settings will override any settings for the same option if these are specified in a run profile used in the same run.

Argument	Use
server:port	Specifies the server and port of the JMX (management) interface.

3.2.3 droporphans

The `droporphans` command is used to remove any orphaned results tables that may be created when processes are terminated unexpectedly. It should not be run when any jobs or processes are running on the EDQ server.

The `droporphans` command takes the following arguments:

Option	Use
`-u` *user name*	Specifies the user name to use to connect to the server. The user must have permission to cancel jobs and must have permission to the project containing the job.
`-p` *password*	Specifies the connecting user's password. If the `-p` option is not set, will prompt the user for the password.
server:port	Specifies the server and port of the JMX (management) interface.

3.2.4 listorphans

The `listorphans` command is used to identify any orphaned results tables. The `listorphans` command takes the same arguments as the `droporphans` command.

3.2.5 scriptorphans

The `scriptorphans` command creates a list of SQL commands for dropping orphaned results tables. This is useful if you want to review exactly which commands will run on the Results database when you drop tables, or if you want to drop the tables yourself manually.

3.2.6 list

The `list` command lists all the available commands.

3.2.7 showlogs

The `showlogs` command starts a small graphical user interface application that allows user session logs to be retrieved.

3.2.8 shutdown

The `shutdown` command shuts down all real-time jobs. These are jobs that are running from real-time record providers (web services or Java Message Service).

The `shutdown` command takes the following arguments:

Option	Use
`-u` *`user name`*	Specifies the user name to use to connect to the server. The user must have permission to cancel jobs and must have permission to the project containing the job.
`-p` *`password`*	Specifies the connecting user's password. If the `-p` option is not set, EDQ will prompt the user for the password.
`-nowait`	Indicates that the command line should not wait for the job to complete. The `-nowait` argument takes no extra values.
`server:port`	Specifies the server and port of the JMX (management) interface.

3.2.9 `version`

The `version` command is used to identify the version of the currently installed instance of .

Enter the following at the command line:

```
java -jar jmxtools.jar version
```

The version number is returned.

3.3 Reviewing Examples

This section lists several possible invocations of the command line interface:

- Listing All the Available Commands
- Listing the Available Parameters for a Command
- Running a Named Job
- Running a Named Job in Operations Mode

3.3.1 Listing All the Available Commands

The following invocation of the command line interface lists all of the available commands:

`java -jar jmxtools.jar -list`

The output is as follows:

```
Available launch names:
<Job tools>
runjob Run named job
shutdown Shutdown realtime jobs
runopsjob Run named job in operations mode

<Logging>
showlogs Show session logs

<Database Tools>
listorphans List orphaned results tables
droporphans Drop orphaned results tables
scriptorphans Create script for dropping orphaned results tables

<System Information>
version Display version number of tools
```

3.3.2 Listing the Available Parameters for a Command

If the command line interface is invoked by specifying a command without the corresponding parameters, it outputs detailed help for the command. For example, to get detailed help on the `runjob` command, invoke the command line interface as follows:

```
java -jar jmxtools.jar runjob
```

The output is as follows:

```
Usage: runjob -job jobname -project project [-u user] [-p pw] [-nowait] [-nolockwait] [-
sslprops props | -ssltrust store] server:port
```

3.3.3 Running a Named Job

This example illustrates how to run a named job in a named project on a specific instance (as specified by machine name and port).

To run a job called "rulecheck" in a project called "Audit" on the local machine with a JMX server on port 8090 using a user named "dnadmin", the command is as follows:

```
java -jar jmxtools.jar runjob -job rulecheck -project audit -u dnadmin
localhost:8090
```

The application prompts the user to enter the password for the `dnadmin` user.

3.3.4 Running a Named Job in Operations Mode

This example illustrates how to run a named job in 'operations mode' in a Windows environment. In operations mode, there is access to the Run Label and Run Profile capabilities so that the configuration of the job can be specified dynamically, and so that the results of the job can be stored by Run Label.

To run a job called "profiling" in a project called "MDM" on a server called "prod01", with a run label of "Nov2011" and a run profile file called `File1.properties`, with a JMX server on port 8090, the command is as follows:

```
java -jar jmxtools.jar runopsjob -job profiling -project MDM -runlabel Nov2011 -
props c:\ProgramData\Oracle\"Enterprise Data
Quality\oedq_local_home\File1.properties" -u dnadmin prod01:8090
```

4

Configuring Additional Database Connections

This chapter describes how you can configure additional database connections for use in Director.

- Using JNDI to Connect to Data Stores
- Connecting to an Oracle Database Using tnsnames.ora
- Connecting to an Oracle Database Using Oracle Internet Directory (LDAP)

The standard options for Director to connect to data stores are described in the online help. Once implemented, these options appear in the Data Store Configuration step of the New Data Store wizard in Director. For help with using this wizard, see the Director online help.

4.1 Using JNDI to Connect to Data Stores

You can configure EDQ to use a Java Naming and Directory Interface (JNDI) data store connection.

1. Define the JNDI data store. JNDI is provided by the hosting application server. For more information about defining JNDI data sources in Oracle WebLogic Server, see "Using DataSource Resource Definitions" in .
2. In the EDQ data store wizard, specify JNDI as the type of data store, and then specify the JNDI name.

4.2 Connecting to an Oracle Database Using tnsnames.ora

You can configure EDQ to use an Oracle Transparent Network Substrate (TNS) data store connection. To use this connection method, you specify a name from a `tnsnames.ora` file as the data source when using the data sources wizard. Only the `tnsnames.ora` file is needed. No other Oracle client software is needed.

4.2.1 To Configure EDQ to Connect Through TNS

To connect EDQ through TNS:

1. Set the `oracle.net.tns_admin` Java system property to a local directory that contains the `tnsnames.ora` file.
2. Create a file named `jvm.properties` in your EDQ local configuration directory (`oedq_local_home` by default) and add an entry similar to the following: `oracle.net.tns_admin = c:\\temp`). This property may have been set already in the application server when EDQ was installed.

For more information about the `tnsnames.ora` file, see "Configuring the Local Naming Method" in .

4.3 Connecting to an Oracle Database Using Oracle Internet Directory (LDAP)

You can configure EDQ to use an Oracle Lightweight Direct Access Protocol (LDAP) data store connection by setting the required Java system properties. These properties are:

```
dn.oracle.directory.servers = ldap://servername:port
```

```
dn.oracle.default.admin.context = dc=domaincontext1,dc=domaincontext2
```

The first property gives the location of your LDAP servers. The second property sets the context within the LDAP tree. Together, these properties enable EDQ to construct an Oracle and LDAP JDBC connection string, which looks similar to:

```
jdbc:oracle:thin:@ldap://servername:port/
unicode,cn=Oraclecontext,dc=domaincontext1,dc=domaincontext2
```

5

Configuring EDQ to Process XML Data Files

This chapter describes how can be configured to read and write XML data files.
This chapter includes the following sections:

- Using Simple XML Data Stores
- Using XML and Stylesheet Data Stores

You can use XML data files in snapshots to read and write the data contained in the file. A snapshot is a staged copy of data in a data store that is used in one or more processes. provides two types of data stores for working with XML data files: Simple XML and XML and Stylesheet. Both are available for server-side and client-side data stores.

5.1 Using Simple XML Data Stores

Simple XML data stores can read and write XML files that have a simple 2-level structure in which the top level tag represents the entity and the lower level tags represent the attributes of the entity. XML files exported from Microsoft Access are an example.

Following is an example of a simple XML file format that could be used with :

```
<dataroot>
  <Person>
    <Id>1</Id>
    <FirstName>Fred</FirstName>
    <LastName>Bloggs</LastName>
    <DateOfBirth>1972-01-31T00:00:00.000+0000</DateOfBirth>
    <Weight>85</Weight>
  </Person>
  <Person>
    <Id>2</Id>
    <FirstName>Jane</FirstName>
    <LastName>Smith</LastName>
    <DateOfBirth>1985-07-16T00:00:00.000+0100</DateOfBirth>
    <Weight>63</Weight>
  </Person>
</dataroot>
```

5.1.1 Reading Simple XML Files

When reads Simple XML files the following occurs:

- The root element name is not used, so it can be anything.
- The record element name appears as the table name in the Table Selection page of the Snapshot Wizard dialog.

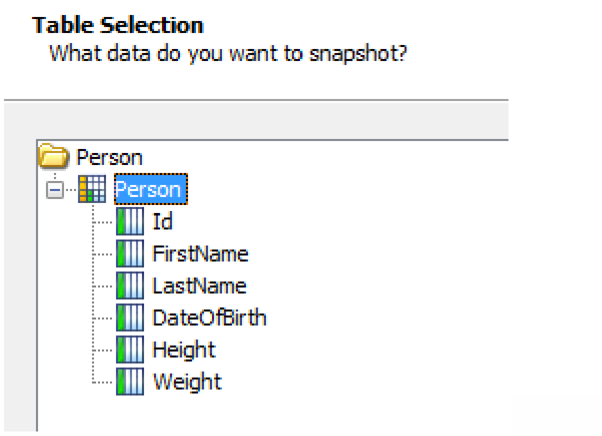
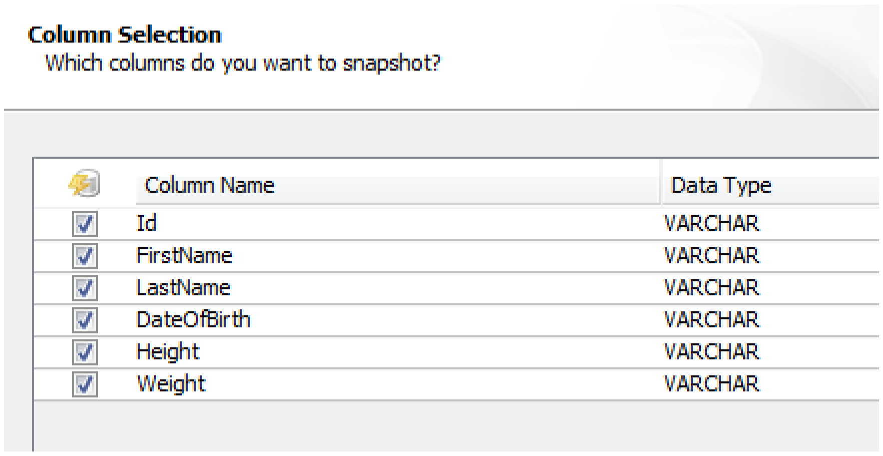


Table Selection
What data do you want to snapshot?

- Person
  - Person
    - Id
    - FirstName
    - LastName
    - DateOfBirth
    - Height
    - Weight

- The lower level element names appear as the column names in the Column Selection page of the Snapshot Wizard and therefore become EDQ attribute names.



Column Selection
Which columns do you want to snapshot?

	Column Name	Data Type
✓	Id	VARCHAR
✓	FirstName	VARCHAR
✓	LastName	VARCHAR
✓	DateOfBirth	VARCHAR
✓	Height	VARCHAR
✓	Weight	VARCHAR

5.1.2 Writing Simple XML Files

When generating Simple XML files using an export to the data store, the name of the data store defines the record XML element name. The element `Person` in the example in Using Simple XML Data Stores shows how this appears in the XML.

The XML element names of the lower level tags are taken from the attribute names. EDQ names are encoded to ensure that invalid XML is not generated. For example, space characters in names are replaced by the character sequence `_x0020_`, so an attribute named `Date Of Birth` would generate XML elements in the following format:

```
<Date_x0020_Of_x0020_Birth>
```

5.2 Using XML and Stylesheet Data Stores

When there is a requirement to work with XML of a different structure than that of Simple XML, then you use the XML and Stylesheet data stores.

These data stores read and write XML conforming to the DN-XML schema and optionally allow the use of a custom stylesheet to:

- Transform XML from a custom XML format to DN-XML during data snapshot

- Transform XML from DN-XML to a custom XML format during data export

For more information about XML stylesheets, see the W3C website found at `http://www.w3.org/Style/XSL/` and `http://www.w3.org/standards/xml`.

5.2.1 Using DN-XML

DN-XML is the format by which custom XML can be processed by .

An example of DN-XML is as follows:

```
<dn:data xmlns:dn="http://www.datanomic.com/2008/dnx">
  <dn:record skip="true">
    <dn:value name="Id" type="string"/>
    <dn:value name="FirstName" type="string"/>
    <dn:value name="LastName" type="string"/>
    <dn:value name="DateOfBirth" type="date"/>
    <dn:value name="Height" type="number"/>
    <dn:value name="Weight" type="number"/>
  </dn:record>
  <dn:record>
    <dn:value name="Id">1</dn:value>
    <dn:value name="FirstName">Fred</dn:value>
    <dn:value name="LastName">Bloggs</dn:value>
    <dn:value name="DateOfBirth">1972-01-31</dn:value>
    <dn:value name="Height">1.85</dn:value>
    <dn:value name="Weight">85</dn:value>
  </dn:record>
  <dn:record>
    <dn:value name="Id">2</dn:value>
    <dn:value name="FirstName">Jane</dn:value>
    <dn:value name="LastName">Smith</dn:value>
    <dn:value name="DateOfBirth">1985-07-16</dn:value>
    <dn:value name="Height">1.65</dn:value>
    <dn:value name="Weight">63</dn:value>
  </dn:record>
</dn:data>
```

This is the equivalent DN-XML for the example given in Using Simple XML Data Stores.

Note that the attribute names are defined differently in DN-XML compared with Simple XML. Because DN-XML uses attribute content to specify attribute names, it is possible to create attributes with spaces and other special characters in their names.

In the previous example, the `<dn:record skip="true">` XML element and its contents allows the definition of the structure of the source including the field names and their data types. All other record elements define a row of data in . This is analogous to the header row in a comma-separated values file. The following data types are permitted:

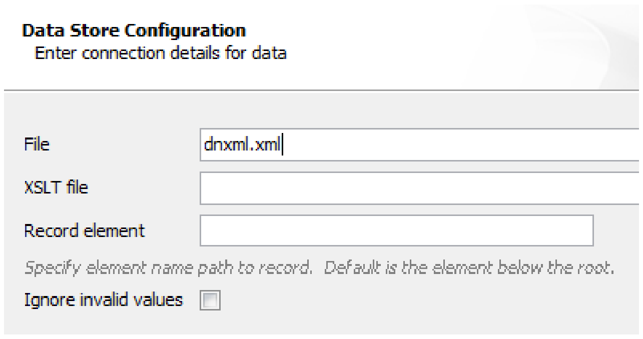
- string
- date
- number

Note:

Date values in DN-XML files should be specified in the XSD date format (ISO 8601). For example, '2008-10-31T15:07:38.6875000-05:00' or without the time component simply as '2008-10-31'.

Within a data record, value elements are used to specify attribute values for the record. The name attribute is used to specify the attribute in question and the text content of the attribute specifies the value for that attribute. For example, the XML fragment, `<dn:value name="FirstName">Fred</dn:value>`, assigns the value 'Fred' to the attribute 'FirstName'.

DN-XML files can be read in to by creating an XML and Stylesheet data store and specifying the location of the XML source file; the XSLT file option should be left blank:



Data Store Configuration
Enter connection details for data

File	dnxml.xml
XSLT file	
Record element	

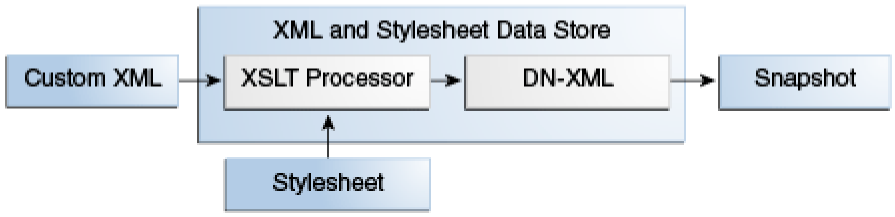
Specify element name path to record. Default is the element below the root.

Ignore invalid values ☐

Similarly, can write DN-XML files by exporting data to an XML and Stylesheet data store with the XSLT option left blank.

5.2.2 Reading Custom XML Files

XML files in custom formats can be read by using the XML and Stylesheet data store configured to use a custom XML stylesheet (XSLT) to transform from the custom schema to the DN-XML schema during data snapshotting.



Following is an example custom XML file that could be read into :

```
<crmdata>
  <contacts>
    <contact id="1">
      <name>
        <firstname>Fred</firstname>
        <surname>Bloggs</surname>
      </name>
      <dob>1972-01-31</dob>
      <properties>
        <property name="height" value="1.85"/>
        <property name="weight" value="85"/>
      </properties>
    </contact>
    <contact id="2">
      <name>
        <firstname>Jane</firstname>
        <surname>Smith</surname>
      </name>
      <dob>1985-07-16</dob>
      <properties>
        <property name="height" value="1.68"/>
        <property name="weight" value="63"/>
      </properties>
    </contact>
  <contacts>
</crmdata>
```

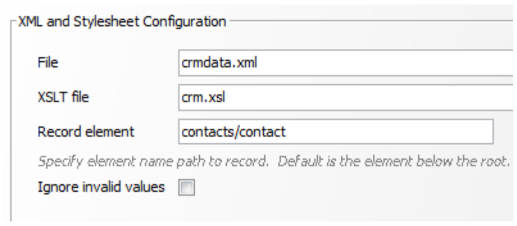
The following XML stylesheet demonstrates one way that the preceding example custom XML can be transformed into a suitable DN-XML format:

```
<xsl:stylesheet version="1.0" xmlns:dn="http://www.datanomic.com/2008/dnx"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/02/xpath-functions">

  <xsl:output method="xml"/>

    <xsl:template match="/">
        <dn:data>

         <!-- Write out the header record -->
         <dn:record skip="true">
             <dn:value name="Id" type="string"/>
             <dn:value name="FirstName" type="string"/>
             <dn:value name="LastName" type="string"/>
             <dn:value name="DateOfBirth" type="date"/>
             <dn:value name="Height" type="number"/>
             <dn:value name="Weight" type="number"/>
         </dn:record>

        <!-- Get each contact record -->
        <xsl:apply-templates select="/crmdata/contacts/contact"/>

      </dn:data>
     </xsl:template>

     <xsl:template match="contact">

       <!-- Write out a data record -->
       <dn:record>
         <dn:value name="Id"><xsl:value-of select="@id"/></dn:value>
         <dn:value name="FirstName"><xsl:value-of select="name/firstname"/></dn:value>
         <dn:value name="LastName"><xsl:value-of select="name/surname"/></dn:value>
         <dn:value name="DateOfBirth"><xsl:value-of select="dob"/></dn:value>
         <dn:value name="Height">
          <xsl:value-of select="properties/property[@name='height']/@value"/>
         </dn:value>
         <dn:value name="Weight">
```

```
          <xsl:value-of select="properties/property[@name='weight']/@value"/>
         </dn:value>
      </dn:record>

    </xsl:template>

  </xsl:stylesheet>
```

5.2.2.1 Configuring the Data Store

The data can be read in to by creating an XML and Stylesheet data store and specifying the location of the XML source file and the XSLT file (stylesheet).



XML and Stylesheet Configuration

File	crmdata.xml
XSLT file	crm.xsl
Record element	contacts/contact

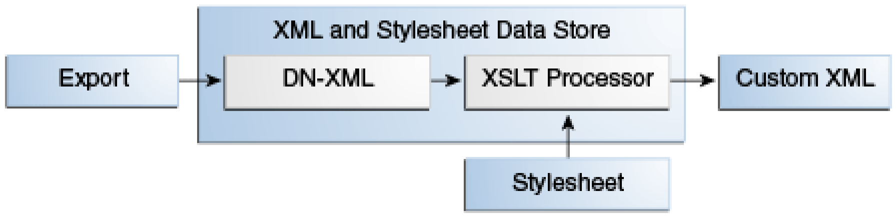
Specify element name path to record. Default is the element below the root.

Ignore invalid values ☐

reads the source XML file in chunks for efficiency breaking up the file on record boundaries. By default uses the element immediately below the root as the record element. If this is not the case in the source XML file then an XPath-style expression to the record element from the root must be specified.

5.2.3 Writing Custom XML Files

XML files in custom formats can be written by using the XML and Stylesheet data store configured to use a custom XSLT to transform from the DN-XML schema to the custom target schema the during data export.



Following is an example target custom XML format that needs to be generated by :

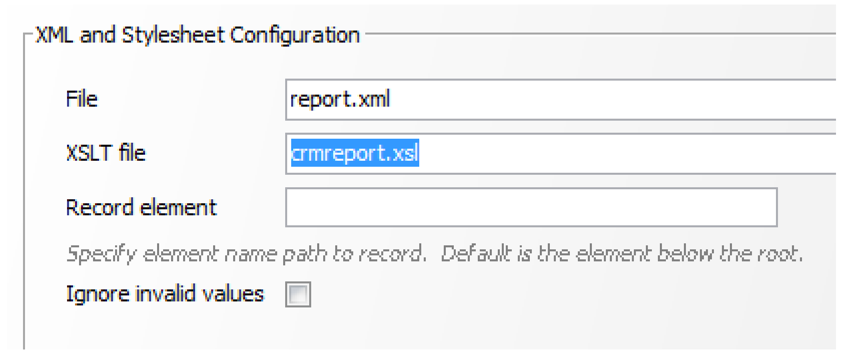
```
<Report>
  <Person Id="1" FullName="Fred Bloggs"/>
  <Person Id="2" FullName="Jane Smith"/>
</Report>
```

The following XML stylesheet demonstrates one way in which the DN-XML format can be transformed into the target custom XML format:

```
<xsl:stylesheet version="1.0"
  xmlns:dn="http://www.datanomic.com/2008/dnx"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/02/xpath-functions">

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <Report>
      <xsl:apply-templates select="/dn:data/dn:record"/>
    </Report>
  </xsl:template>

  <xsl:template match="dn:record">
    <Person>
      <xsl:attribute name="Id">
        <xsl:value-of select="dn:value[@name = 'Id']"/>
      </xsl:attribute>
      <xsl:attribute name="FullName">
        <xsl:value-of select="dn:value[@name = 'FirstName']"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="dn:value[@name = 'LastName']"/>
      </xsl:attribute>
    </Person>
  </xsl:template>

</xsl:stylesheet>
```

5.2.3.1 Configuring the Data Store

The data can be written by by creating an XML and Stylesheet data store and specifying the destination for the custom XML file and XSLT (stylesheet) file.



XML and Stylesheet Configuration

File	report.xml
XSLT file	crmreport.xsl
Record element	

Specify element name path to record. Default is the element below the root.

Ignore invalid values ☐

6

Using the EDQ Configuration API

EDQ provides a set of REST-based interfaces that enable you to perform various configuration tasks programmatically, using any preferred programming language.
In this chapter, the EDQ service is assumed to be installed at:

```
http://edqserver:8001/edq
```

This chapter provides a detailed description of these interfaces and the operations that can be performed using these interfaces. It includes the following topics:

- REST Interface for Projects
- REST Interface for Data Stores
- REST Interface for Snapshots
- REST Interface for Processes
- REST Interfaces for Jobs
- REST Interface for Reference Data
- REST Interface for Web Services
- Example: Profiling from an External Application

6.1 REST Interface for Projects

The REST interface for working with EDQ projects is

```
http://edqserver:8001/edq/config/projects
```

This interface allows you to perform the following tasks:

- Retrieving a List of EDQ Projects
- Creating a Project
- Deleting a Project

6.1.1 Retrieving a List of EDQ Projects

To get a list of all projects that are available with the current EDQ installation, you need to simply run an HTTP GET operation on the REST interface for EDQ projects, as shown in the following code:

```
GET http://edqserver:8001/edq/config/projects
```

When this code runs successfully, a list of projects is generated in JSON format:

```
[
   {
      "id":10,
      "name":"My Project"
   },
   {
        "id":12,
        "name":"Scratch Project"
    }
]
```

6.1.2 Creating a Project

To create a new project, you need to create a JSON object that describes the project to be created and then send it in the request body of the REST call using an HTTP POST.

For example:

```
POST http://edqserver:8001/edq/config/projects

{ "name" : "Profile Customer Names" ,
  "description" : "Profile my customers" }
```

This code returns a response of type "OK" with a response body similar to the following:

```
{"id":14,"name":"Profile Customer Names"}
```

However, if an error occurs while creating a project, then a response of type "500 Internal Server Error" is generated, along with an error message similar to the following:

```
"Profile Customer Names" already exists (Code: 205,130)
```

6.1.3 Deleting a Project

To delete a project, you need to call HTTP DELETE on the REST interface and specify the project you need to delete, as a query parameter.

There are two ways to specify a project for deletion:

- By ID using `pid=<NN>`
- By name using `pname=<Name>`

To delete a project by ID, do the following:

```
DELETE http://edqserver:8001/edq/config/projects?pid=14
```

To delete a project by name, do the following:

```
DELETE http://edqserver:8001/edq/config/projects?pname=Profile%20Customer%20Names
```

In both cases, the result is a string similar to the following:

```
Project Profile Customer Names deleted
```

If an invalid project is specified then a response of type 406 'Not Acceptable' is returned with an appropriate string message, for example:

```
Bad project ID "14" (Code: 205,454)
```

or

```
No project named "Profile Customer Names" (Code: 205,453)
```

6.2 REST Interface for Data Stores

You can query and manipulate data stores in EDQ, using the following interface:

```
http://edqserver:8001/edq/config/datasources
```

This interface allows you to perform the following tasks:

- Retrieving a List of Data Stores
- Creating a Data Store
- Deleting a data store

6.2.1 Retrieving a List of Data Stores

To get a list of data stores, call the interface with a valid project name.

For example:

```
GET http://edqserver:8001/edq/config/datasources?pid=14
```

If successful, an OK response is returned along with a list of data stores in the response body.

For example:

```
[
   {
      "client":false,
      "id":36,
      "name":"Individuals",
      "properties":[
         {
            "name":"quote",
            "value":"\""
         },
         {
            "name":"encoding",
            "value":"ISO-8859-1"
         },
         {
            "name":"file",
            "value":"Customer/customerindividuals.csv"
         },
         {
            "name":"cols",
            "value":""
         },
         {
            "name":"project",
            "value":"59"
         },
         {
            "name":"hdr",
            "value":"1"
         },
         {
            "name":"usepr",
            "value":"0"
         },
         {
            "name":"skip",
            "value":""
         },
         {
            "name":"sep",
            "value":","
         }
      ],
      "species":"servertxt"
   }
]
```

6.2.2 Creating a Data Store

To create a data store you need to create a JSON object that describes the data store and then POST it to the endpoint specifying the project (by name or id) that will own the data store.

For example:

A server-based `.csv` file that has been placed in the landing area.

```
POST http://edqserver:8001/edq/config/datasources?pid=14

{
   "client":false,
   "name":"Individuals",
   "properties":[
      {
         "name":"quote",
         "value":"\""
      },
      {
         "name":"encoding",
         "value":"ISO-8859-1"
      },
      {
         "name":"file",
         "value":"Customer/customerindividuals.csv"
      },
      {
         "name":"hdr",
         "value":"1"
      },
      {
         "name":"usepr",
         "value":"0"
      },
      {
         "name":"sep",
         "value":","
      }
   ],
   "species":"servertxt"
}
```

A server-based Oracle schema:

```
POST http://edqserver:8001/edq/config/datasources?pid=14

{
   "client":false,
   "name":"Staging",
   "properties":[
      {
         "name":"service",
         "value":"sid"
      },
      {
         "name":"sid",
         "value":"orcl"
      },
      {
         "name":"user",
         "value":"staging"
      },
      {
         "name":"port",
         "value":"1521"
      },
      {
         "name":"password",
         "value":"staging"
      },
      {
         "name":"host",
         "value":"localhost"
      }
   ],
   "species":"oracle"
}
```

If successful, an OK response is returned along with the name and ID of the data store, as shown in the following example:

```
{"id":42,"name":"Staging"}
```

The value of the `species` parameter varies depending on the type of data store being used in a project. For example, if you are using the Oracle database, the value of `species` would be "`oracle`". Each species parameter has its own set of properties.

The following table lists the properties for the species "oracle":

Property	Type	Required	Description
host	String	Yes	The machine hosting the database
port	Number	Yes	Port number of the database
sid	String	Yes	Database Identifier
service	Choice of sid or srv	Yes	Whether the database identifier above is a SID or a SERVICE (srv) name
user	String	Yes	User to log in as
password	String	No	Password for the user
schema	String	No	The schema to use (usually left empty)

The following table lists the properties for the species "servertext":

Property	Type	Required	Description
file	String	Yes	The name and location of the file in the landing area
userpr	Boolean	Yes	Use project specific landing area
hdr	Boolean	Yes	Treat the first line as header
sep	String	No	The field delimiter
quote	Choice	No	Quote character

Property	Type	Required	Description
cols	Integer	No	Number of columns to read
encoding	String	Yes	The character encoding of text
skip	Integer	No	The number of the lines to skip at the start

Note:

For Boolean type use 0 for false and 1 for true.

For quote, the value needs to be a double quote like this ""\"", or a single quote like this "'", or an empty value like this "".

For the species, "other", which uses a JDBC connection, the properties are listed in the following table:

Property	Type	Required	Description
driver	String	Yes	The JDBC driver java class
url	String	Yes	The address of the database
user	String	No	The user to log in as
password	String	No	The user's password

6.2.3 Deleting a data store

To delete a data store call HTTP DELETE on the endpoint by specifying either a data store id or a valid project (by name or id) and a data store name, for example:

```
DELETE http://edqserver:8001/edq/config/datasources?id=42
```

or

```
DELETE http://edqserver:8001/edq/config/datasources?pid=14&name=Staging
```

When the deletion is successful, an OK response is returned without any response body.

6.3 REST Interface for Snapshots

The REST interface for snapshots is:

```
http://edqserver:8001/edq/config/snapshots
```

It allows you to perform the following tasks:

- Retrieving a List of Snapshots
- Creating a Snapshot
- Deleting a Snapshot

6.3.1 Retrieving a List of Snapshots

To retrieve a list of snapshots specify a valid project, for example:

```
GET http://edqserver:8001/edq/config/snapshots?pid=14
```

or

```
GET http://edqserver:8001/edq/config/snapshots?pname=Profile%20Customer%20Names
```

If successful, it returns an OK response with a list of snapshots in the response body.

For example:

```
[
   {
      "columns":[
         "TITLE",
         "FULLNAME",
         "GIVENNAMES",
         "FAMILYNAME",
         "NAMETYPE",
         "PRIMARYNAME",
         "ADDRESS1",
         "ADDRESS2",
         "ADDRESS3",
         "ADDRESS4",
         "CITY",
         "STATE",
         "POSTALCODE"
      ],
      "datasource":"Individuals",
      "name":"Individuals",
      "table":"customerindividuals.csv"
   }
]
```

6.3.2 Creating a Snapshot

To create a snapshot, you need to create a JSON object that describes the snapshot and specify the project where it will be created.

For example:

```
POST http://edqserver:8001/edq/config/snapshots?pid=14

{
   "name":"Individuals",
   "description":"Customer data",
   "datasource":"Individuals",
   "table":"customerindividuals.csv",
   "columns":[
      "TITLE",
      "FULLNAME",
      "GIVENNAMES",
      "FAMILYNAME",
      "NAMETYPE",
      "PRIMARYNAME",
      "ADDRESS1",
      "ADDRESS2",
      "ADDRESS3",
      "ADDRESS4",
      "CITY",
      "STATE",
      "POSTALCODE"
   ],
   "sampling":{
      "number":100,
      "offset":0,
      "ordering":"ascending",
      "count":"true"
   }
}
```

If successful, an OK response is returned with the snapshot ID in the response body.

For example:

```
{"id":68,"name":"Individuals"}
```

6.3.3 Deleting a Snapshot

To delete a snapshot, you need to specify either a snapshot ID or a valid project and snapshot name.

For example:

```
DELETE http://edqserver:8001/edq/config/snapshots?id=68
```

or

```
DELETE http://edqserver:8001/edq/config/snapshots?pid=14&name=Individuals
```

or

```
DELETE http://edqserver:8001/edq/config/snapshots?
pname=Profile%20Customer%20Names&name=Individuals
```

When the specified snapshot is deleted successfully, it returns an OK response with a string message in the response body, which is similar to the following:

```
Snapshot Individuals deleted
```

6.4 REST Interface for Processes

The interface for EDQ processes is:

```
http://edqserver:8001/edq/config/processes
```

Using this interface, you can perform the following tasks:

- Retrieving a List of Processes
- Deleting a Process

The following sub-level interface allows you to create a simple profiling process:

```
http://edqserver:8001/edq/config/processes/simpleprocess
```

See Creating a Simple Process section for details.

6.4.1 Retrieving a List of Processes

To get a list of processes in a project, you need to call HTTP GET on the processes interface with a valid project name.

Example:

```
GET http://edqserver:8001/edq/config/processes?pid=14
```

or

```
GET http://edqserver:8001/edq/config/processes?pname=Profile%20Customer%20Names
```

An OK response is returned along with a list of processes.

Example:

```
[{"name":"Profile Names","id":31}]
```

If the request is not successful, an error response would either be a 404 'Not Found' or 500 'Internal Server Error' along with a string in the response body describing the error.

6.4.2 Deleting a Process

To delete a process, specify either the process ID, or the project ID or name, and the process name. For example:

```
DELETE http://edqserver:8001/edq/config/processes?id=31
```

or

```
DELETE http://edqserver:8001/edq/config/processes?pid=14&name=Profile%20Names
```

or

```
DELETE http://edqserver:8001/edq/config/processes?
pname=Profile%20Customer%20Names&name=Profile%20Names
```

When the deletion is successful, an OK response is returned with a string message and response body, as shown in the following example:

```
Process Profile Names deleted
```

If the deletion is not successful, then either of the following errors along with a response string are returned:

- 404 "Not Found"
- 500 "Internal Server Error"

6.4.3 Creating a Simple Process

The interface currently only supports creation of simple profiling processes. To create a simple process, you need to create a JSON object that describes the process you want to create and specify the project where it should be created. The HTTP POST operation is used to post this information to the interface.

For example:

```
POST http://edqserver:8001/edq/config/processes/simpleprocess?pid=14
{
   "name":"Profile Names",
   "description":"Profile Individuals Names",
   "reader":{
      "name":"Read from Individuals",
      "stageddata":"Individuals"
   },
   "processors":[
      {
         "name":"Do Quickstats",
         "type":"dn:quickstatsprofiler",
         "columnlist":[
            "GivenNames",
            "FamilyName"
         ]
      },
      {
         "name":"Do Frequency Profiling",
         "type":"dn:attributefrequencycountsprofiler"
      }
   ]
}
```

When the simple process is created successfully, an OK response is returned along with the name and ID of the process in the response body.

For example:

```
{"id":33,"name":"Profile Names"}
```

An error response would be generated in the following cases:

- 404 'Not Found' if the project does not exist
- 400 'Bad Request' if the JSON object is malformed
- 500 'Internal Server Error' if a server error occurs during creation, along with a string in the response body describing the error.

The full list of supported processors is mentioned in the following table:

Processor	Type
Quick Stats Profiler	dn:quickstatsprofiler
Data Types Profiler	dn:datatypesprofiler
Max/Min Profiler	dn:maxandminprofiler
Length Profiler	dn:lengthprofiler
Record Completeness Profiler	dn:recordcompletenessprofiler
Character Profiler	dn:characterprofiler
Frequency Profiler	dn:attributefrequencycountsprofiler
Patterns Profiler	dn:attributepatternsprofiler

6.5 REST Interfaces for Jobs

The `/config/jobs` interface performs the following tasks on EDQ jobs:

- Retrieving a List of Jobs

- Deleting a Job
- Creating a Simple Job

The URL for this interface is similar to the following:

```
http://edqserver:8001/edq/config/jobs
```

There is another REST interface, `/jobs` to perform the following tasks:

- Running a Job
- Cancelling a Running Job
- Getting the Status of a Job
- Getting the Details of All Running Jobs

The URL for this interface is similar to the following:

```
http://edqserver:8001/edq/jobs
```

6.5.1 Retrieving a List of Jobs

You can get a list of jobs for a project using HTTP GET. You must specify at least one project (there could be more than one project) as the query parameter.

The project can be specified by using the `pid` or `pname` in the query parameter, as shown in the following example:

```
GET http://edqserver:8001/edq/config/jobs?pid=14
```

or

```
GET http://edqserver:8001/edq/config/jobs?pname=Profile%20Customer%20Names
```

The response to this operation would list all the jobs for the specific project or projects, as shown in the following example:

```
[
   {
      "id":99,
      "name":"Profile Names Job"
   },
   {
      "id":98,
      "name":"Profile Individuals Job"
   }
]
```

6.5.2 Deleting a Job

To delete a job you need to either specify a valid job ID or a valid project (using one of the query parameters) and a valid job name.

To delete a job by job ID:

```
DELETE http://edqserver:8001/edq/config/jobs?id=99
```

To delete a job by job name:

```
DELETE http://edqserver:8001/edq/config/jobs?pid=14&name=Profile%20Names%20Job
```

or

```
DELETE http://edqserver:8001/edq/config/jobs?
pname=Profile%20Customer%20Names&name=Profile%20Names%20Job
```

When the job is deleted successfully, a message appears to confirm the deletion:

```
Job Profile Names Job deleted
```

6.5.3 Creating a Simple Job

A simple job has a single phase and contains a single process. To create a simple job, you need to specify a valid project that owns the job (using one of the query parameters). Also, you need to create a JSON object describing the job to create.

For example:

```
POST http://edqserver:8001/edq/config/jobs/simplejob?pid=14
{
"name" : "Profile Names Job" ,
"process" : "Profile Names" ,
"description" : "Profile Customer Names"
"resultsdrilldown" : "none"
}
```

The attribute `resultsdrilldown` can have any of the following values: `none`, `sample`, `limited`, `all`. However, the `sample` and `limited` values have the same implication.

6.5.4 Running a Job

With the `/jobs/run` interface you can run a named job using HTTP POST. The required parameters are the project name or project ID and the job name. Optionally, you can specify run label and overrides.

The following example shows the URL and associated payload used with running a job named "`Real-time Start All`" for the project "`Profile Customer Name`":

```
POST http://edqserver:8001/edq/jobs/run

{
   "project":"Profile Customer Name",
   "job":"Real-time Start All",
   "runlabel": "uk",
   "runprofile": "profiling",
   "overrides":[
      {
         "name":"a",
         "value":"b"
      },
      {
         "name":"c",
         "value":"d"
      }
   ]
}
```

The JSON response to this request would be similar to the following:

```
{
"executionID": 2,
"runeverywhere": false
}
```

In this example, the job "`Real-time Start All`" returns the "`runeverywhere`" value as `false`, which implies that this job can run only in one place. In such cases, an `executionID` is returned for the job. This `executionID` can be used to cancel a job and query a job's status.

However, if the value of "`runeverywhere`" were true, then only the "`jobtype`" would be returned in the JSON response. For "`runeverywhere`" jobs, cancel and query calls are not supported.

6.5.5 Cancelling a Running Job

To cancel a running job, the HTTP POST operation is used. The interface URL is similar to the following:

```
POST http://edqserver:8001/edq/jobs/cancel
```

You only need the `executionID` of the job to cancel it. The following example illustrates cancelling a job with the `executionID 12`.

```
{
"executionID": 12,
"type" : "immediate"
}
```

The following options are available with the "`type`" parameter:

- `immediate`: This option cancels the job as early as possible.
- `keepresults`: This option cancels the job but retains the results that have been generated so far.
- `shutdown`: This options is used to cancel or shutdown a job that runs a web service.

No response is returned when a job is cancelled successfully.

6.5.6 Getting the Status of a Job

To get the status of an individual job, the `executionID` is passed in the URL. The URL looks similar to the following:

```
GET http://edqserver:8001/edq/jobs/status?xid=executionID
```

For example, if the execution ID of the job "`Real-time Start All`" is 14, the URL would be:

```
GET http://edqserver:8001/edq/jobs/status?xid=14
```

The JSON response would be as follows:

```
{
   "executionid": 14,
   "project": "Profile Customer Name",
   "job": "Real-time START ALL",
   "server": "edqserver",
   "starttime": "2016-04-20T08:44:48+01:00",
   "endtime": "2016-04-20T08:45:22+01:00",
   "complete": true,
   "status": "finished"}
```

In this example, the "`Real-time Start All`" job triggers other jobs. Once all the jobs in the project are triggered, the status of the `executionID 14` shows `finished`. However, the jobs that are triggered by the "`Real-time Start All`" job, may still show the status as `running`.

6.5.7 Getting the Details of All Running Jobs

The status of all running jobs in a project can be retrieved using the `/jobs/running` interface, which would be represented by a URL similar to the following:

```
GET http://edqserver:8001/edq/jobs/running
```

The following example shows the output in JSON format:

```
{
      "executionid": 4,
      "project": "Profile Customer Name",
      "job": "Real-time Individual Clean",
      "server": "edqserver",
      "starttime": "2016-04-19T10:05:30.74+01:00",
      "endtime": "2016-04-19T16:05:41+01:00",
      "complete": false,
      "status": "running"
    }
```

Optionally, you can provide other query parameters such as project name, job name, and run label. For jobs without a run label, omit the run label parameter and set it to empty filter jobs with no run label. The URL in this case would be similar to the following:

```
/jobs/running?[project=project[&job=job][&runlabel=]
```

6.6 REST Interface for Reference Data

Reference data can exist within a project or outside of all projects at system level. To refer to reference data at system level, specify the project ID as 0 or `pid=0`.

The interface for reference data is:

```
http://edqserver:8001/edq/config/referencedata
```

The interface for reference data content is:

```
http://edqserver:8001/edq/config/referencedata/contents
```

You can use this interface to perform the following tasks:

- Retrieving a List of Reference Data
- Retrieving Contents of Reference Data
- Creating Reference Data
- Deleting Reference Data

6.6.1 Retrieving a List of Reference Data

To get a list of all reference data defined at system level specify the project using either of the following parameters:

By `pid = 0`:

```
GET http://edqserver:8001/edq/config/referencedata?pid=0
```

By `pname`:

```
GET http://edqserver:8001/edq/config/referencedata?
pname=Profile%20Customer%20Names
```

A list of JSON objects, which represent reference data, are returned. The output looks similar to the following:

```
[
   {
      "activerows":2,
      "category":"charactertokeymap",
      "columns":[
         {
            "key":true,
            "name":"Name",
            "type":"STRING",
            "unique":true
         },
         {
            "name":"Value",
            "type":"STRING",
            "value":true
         }
      ],
      "id":39,
      "name":"Tokens",
      "totalrows":2
   }
]
```

6.6.2 Retrieving Contents of Reference Data

To list the contents of the reference data, the reference data contents interface is used. You must specify a valid project or use `pid=0` for system level.

For example:

```
GET http://edqserver:8001/edq/config/referencedata/contents?pid=0&id=40
```

or

```
GET http://edqserver:8001/edq/config/referencedata/contents?
pid=0&name=ShortNameMap
```

This returns information about the reference data rows, as shown in the following code snippet:

```
{
   "activerows":4,
   "columns":[
      {
         "key":true,
         "name":"ShortName",
         "type":"STRING",
         "unique":true,
         "value":true
      },
      {
         "key":true,
         "name":"LongName",
         "type":"STRING",
         "value":true
      }
   ],
"description":"Map short names to long names",
   "id":43,
   "name":"ShortNameMap",
   "rows":[
      {
         "data":[
            "Jeff",
            "Jeffrey"
         ]
      },
      {
         "data":[
            "Jon",
            "Jonathan"
         ]
      }
   ],
   "totalrows":4
}
```

6.6.3 Creating Reference Data

To create reference data you need to create a JSON object describing the reference data, which you will post to the interface specifying either `pid=0` for system level, or a valid project name.

For example:

```
POST
http://edqserver:8001/edq/config/referencedata?pname=Profile%20Customer%20Names

{
"name" : "ShortNameMap",
"description" : "Map short names to long names",
"columns":
        [
            {
                "key": true,
                "name": "ShortName",
                "type": "STRING",
                "unique": true,
                "value": true
            },
            {
                "key": true,
                "name": "LongName",
                "type": "STRING",
                "value": true
            }
    ],
"rows":
    [
        {
            "data":
            [
                "Jeff",
                "Jeffrey"
            ]
        },
        {
            "data":
            [
                "Jon",
                "Jonathan"
            ]
        }
]
}
```

On successful creation, a response similar to the following is returned:

```
{"id":40,"name":"ShortNameMap"}
```

6.6.4 Deleting Reference Data

To delete reference data, you need to call HTTP DELETE on the reference data interface, with either a valid reference data ID or a valid project (including `pid=0` for system level) and a valid reference data name.

To delete by reference data ID:

```
DELETE http://edqserver:8001/edq/config/referencedata?id=40
```

To delete by reference data name:

```
DELETE http://edqserver:8001/edq/config/referencedata?pid=0&name=ShortNameMap
```

After a successful deletion, the response returns a string message, such as the following:

```
Reference data ShortNameMap deleted
```

6.7 REST Interface for Web Services

The interface for web services is:

```
http://edqserver:8001/edq/config/webservices
```

It allows you to perform the following tasks when you call the respective get, post, or delete operations:

- Retrieving a List of Web Services
- Creating or Updating a Web Service
- Deleting a Web Service

6.7.1 Retrieving a List of Web Services

You can get a list of web services defined for a valid project by using the HTTP GET operation on the web services interface. To get a list of web services, specify a valid project using either the `pid` or `pname` parameter.

For example:

```
GET http://edqserver:8001/edq/config/webservices?pid=14&pid=20
```

A successful call returns a list of web services, and their input and output interfaces, in the response body:

```
[
    {
        "id":1,
            "inputs":{
               "attributes":[
                  {
                     "name":"Name",
                     "type":"STRING"
                  }
               ],
               "multirecord":false
            },
            "name":"Long Names",
            "outputs":{
               "attributes":[
                  {
                     "name":"LongName",
                     "type":"STRING"
                  }
               ],
               "multirecord":false
            }
         }
      ]
```

6.7.2 Creating or Updating a Web Service

You can create and update web services by creating an appropriate JSON object, which you then POST to the web services interface.

To create a web service you need to specify a valid project (by name or id), as shown in the following example:

```
POST http://edqserver:8001/edq/config/webservices?pid=14

{
   "name":"Name Gender",
   "inputs":{
      "attributes":[
         {
            "name":"Name",
            "type":"STRING"
         }
      ],
      "multirecord":false
   },
   "outputs":{
      "attributes":[
         {
            "name":"Gender",
            "type":"STRING"
         }
      ],
      "multirecord":false
   }
}
```

If successful, the name and ID of the web service is returned in the response body.

Example:

```
{"id":4,"name":"Name Gender"}
```

To update a web service, you need a JSON object that is identical in structure, but with an additional ID attribute to identify the existing web service. For an update you do not specify a project.

Example:

```
POST http://edqserver:8001/edq/config/webservices

{
   "id":4,
   "name":"Name Gender",
   "inputs":{
      "attributes":[
         {
            "name":"First Name",
            "type":"STRING"
         },
         {
            "name":"Last Name",
            "type":"STRING"
         }
      ],
      "multirecord":false
   },
   "outputs":{
      "attributes":[
         {
            "name":"Gender",
            "type":"STRING"
         }
      ],
      "multirecord":false
   }
}
```

If successful, the name and ID of the web service is returned in the response body, as shown in the following example:

```
{"id":4,"name":"Name Gender"}
```

6.7.3 Deleting a Web Service

To delete a web service call HTTP DELETE on the web service interface. Specify either the web service ID or a valid project (by name or ID) and the web service name.

For example:

```
DELETE http://edqserver:8001/edq/config/webservices?id=5
```

or

```
DELETE http://edqserver:8001/edq/config/webservices?pid=14&name=Name%20Gender
```

If successful, an OK response is returned but without a response body.

6.8 Example: Profiling from an External Application

Consider a scenario where an external application needs to profile data in a table in an Oracle database, using EDQ. In such a case, you can programmatically profile this table using the

REST-based APIs. For this example, a CUSTOMERS table in a CustomerDB database will be used.

To generate and run the profiling job on the CUSTOMERS table, the following tasks are performed:

1. Create a project by using the following URL:

   ```
   POST http://edqserver:8001/edq/config/projects
   ```

   The project name (`pname`) is "Profile Customer". The JSON code is:

   ```
   {
   "name":"Profile Customer",
   "description": "Profiling customers in the CUSTOMERS table"
   }
   ```

2. Create a data store using an Oracle database, CustomerDB, by using the following URL:

   ```
   POST http://edqserver:8001/edq/config/datasources?pid=4
   ```

   An example JSON code to create the data store is:

   ```
   {
      "client":false,
      "name":"CustomersDB",
      "properties":[
         {
            "name":"service",
            "value":"sid"
         },
         {
            "name":"sid",
            "value":"orcl"
         },
         {
            "name":"user",
            "value":"CRM"
         },
         {
            "name":"port",
            "value":"1521"
         },
         {
            "name":"password",
            "value":"welcome123"
         },
         {
            "name":"host",
            "value":"localhost"
         }
      ],
      "species":"oracle"
   }
   ```

   Note:

   To determine the `pid` or the project ID for the project "Profile Customer", use the HTTP GET operation with the URL:

   ```
   GET http://edqserver:8001/edq/config/projects
   ```

3. Create a snapshot by using the following URL:

   ```
   POST http://edqserver:8001/edq/snapshots?pid=4
   ```

   The JSON code for creating the snapshot is:

   ```
   {
      "name":"CustomersDB.Customers",
      "description":"Customer details",
      "datasource":"CustomersDB",
      "table":"Customers",
      "columns":[
         "ID",
         "FULLNAME",
         "GIVENNAME",
         "FAMILYNAME",
         "Street",
         "City",
         "State",
         "PostalCode",
         "State",
         "Phone",
         "Cell",
         "Work",
         "eMail",
         "DoB",
         "Gender",
         "Active",
         "CreditLimit",
         "StartDate",
         "EndDate"
      ],
      "sampling":{
         "number":100,
         "offset":0,
         "ordering":"ascending",
         "count":"true"
      }
   }
   ```

   The result for this is displayed as:

   ```
   {
      "id": 85,
      "name": "CustomersDB.Customers"
   }
   ```

   The snapshot with the name "CustomersDB.Customers" is created.

4. Create a simple process by using the following URL:

   ```
   POST http://edqserver:8001/edq/config/processes/simpleprocess?pid=4
   ```

   For this example, a simple process is created with a `Quickstats Profiler` and a `Frequency Profiler`, both profiling only the Name fields. This can be done using the following example JSON.

   ```
   {
      "name":"Profile Names",
      "description":"Profile Customer Names",
      "reader":{
         "name":"Read from Customers",
         "stageddata":"Connection to Customers"
      },
   ```

```
   "processors":[
      {
         "name":"Do Quickstats",
         "type":"dn:quickstatsprofiler",
         "columnlist":[
            "GIVENNAME",
            "FAMILYNAME"
         ]
      },
      {
         "name":"Do Frequency Profiling",
         "type":"dn:attributefrequencycountsprofiler"
      }
   ]
}
```

The response to this request is:

```
{
   "id": 267,
   "name": "Profile Names"
}
```

5. Create a simple job by using the following URL:

   `POST http://edqserver:8001/edq/jobs/simplejob?pid=4`

```
{
   "name":"Profile Customer Job",
   "process":"Profile Names",
   "description":"Profiling Customer Names",
   "resultsdrilldown":"none"
}
```

   The response to the request is:

```
{
  "id": 211,
  "name": "Profile Customer Job"
}
```

6. Run the job using the following URL:

   `POST http://edqserver:8001/edq/jobs/run`

   The JSON code for running the job "Profile Customer Job", which in turn would run the profiling process, is:

```
{
"project":"Profile Customer",
"job":"Profile Customer Job"
}
```

   The response is:

```
{
   "executionID": 20,
   "runeverywhere": false
}
```

Once this job is running, you can check the status of this execution of the job using the following URL:

`GET http://edqserver:8001/edq/jobs/status?xid=20`

Running this URL displays the status of the job, as shown in the following code:

```
{
   "complete": true,
   "endtime": "2016-04-29T14:05:41+01:00",
   "executionid": 1,
   "job": "Profile Customer Job",
   "project": "Profile Customer",
   "server": "edq_server1",
   "starttime": "2016-04-29T14:05:38+01:00",
   "status": "finished"
}
```

If required, you can cancel the job using the following URL:

```
POST http://edqserver:8001/edq/jobs/cancel
```

The JSON code to cancel a job is:

```
{
"executionID": 12345,
"type" : "immediate"
}
```

This would cancel the job instantly, without saving the results. For other options that can be used with "type", see Cancelling a Running Job.

To log in to EDQ Director to view the results of a profiling job that was executed successfully, use the following URL:

```
http://edqserver:8001/edq/blueprints/director/jnlp?
projectid=1&processid=1&processornum=2
```

The `projectid` and `processid` are the same that are generated using the corresponding REST API calls and the `processornum` value is set to 2, which is the first processor after the reader.

This URL opens the Director UI with the focus on the first profiling processor in the job so that its results can be viewed immediately.

An external application may include an option to remove generated jobs, which would execute calls to the relevant deletion calls. The simplest version of this is to delete the whole project. For details on deleting a project, see Deleting a Project.

7 Using the EDQ Case Management API

EDQ provides a set of REST-based interfaces to allow case retrieval and update, filter execution, and load testing.

To open the Case Management API, select the Case Management API Specification from the Web Services dropdown menu in the Launchpad; the service loads automatically once selected. In this chapter, the EDQ service is assumed to be installed at:

```
http://edqserver:8001/edq
```

This chapter provides a detailed description of these interfaces and the operations that can be performed using these interfaces. It includes the following topics:

- Retrieving Details of a Case or Alert
- Executing a Saved Filter or Report
- Updating Case or Alert Details
- Adding Comments to One or More Cases or Alerts
- Deleting One or More Cases or Alerts
- Bulk Deleting Cases or Alerts using a Filter
- Executing Multiple Filters for Load Testing
- Bulk Updating Cases or Alerts using a Filter
- Running a Report
- Exporting Cases and Alerts
- Canceling a Bulk Operation
- Filter Definition in JSON Format
- Checking the Execution Status of a Bulk Operation
- Using the Case Management Administration REST APIs

7.1 Retrieving Details of a Case or Alert

To get details for a single case or alert, call the interface with an external or internal ID.

```
GET http://edqserver:8001/edq/cm/getcase/{id}[?sourcedata=true]
```

The following table lists the supported input properties for this call:

Property	Type	Required	Description
Id	String	Yes	The case identifier. Specify either an external ID such as S2-1241, or a numeric internal ID.

Property	Type	Required	Description
sourcedata	Boolean	No	Set to true to include source and relationships data (alerts only). **Note:** This property is available in EDQ 12.2.1.4.3 and later releases.

The following table lists the output properties displayed for each case:

Property	Type	Required	Description
Id	String	Yes	The external case ID. You must specify either the external case ID or the internal case ID.
internalid	Integer	Yes	The internal case ID. You must specify either the external case ID or the internal case ID.
source	String	Yes	The case source name.
sourceid	String	Yes	The case source identifier.
type	String	Yes	The type of the case or alert.
parentid	Integer	No	For alerts, the internal ID of the containing case.
key	String	Yes	The case key.
description	String	No	The description of the case.
createdby	String	Yes	The name of the user who created the case.
createdbyid	Integer	Yes	The internal ID of the user who created the case.
createdbydispl ay	String	Yes	The display name of user who created the case.
createdwhen	String	Yes	The creation time stamp.
modifiedby	String	Yes	The name of user who made the most recent modification.
modifiedbyid	Integer	Yes	The internal ID of user who made most recent modification.
modifiedbydisp lay	String	Yes	The display name of user who made most recent modification.
modifiedwhen	String	Yes	The time stamp of the most recent modification.
assigneduser	String	No	The name of the currently assigned user. This value is not present if the case is unassigned.
assigneduserid	String	No	The internal ID of the currently assigned user. This value is not present if the case is unassigned.
assigneduserdi splay	String	No	The display name of the currently assigned user. This value is not present if the case is unassigned.

Property	Type	Required	Description
assignedby	String	No	The name of the user who made the most recent assignment.
assignedbyid	Integer	No	The internal ID of the user who made the most recent assignment.
assignedbydisplay	String	No	The display name of the user who made the most recent assignment.
assignedwhen	String	No	The time stamp of the most recent assignment.
priority	Integer	Yes	The case priority. minimum: 0 maximum: 3
state	String	Yes	The current state.
statechangedby	String	No	The name of the user who made the most recent state change.
statechangedbyid	Integer	No	The internal ID of the user who made the most recent state change.
statechangedbydisplay	String	No	The display name of the user who made the most recent state change.
statechangedwhen	String	No	The time stamp of the most recent state change.
derivedstate	String	Yes	The derived state
reviewflag	Boolean	Yes	The new review flag.
reviewflagupdatedby	String	No	The name of the user who made the most recent review flag change.
reviewflagupdatedbyid	String	No	The internal ID of the user who made the most recent review flag change.
reviewflagupdatedbydisplay	String	No	The display name of the user who made the most recent review flag change.
reviewflagupdatedwhen	String	No	The time stamp of the most recent review flag change.
permission	String	No	The case permission. The value is the internal key defined in Case Management Administration.
extendedattribute*n*	String	No	The extended attribute
extendedattribute*n*modifiedby	String	No	The name of the user who made most recent modification to extended attribute *n*.
extendedattribute*n*modifiedbyid	Integer	No	The internal ID of the user who made most recent modification to extended attribute *n*.
extendedattribute*n*modifiedbydisplay	String	No	The display name of the user who made most recent modification to extended attribute *n*.
extendedattribute*n*modifiedwhen	String	No	The time stamp of the most recent modification to extended attribute *n*.
extendedattribute*n*label	String	No	The label of the extended attribute *n*.

Unset values are omitted. The extended attributes, which are included in the JSON object, are derived from the definitions in the system `flags.xml` file in the `oedq_local_home/casemanagement` directory..

The timestamps are returned in the ISO format, which is *YYYY-MM-DDTHH:MM:SS.mmmZ*

If the `sourcedata` flag is set to **true**, and the result is for an alert, the output properties also include the `sourcedata` and `relationships` attributes.

The following table lists the output properties that are displayed for the `sourcedata` attribute:

Property	Description
relationships	Relationships data
sourcedata	Source data

The following table lists the output properties that are displayed for the `relationship` attribute:

Property	Description
inputname	The internal name of the input stream.
recordid	The internal ID of the source record.
relatedinputname	The internal name of the related input stream.
relatedrecordid	The internal ID of the related source record.
rulename	The internal ID of the related source record.
priorityscore	The name of the rule that generated the relationship.

The **sourcedata** object is an array of input objects, each containing the input stream name and label, and the associated attribute labels, along with the source records.

The following is an example of the relationship and source data from a simple alert:

```
"relationships": [
  {
    "inputname": "workdata 1",
    "recordid": 77,
    "relatedinputname": "refdata 1",
    "relatedrecordid": 199342,
    "rulename": "MatchRule1",
    "priorityscore": 0
  }
],
"sourcedata": [
  {
    "inputname": "refdata 1",
    "label": "refdata",
    "labels": {
      "ID": "ID",
      "TITLE": "TITLE",
      "FIRSTNAME": "FIRSTNAME",
      "LASTNAME": "LASTNAME",
      "TELEPHONE": "TELEPHONE",
      "NewAttribute": "AGE",
      "NewAttribute1": "type"
    },
    "records": [
      {
        "recordid": 199342,
        "data": {
          "ID": 48,
          "TITLE": "Doctor",
          "FIRSTNAME": "Riaz",
          "LASTNAME": "Audoire",
          "TELEPHONE": "05305 937614",
          "NewAttribute": 67,
          "NewAttribute1": "I"
        }
      }
    ]
  },
  {
    "inputname": "workdata 1",
    "label": "workdata",
    "labels": {
      "ID": "ID",
      "TITLE": "TITLE",
      "FIRSTNAME": "FIRSTNAME",
      "LASTNAME": "LASTNAME",
      "TELEPHONE": "TELEPHONE",
      "NewAttribute": "AGE",
      "NewAttribute1": "type"
    },
    "records": [
      {
        "recordid": 77,
        "data": {
          "ID": 48,
          "TITLE": "Doctor",
          "FIRSTNAME": "Riaz",
          "LASTNAME": "Audoire",
          "TELEPHONE": "05305 937614",
          "NewAttribute": 3,
          "NewAttribute1": "I"
        }
      }
    ]
  }
]
```

7.2 Executing a Saved Filter or Report

Use `runfilter` to run a saved case management filter or report. The retrieved results are generated and displayed in JSON format. The filter results are limited according to the same rules as in the Case Management application. This call returns a limited number of results, along with indications of the total number of results available. You can specify the limit. The default limit is 100.

To execute a saved case management filter or report, use the following interface:

```
GET http://edqserver:8001/edq/cm/runfilter?filter=NAME[&global=true][&limit=100]
[&sql=false]
```

You can also create a JSON object that describes the filter you want to execute and then send it in the request body of the REST call using an HTTP POST. For example,

```
POST http://edqserver:8001/edq/cm/runfilter
{
  "filter": "string",
  "global": true,
  "limit": 50,
  "sourcedata": true
}
```

The following table lists the supported input properties for `runfilter`:

Property	Description
filter	The name of the filter or inline filter definition. See Filter Definition in JSON Format.
global	If set to true, specifies the global filter.
orderby	An array of strings that defines the order by columns for the query. Use the same column names as in the filter definition. Add an `asc` or `desc` suffix to control the order direction. For example: `orderby: ["source", "description desc", "extendedattribute4 asc"]`
limit	The result limit in the range 1-1000. Minimum: 1 Maximum: 1000 The default value is 100.
sql	Set to false to force non-SQL execution if available. This attribute controls the use of SQL for the `GET` call.
sourcedata	Set to true to include source and relationships data (alerts only).

When this code runs successfully, the details of the case are generated and displayed in JSON format:

```
{
  "version": 1,
  "report": false,
  "count": 1000,
  "more": true,
  "cases": [
     array of case/alert details
   ]
}
```

The **count** and **more** attributes reflect the values seen in the UI. For example, if a filter shows *100 items, 196 total on server*, then **count** = 196 and **more** = false. If a filter shows *100 items, > 1,000 on server*, then **count** = 1000 and **more** = true.

For a report execution, the result is a JSON object with the following structure:

```
{
  "version": 1,
  "report": true,
  "xlabels": [array of X axis labels],
  "ylabels": [array of Y axis labels],
  "scores":  [2 dimensional array of row/col values]
}
```

Reporting supports multiple items on each axis so the elements in the label arrays are arrays of strings.

7.3 Updating Case or Alert Details

To update case or alert details you need to create a JSON object that specifies the details that you want to change.

For example,

```
POST http://edqserver:8001/edq/cm/update
```

The payload to the request is a JSON object with the attributes listed in the following table:

Property	Type	Description
failonanyerror	Boolean	If **true**, fail the request if any updates end in error. Invalid values in updates, such as priority out of range, always cause the call to fail. Errors that occur in individual updates include permission problems.
updates	Array	An array of individual case/alert update requests. The array may not contain more than 1000 elements.

The following table lists the attributes that each object in the updates array can contain:

Property	Type	Required	Description
Id	String	Yes	The external case ID. You must specify either the external case ID or the internal case ID.
internalid	Integer	Yes	The internal case ID. You must specify either the external case ID or the internal case ID.
description	String	No	The description of the case.
assigneduser	String	No	The name of the currently assigned user. This value is not present if the case is unassigned.
priority	Integer	Yes	The case priority. minimum: 0 maximum: 3
reviewflag	Boolean	No	The new review flag.
permission	String	No	The case permission. The value is the internal key defined in Case Management Administration.
extendedattribute*n*	String	No	The extended attribute.
statechange	Object	No	The state change definition.

The **statechange** object can contain these objects:

Property	Description
transition	(Required) The transition name.
comment	The comment for the state change.
restrictingpermission	The Restricting Permission to apply to the comment. Specify the *key* for the permission, not the UI name.

For example,

```
{
  "updates": [
    {
      "id": "ECS-234",
      "assigneduser": "john.sheridan",
      "priority": 3,
      "statechange": {
        "transition": "toMatch",
        "comment": "Updated by API call"
      }
    }
  ]
}
```

The result of a successful call (status 200) is an object with a **status** array containing the results of each individual case/alert update. Each status object contains these attributes:

Property	Description
ok	If true, indicates that the update was successful.
reason	If `ok` is set to **false**, the reason code for the failure.
message	Additional message if the update failed.

The following is an example error response:

```
{ "status": [
    { "ok": false,
      "reason": "unknown_user",
      "message": "Unknown user \"tamie.grindy\""
    }
  ]
}
```

7.4 Adding Comments to One or More Cases or Alerts

To add comments to one or more cases or alerts you need to create a JSON object that specifies the case/alert comment definitions.

For example,

`POST /edq/cm/addcomments`

The payload to the request is a JSON object with the attributes listed in the following table:

Property	Type	Description
failonanyerror	Boolean	If **true**, fail the request if any updates end in error. Invalid values in updates, such as unknown permission, always cause the call to fail. Errors that occur in individual updates include missing cases.
comments	Array	An array of individual case/alert comment definitions. The array may not contain more than 1000 elements.

The following table lists the attributes that each object in the **comments** array can contain:

Property	Type	Required	Description
Id	String	No	The external case ID. You must specify either the external case ID or the internal case ID.
internalid	Integer	No	The internal case ID. You must specify either the external case ID or the internal case ID.
comment	String	Yes	The comment text.
restrictingpermission	String	No	The Restricting Permission to apply to the comment. Specify the *key* for the permission, not the UI name.

The result of a successful call (status 200) is an object with a **status** array containing the results of each individual case/alert update.

7.5 Deleting One or More Cases or Alerts

The user making the call must have the case management bulk delete permission. To delete cases or alerts create a JSON object that includes the array of IDs of case/alert that you want to delete and then send it in the request body of the REST call using an HTTP POST. For example,

```
POST http://edqserver:8001/edq/cm/deletecases
{ "deleteemptycases": true,
  "ids": [
     1213423,
     454344,
     "X1-123",
     "X1-456"
  ]
}
```

The payload to the POST is a JSON object with the following attributes:

Property	Type	Description
ids	array	Array IDs of case/alerts to delete. Each element is a numeric internal ID or a string external ID, such as "S2-1231". The array may not contain more than 1,000 elements.
deleteemptycases	Boolean	If **true**, cases for which all contained alerts were deleted are also deleted. The default is **false**.

The result of a successful call is an object with the following attribute:

Property	Type	Description
count	number	Number of cases/alerts deleted by the call.

7.6 Bulk Deleting Cases or Alerts using a Filter

The user making the call must have the case management bulk delete permission. To delete cases or alerts in bulk using a filter, create a JSON object that includes the filter name and then send it in the request body of the REST call using an HTTP POST. Use the following interface:

```
POST http://edqserver:8001/edq/cm/bulkdelete
```

The payload to the POST is a JSON object with the following attributes:

Property	Type	Description
filter	String or JSON Object	The name of the filter or inline filter definition. See Filter Definition in JSON Format.
global	Boolean	Set to **true** to specify a global filter.
deleteemptycases	Boolean	If **true**, cases for which all contained alerts were deleted are also deleted. The default is **false**.

The bulk deletion runs asynchronously. The result to the call is a JSON object containing a key corresponding to the execution.

```
{
  "executionkey":"a279513d1ea44b0198094ac31ae68258"
}
```

You can use the key to poll for execution completion. See Checking the Execution Status of a Bulk Operation.

If you want to cancel the bulk delete operation, use the `bulkoperation` method. See Canceling a Bulk Operation.

7.7 Bulk Updating Cases or Alerts using a Filter

The user making the call must have the case management bulk update permission, as well as any permissions defined with transitions used in state change definitions.

To update cases or alerts in bulk using a filter, create a JSON object that includes the filter name and then send it in the request body of the REST call using an HTTP POST. Use the following interface:

```
POST http://edqserver:8001/edq/cm/bulkupdate
```

The payload to the POST is a JSON object with the following attributes:

Property	Type	Description
filter	String or JSON Object	The name of the filter or inline filter definition. See Filter Definition in JSON Format.
global	Boolean	Set to **true** to specify a global filter. This value is allowed with named filters only.

Property	Type	Description
assignment	JSON object	The group and user assignment definition.
statechanges	JSON object	The map of origin state to transition definition. **statechanges** may be specified if the sources and case types defined in the filter all map to the same workflow.
edits	JSON objects	The attribute edits. Supported keys are description, priority, reviewflag, permission, stateexpiry, and extendedattribute*N*.

The object used with **assignment** has these attributes:

Property	Type	Description
groups	Array of strings	The list of group names.
users	Array of strings	The list of user names. Use **null** to specify "unassigned".
groupbycase	Boolean	Set to **true** to group assignments by case.

The **edits** object can have these attributes:

Property	Type	Description
description	String	The case or alert description. Use **null** to clear the current value.
priority	Integer	The case or alert priority in the range 0-3.
reviewflag	Boolean	The review flag.
permission	String	The case or alert permission.
stateexpiry	Date	The timestamp of the state expiry.
extendedattributeN	String, number or boolean	The extended attribute.

For example

```
{ "filter": {
    "source": "WFP",
    "type": "alert",
    "state": ["one", "two"]
  },

  "statechanges": {
    "one": {
      "transition": "toThree",
      "comment": "one to three"
    }
  },

  "edits": {
    "description": "Alert description",
    "priority": 1,
    "extendedattribute3": "string 3",
    "extendedattribute1": true,
    "permission": "CMPermission1",
    "stateexpiry": "2024-12-10T00:00:00Z"
  },

  "assignment": {
    "groups": ["Administrators"],
    "users": ["user1", null]
  }
}
```

The bulk update operation runs asynchronously. The result to the call is a JSON object containing a key corresponding to the execution.

```
"executionkey":"a279513d1ea44b0198094ac31ae68258"
```

You can use the key to poll for execution completion. See Checking the Execution Status of a Bulk Operation.

The result object returned with the execution status when complete has these attributes:

Property	Type	Description
processed	Integer	The count of cases or alerts processed during the bulk update.
rejected	Integer	The count of cases or alerts that could not be updated due to locking issues.
failedassignments	Integer	The count of cases or alerts that could not be assigned because no user had view permission.
failedtransitions	Integer	The count of cases or alerts that had a defined transition but the transition was rejected.

If you want to cancel the bulk update operation, use the `bulkoperation` method. See Canceling a Bulk Operation.

7.8 Running a Report

To run a a case management report, using a generic filter, use the following interface:

```
http://edqserver:8001/edq/cm/report
```

The payload to the POST is a JSON object with the attributes listed in the following table:

Property	Type	Description
filter	JSON object	The filter definition.
xaxis	Array of strings	The attribute names for X axis.
yaxis	Array of strings	The attribute names for Y axis.
xaggregation	Number or date aggregation	The aggregation definition for X axis.
yaggregation	Number or date aggregation	The aggregation definition for Y axis.

The **xaxis** and **yaxis** attributes specify the attributes used to form the X and Y axis. The attribute names are as used in filter definitions. If an aggregation definition is supplied, the corresponding axis must have a single numeric flag or date attribute. If the aggregation attribute is set to **null**, no aggregation is performed and the axis contains discrete values.

The aggregation definition for a numeric flag has the attributes listed in the following table:

Property	Type	Description	Default value
granularity	Number	The axis granularity.	10
offset	Number	Offset	0
hideempty	Boolean	If **true**, hide empty rows/columns.	**false**

The aggregation definition for a date has these attributes:

Property	Type	Description	Default value
granularity	Number	The axis granularity. Must be YEAR, QUARTER, MONTH, WEEK, DAY, HOUR, MINUTE or SECOND.	MONTH
hideempty	Boolean	If **true**, hide empty rows/columns.	**false**

7.9 Exporting Cases and Alerts

The user must have the Export to Excel case management permission.

To initiate an asynchronous operation to export cases and alerts to a file or URL, use the following interface:

```
POST http://edqserver:8001/edq/cm/export
```

Supported output formats are JSON, JSON lines, XLSX and CSV. The payload to the POST is a JSON object that includes the following attributes:

Property	Type	Description
filter	String or JSON object	The name of the filter or inline filter definition. See Filter Definition in JSON Format.
global	Boolean	Set to **true** for a global named filter. This value is allowed with named filters only.
orderby	Array of strings	The column ordering.
sourcedata	Boolean	Set to **true** to include source and relationships data (alerts only). Supported with JSON and JSON lines output only.
limit	Number	The limit on the number of cases and alerts that will be written to the export file.
destination	JSON object	The output destination definition.
format	String	The output file format. Must be **json**, **jsonlines**, **xlsx** or **csv**.
label	String	An optional label to describe the export. This label is returned in the bulk status reports and is available in the export trigger.

The **destination** object is required and contains these attributes:

Property	Type	Description
location	String	(Required) The file name or URL for the package file. If a non-absolute file name is specified, the location is relative to the "local home" configuration directory.

Property	Type	Description
credentials	String	The name of stored credentials used to authenticate a request to a URL location. This value is ignored if the location is a file. If the URL requires a simple username/password authentication, use "basic" stored credentials.
platformcredentials	Boolean	Set to **true** to use platform credentials for the file upload and download operations. Supported on Oracle Cloud Infrastructure (OCI), Amazon Web Services (AWS) and Google Cloud Platform (GCP).
method	String	The HTTP method used for uploads. This value is ignored if the location is a file. The default method is **PUT**.
multipart	String	The multipart mechanism used for uploading large files. Supported values are **aws**, **s3**, and **oci**. **aws** and **s3** are equivalent.

The result to the call is a JSON object containing a key corresponding to the execution.

```
"executionkey":"a279513d1ea44b0198094ac31ae68258"
```

You can use the key to poll for execution completion. See Checking the Execution Status of a Bulk Operation.

When the export is completed, a trigger is run with the `/casemanagement/export` path. The single argument passed to the trigger is an object with the following attributes:

Property	Type	Description
failed	Boolean	Is **true** if the export completed with an error.
cancelled	Boolean	Is **true** if the export was cancelled.
start	String	The date and time when the export started.
end	String	The date and time when the export completed.
error	String	The error message if the execution completed with an error. This value is present only if **failed** is **true**.
userid	Integer	The ID of user who is initiating the export.
label	String	The label copied from the run export payload.
location	String	The output location. The value is an absolute file name or URL.
format	String	The output file format.

The following is an example trigger that sends the output file in an email attachment:

```
addLibrary("usercache");
addLibrary("mail");
addLibrary("logging");

function getPath() {
  return "/casemanagement/export"
}

function run(path, id, env, status) {
  if (!status.failed && !status.cancelled) {
    logger.log(Level.INFO, "CM: {0}: {1} exported {2} to {3} ({4})",
status.label, usercache.getUser(status.userid).userName, status.count,
status.location, status.format)

    var mh  = Mail.open({enabled : true});
    var msg = mh.newMessage("An export from cm")

    msg.text = "Export complete"
    msg.addTo("someuser@example.com")
    msg.type = "text/plain";
    msg.addAttachment(status.location, "application/jsonl")
    msg.send()
  }
}
```

If you want to cancel the bulk export operation, use the `bulkoperation` method. See Canceling a Bulk Operation.

7.10 Filter Definition in JSON Format

The filter used with the **runfilter**, **bulkdelete**, **bulkupdate** and export APIs can be the name of a global filter or a JSON map defining the search conditions. These conditions can include any of the standard and extended attributes that are supported in the Case Management UI. Note that, currently, searches using case comments and transitions are not supported.

The **runfilter** payload supports an additional **orderby** attribute, which is an an array of strings that define the order by columns for the query.

The available attributes in the filter map are listed in the following table:

Property	Type	Description	Null values allowed	Can be negated
id	String or array of strings	The Case ID.	No	Yes
internalid	Integer or array of integers	The Case internal ID.	No	Yes
key	String	Free-form text search on the case key.	No	No
description	String	Free form text search on the case description.	No	No
source	String or array of strings	The source name.	No	No
sourceid	String or array of strings	The source identifier.	Yes	Yes
type	Case/alert type	Must be "case" or "alert".	No	No
createdbyid	Integer or array of integers	The ID of user who created the case or alert.	No	Yes
createdwhen	Date, array of dates, or date range	The timestamp when the case or alert was created.	Yes	Yes
modifiedbyid	Integer or array of integers	The ID of the user who modified the case or alert.	No	Yes

Property	Type	Description	Null values allowed	Can be negated
modifiedwhen	Date, array of dates, or date range	The timestamp when the case or alert was modified.	Yes	Yes
assigneduserid	Integer or array of integers	The ID of the currently assigned user. Specify -1 if the case is unassigned.	No	Yes
assignedbyid	Integer or array of integers	The ID of the user who made the most recent assignment.	No	Yes
assignedwhen	Date, array of dates, or date range	The timestamp of the most recent assignment.	Yes	Yes
state	String or array of strings	The current state	No	Yes
statechangebyid	Integer or array of integers	The ID of the user who made the most recent state change.	Yes	Yes
statechangedwhen	Date, array of dates, or date range	The timestamp of the most recent state change.	Yes	Yes
stateexpiry	Date, array of dates, or date range	The timestamp of the case or alert state expiry.	Yes	Yes
derivedstate	String or array of strings	The derived state	Yes	Yes
permission	String or array of strings	The case permission. The value is the internal key defined in Case Management Administration.	Yes	Yes
reviewflag	Boolean	The new review flag. Must be **true** or **false**.	No	No
reviewflagupdatedbyid	Integer or array of integers	The ID of the user who made the most recent review flag change.	Yes	Yes
reviewflagupdatedwhen	Date, array of dates, or date range	The timestamp of the most recent review flag change.	Yes	Yes
priority	Integer or array of integers	The case priority. Values must be in the 0-3 range.	Yes	Yes
extendedattributeN	Value, array or range	The extended attribute *n*.	Yes	Yes
extendedattributeNmodifiedbyid	Integer or array of integers	The ID of the user who made most recent modification to extended attribute *n*.	Yes	Yes
extendedattributeNmodifiedwhen	Date, array of dates, or date range	The timestamp of the most recent modification to extended attribute *n*.	Yes	Yes
sourceattributes	Map of source to array of tests	The filter on source and relationship attributes. See Filtering with source attributes.		

For filter attributes that have timestamp values, see Filtering with dates.

For filtering using extended attributes, see Filtering extended attributes.

For an attribute test that has a *Yes* in the **Can be negated?** column, you can invert the test by adding a boolean attribute with the filter name suffixed with "negated". For example the following filter settings select all alerts with no permission set:

```
"type": "alert",
"permission": null,
"permissionnegated": true
```

Filtering with source attributes

The **sourceattributes** filter value is a map of the source name to an array of tests for the attributes and relationships associated with the source. Each element in the test array has the following attributes:

Attribute	Type	Description
input	String	The name of the input stream as seen in the Case Management UI. For relationship tests use **Relationships**.
attribute	String	The name of the attribute in the stream as seen in the Case Management UI.
value	Object	The value to compare against. The value could be **null**, a scalar value, an array, or a number or date range.
negated	Boolean	Set to **true** to negate the test.

Each source name must also be specified in the filter **source** list.

The **input** and **attribute** values are the stream and source attribute labels as seen in the Source Attributes section in the Browser Pane of the Case Management UI. The UI allows you to edit the labels and use the same label more than once for a stream. If an ambiguous label is used in a filter, the API will reject the call.

For example:

```
{ "source": ["Screening", "Test"],
  "sourceattributes": {
    "Screening": [
      { "input": "workdata",
        "attribute": "firstname",
        "value": "j%"
      },
      { "input": "workdata",
        "attribute": "lastname",
        "value": ["kirk", "doohan"]
      }
    ],
    "Test": [
      { "input": "Relationships",
        "attribute": "Rule Name",
        "value": "R0012 or R002%",
        "negated": true
      },
      { "input": "Watchlist",
        "attribute": "dob",
        "value": { "from": "1990-01-01T00:00:00Z", "to":
"2023-11-21T12:34:56Z" }
      }
    ]
  }
}
```

Filtering with dates

Filter attributes that are timestamps can be specified as **null**, a single ISO timestamp string, an array of timestamp strings, or a date range object. Timestamp strings can omit the *.mmm* millisecond portion.

For example, these are all valid settings of the **createdwhen** attribute:

```
"createdwhen": null
"createdwhen": "2023-10-11T12:34:56Z"
"createdwhen": ["2023-10-11T12:34:56Z", "2024-01-08T11:01:12Z"]
"createdwhen": {"from": "2023-01-01T00:00:00Z", "to": "2024-12-31T23:59:59Z"}
```

The date range object can contain the following attributes:

Attribute	Type	Description
from	Date string	The lower bound of the date range. You must specify either the **from** or the **to** value.

Attribute	Type	Description
to	Datestring	The upper bound of date range. You must specify either the **from** or the **to** value.
inclusive	Boolean	If **true**, the range is inclusive of the upper bound.

Filtering extended attributes

The value specified with an **extendedattributeN** filter depends on the type of the associated flag.

The following table lits the flag type and the associated value format:

File Type	Value format
boolean	**null**, **true**, or **false**
string	**null**, string or array of strings

File Type	Value format
number	**null**, number, array or numbers, or number range

A number range is specified a JSON object with these attributes:

Attribute	Type	Description
from	Date string	The lower bound of the number range.
to	Date string	The upper bound of number range.

Attribute	Type	Description
inclusive	Boolean	If **true**, the range is inclusive of the upper bound.

Examples:

```
"extendedattribute1": null,
"extendedattribute2": true,
"extendedattribute3": "string",
"extendedattribute4": ["monday", "tuesday"],
"extendedattribute5": 20,
"extendedattribute6": [2,4,5,8,37.4],
"extendedattribute7": {"from": 10, "to": 55, "inclusive": true]
```

7.11 Canceling a Bulk Operation

You can cancel a bulk operation such as update, delete or export using the following interface:

```
DELETE http://edqserver:8001/edq/cm/bulkoperation/EXECUTIONKEY
```

The payload to the **POST** call is a JSON object containing an `executionkey` attribute that specifies the execution key that was returned by the call that initiated the bulk operation.

The result is a JSON object containing a **cancelled** attribute, which is set to **true** if the operation was not completed when the cancel request was received.

7.12 Checking the Execution Status of a Bulk Operation

You can check the status of bulk operations such as update, delete or export using the execution key that you get as a result when the call runs. The status of a completed execution is available for one hour after completion.

To get the status of a bulk operation, use the following interface:

```
GET http://edqserver:8001/edq/cm/bulkstatus/EXECUTIONKEY
```

The EXECUTIONKEY component of the URL is the key returned from a previous bulk delete call. If the key is not known, the request fails with 404 error.

The payload to the POST is a JSON object with the following attributes:

Property	Type	Description
complete	Boolean	Is **true** if the execution is complete. Is **false** otherwise.

Property	Type	Description
failed	Boolean	Is **true** if the execution completed with an error. Is **false** if the execution is not complete or has finished without error.
start	String	The start date and time when the execution started.
end	String	The start date and time when the execution completed. This value is present only if **complete** is **true**.
error	String	The error message if the execution completed with an error. This value is present only if **complete** and **failed** are **true**.
result	Object	The result object containing the count of cases and alerts deleted, updated or exported.

For example:

```
{
  "complete": true,
  "failed": false,
  "start": "2022-08-24T17:34:44.990Z",
  "end": "2022-08-24T17:34:45.448Z",
  "result": {
    "count": 20
  }
}
```

7.13 Executing Multiple Filters for Load Testing

To execute multiple saved filters and report timing, use the following interface:

```
POST http://edqserver:8001/edq/cm/filtertest?[?sql=false]
```

The payload to the POST is a JSON object that includes the following attributes:

Property	Type	Description
filter	String	The name of the filter.
global	Boolean	Set to **true** to specify a global filter.
count	Integer	The repetition count for the filter. The total number of filter executions in one test should not exceed the maximum number of filter execution threads. The default filter execution thread limit is 100. You can set the thread limit by setting the `cm.search.thread.limit` property in `director.properties`.

For example:

```
{
  "tests": [
    { "filter": "NAME1", "global": true, "count": 50 },
    { "filter": "NAME2", "global": true, "count": 50 },
    ...
  ]
}
```

The result of the call is a JSON object:

```
{
  duration: total duration of tests in milliseconds,
  results: [
    { "filter": "NAME1", "global": true, "count": 50, "xids": [internal
execution ids for each invocation] },
    ...
  ]
}
```

The **filter**, **global** and **count** attributes in the **results** array elements reflect the values in the input object. **xids** are the internal execution IDs for each execution of the filter.

7.14 Using the Case Management Administration REST APIs

You can use REST APIs to list and delete case sources, workflows, permissions, and global filters. These APIS are primarily for use with the automated REST testing framework.

The REST interface for the Case Management Administration REST APIs is:

`http://edqserver:8001/edq/cmadmin`

This interface allows you to perform the following tasks:

- Get list of Case Sources
- Delete a Case Source
- Get list of Case Workflows
- Delete a Case Workflow
- Get list of Case Management Dynamic Permissions
- Delete a Case Management Dynamic Permission
- Get list of Case Global Filters
- Delete a Case Global Filter

Get list of Case Sources

The user must have access to the Case Management Administration application.

To get the list of case sources, use the following interface:

```
GET http://edqserver:8001/edq/cmadmin/sources
```

The result is an array of objects each containing these attributes:

Property	Type	Description
name	String	The name of the case source.
prefix	String	The case source prefix.
description	String	The case source description.

Property	Type	Description
permission	String	The permission associated with the case source.

Delete a Case Source

The user must have access to the Case Management Administration application and must have the **Delete Sources** permission.

To delete a case source, use the following interface:

```
DELETE http://edqserver:8001/edq/cmadmin/source/NAME
```

This call deletes the case source that you specify in the *NAME* attribute.

Get list of Case Workflows

The user must have access to the Case Management Administration application.

To get the list of case workflows, use the following interface:

```
GET http://edqserver:8001/edq/cmadmin/workflows
```

The result is an array of objects each containing these attributes:

Property	Type	Description
version	Number	The workflow version.
name	String	The name of the workflow.
label	String	The label of the workflow.
description	String	The description of the workflow

Delete a Case Workflow

The user must have access to the Case Management Administration application and must have the **Delete Sources** permission.

To delete a case workflow, use the following interface:

```
DELETE http://edqserver:8001/edq/cmadmin/workflow/NAME
```

This call deletes the case workflow that you specify in the *NAME* attribute.

Get list of Case Management Dynamic Permissions

The user must have access to the Case Management Administration application.

To get the list of Case Management dynamic permissions, use the following interface:

```
GET http://edqserver:8001/edq/cmadmin/permissions
```

The result is an array of objects each containing these attributes:

Property	Type	Description
key	String	The internal key for the permission.
name	String	The name of the permission.
description	String	The description of the permission.

Delete a Case Management Dynamic Permission

The user must have access to the Case Management Administration application and must have the **Delete Sources** permission.

To delete a Case Management dynamic permission, use the following interface:

```
DELETE http://edqserver:8001/edq/cmadmin/permission/KEY
```

This call deletes the Case Management dynamic permission that you specify in the *KEY* attribute.

Get list of Case Global Filters

The user must have access to the Case Management Administration application, and must have the **Edit Global Filters** permission. If the user does not have the **Edit Global Filters** permission, the results are filtered by the permission associated with each filter, if any.

To get the list of case global filters, use the following interface:

```
GET http://edqserver:8001/edq/cmadmin/filters
```

The result is an array of objects each containing these attributes:

Property	Type	Description
name	String	The name of the filter.
description	String	The description of the filter.
permission	String	The permission associated with the filter.
report	Boolean	Is **true** if the filter generates a report.

Delete a Case Global Filter

The user must have access to the Case Management Administration application, and must have the **Edit Global Filters** permission.

To delete a case global filter, use the following interface:

```
DELETE http://edqserver:8001/edq/cmadmin/filter/NAME
```

This call deletes the case global filter that you specify in the *NAME* attribute.

8

Using the EDQ System Administration API

EDQ provides a set of REST-based interfaces for EDQ system report generation, package import and export, security and encryption configuration, web push notifications, Launchpad application configuration, and users and user group configuration.

To open the System Administration API, select the System Administration Rest API Specification from the Web Services dropdown menu in the Launchpad; the service loads automatically once selected. In this chapter, the EDQ service is assumed to be installed at:

```
http://edqserver:8001/edq
```

This chapter provides a detailed description of these interfaces and the operations that can be performed using these interfaces. It includes the following topics:

- Using REST APIs to Generate System Reports
- Using REST APIs for Importing and Exporting Configuration Objects
- REST API for Security and Encryption
- REST API for Web Push Notifications
- REST Interface for Launchpad Applications
- Using REST APIs for User Group Management
- REST Interface for Creating and Updating Users

8.1 Using REST APIs to Generate System Reports

The EDQ sysreport is a support tool that you can use to retrieve configuration and runtime information from a running system. EDQ provides a REST API to generate the sysreport as a JSON object. You can use the output file for diagnostics and analysis.

In this section, the EDQ service is assumed to be installed at:

```
http://edqserver:8001/edq
```

To run this REST API you need to have system administration permission. Use the following interface the generate the sysreport:

```
GET http://server:port/edq/admin/sysreport[?hideenv=true]
```

The top level attributes in the output JSON are:

Attribute	Description
configdb	The type and version information for the configuration schema database.
configpath	The configuration path as array of strings.
configproperties	The EDQ configuration properties.
environment	The environment variable. This attribute is omitted if **?hideenv=true** is used on the URL.
licencepacks	The selected licence packs as array of strings.

Attribute	Description
memory	The basic memory usage information.
memorypools	The Java memory pool information.
os	The OS and CPU information.
resultsdb	The type and version information for the results schema database.
securityrealms	The name and label for each configured security realm.
server	The server information.
starttime	The server start date and time.
systemproperties	The Java system properties.
version	The EDQ version string.
vm	The Java VM information.
webroot	The path to web root.

You can use this REST API to request for a subset of the full report.

Use the following interface:

```
http://server:port/edq/admin/sysreport/subpath[?hideenv=true]
```

The *subpath* in the URL is used to select a subset within the full report. Each element in the path corresponds to an attribute in the JSON object.

For example,

To retrieve just the information on the configuration schema database, use the following:

```
http://server:port/edq/admin/sysreport/configdb
```

To retrieve just the database type, use the following:

```
http://server:port/edq/admin/sysreport/configdb/dbtype
```

To return the Java version, use the following:

```
http://server:port/edq/admin/sysreport/systemproperties/java.version
```

8.2 Using REST APIs for Importing and Exporting Configuration Objects

EDQ provides a set of REST-based interfaces to automate transfer of configuration between EDQ environments. You can use these REST APIs to package, export, and import the following configuration objects:

- Projects
- Global Reference Data
- Global Data Stores
- Global Published Processors
- Stored Credentials
- Case Sources

- Case Workflows
- Case Permissions
- Global case filters

The package file that is written and read by the APIs is in ZIP format. You can encrypt the configuration objects and stored credentials using a password supplied in the request payload. You can add an "import" Autorun task to import package files at system startup. See Using the Autorun Chores for more information about the `import` chore.

This chapter provides a detailed description of these interfaces and the operations that can be performed using these interfaces. It includes the following topics:

- REST Interface for Packaging Configuration
- JSON Payload Format
- Packaging Task Status Result Format
- Packaging REST API Triggers

8.2.1 REST Interface for Packaging Configuration

The REST interface for working with EDQ packaging, exporting, and importing configuration is

http://edqserver:*port*/edq/package

This interface allows you to perform the following tasks:

- **Exporting configuration**
  You must have the **Package** functional permission to package and export configuration objects. If case management objects are included in the export, you must have access to the Case Management Administration application.

  To package and export configuration objects, use the following interface:

  POST http://edqserver:*port*/edq/package/export

- **Importing configuration**
  You must have create and delete function permissions for all the objects that are being imported. If case management objects are included in the import, you must have access to the Case Management Administration application.

  To import configuration objects, use the following interface:

  POST http://edqserver:*port*/edq/package/import

- **Getting packaging task status**
  Export and import calls return immediately and the packaging task runs asynchronously. The result of package calls is a JSON object containing an "id" attribute, which is the "execution ID" for the task. For example:

  ```
  {"id":"58636dd0-22b6-4d7d-be16-74908d30404f"}
  ```

  To get status information on a packaging task, use the following interface:

  GET http://edqserver:*port*/edq/package/status/*executionid*

  Packaging status is retained in the system for one hour after the task has completed. If you do not specify the execution ID in the get status call, the status for all retained packaging tasks is returned.

  For example:

  ```
  GET http://server/edq/package/status/58636dd0-22b6-4d7d-be16-74908d30404f
  ```

returns status information specific to this execution ID.

```
GET http://server/edq/package/status
```

returns status information for all retained packaging tasks. See Packaging Task Status Result Format for more details.

8.2.2 JSON Payload Format

The REST APIs use a JSON payload to define the configuration objects for the packaging, export, and import.

The following table lists the attributes of the JSON payload:

Attribute	Description
casemanagement	Defines the case management objects to export/import. See `casemanagement` attributes.
destination	**Required**. Destination definition. See `destination` attributes.
password	Encryption password. If present, configuration objects and stored credentials are encrypted in the package file.
preserve	If **true**, existing object are not overwritten. Applies to import only.
label	Label describing packaging task. This information is shown in log messages and included in status reports.
datastores	Selector for global data stores to export/import.
publishedprocessors	Selector for global published processors to export/import.
referencedata	Selector for global reference data to export/import.
projects	Selector for projects to export/import.
storedcredentials	Selector for stored credentials to export/import.

`casemanagement` attributes

The **casemanagement** value is an object with these attributes:

Attribute	Description
sources	Selector for case sources to export/import.
workflows	Selector for case workflows to export/import.
permissions	Selector for case permissions to export/import. Case permissions are matched using the unique internal permission *keys*, which can be examined in the Case Management Administration application.
filters	Selector for global case filters to export/import. Applies to EDQ 14.1.2 and later versions.

`destination` attributes

The payload **destination** value is an object containing these attributes:

Attribute	Description
location	**Required**. File name or URL for the package file. If a non-absolute file name is specified, the location is relative to the "local home" configuration directory.

Attribute	Description
credentials	Name of stored credentials used to authenticate a request to a URL location. Ignored if the location is a file. If the URL requires simple username/password authentication, use "basic" stored credentials.
platformcredentials	If **true**, uses platform credentials for the file upload/download. Supported on Oracle Cloud Infrastructure (OCI) Object Storage, Amazon Web Services (AWS), and Google Cloud Platform (GCP).
method	HTTP method used for uploads. Ignored if the location is a file. The default is **PUT**.

Examples:

Use a file in the EDQ landing area:

```
{ ...
  "destination": {
    "location": "landingarea/pkg/package2.zip"
  },
  ...
}
```

Use OCI object storage, with stored credentials:

```
{ ...
  "destination": {
    "location": "https://objectstorage.us-phoenix-1.oraclecloud.com/n/
mytenancy/b/bucket1/o/package.zip",
    "credentials": "OCI 1"
  },
  ...
}
```

Use AWS S3, with stored credentials:

```
{ ...
  "destination": {
    "location": "https://mystorage.s3.us-east-2.amazonaws.com/package.zip",
    "credentials": "aws1"
  },
  ...
}
```

Use GCP, with stored credentials:

```
{ ...
  "destination": {
    "location": "https://storage.googleapis.com/upload/storage/v1/b/
edqstorage/o?name=package.zip",
    "method": "POST",
    "credentials": "GCP 1"
  },
  ...
}
```

Output Selectors

The selector object used to specify objects to export/import has these attributes:

Attribute	Description
include	Array of "glob-style" patterns used to select objects to include. Use an asterix (*) to match any characters and a question mark (?) to match a single character.
exclude	Array of "glob-style" patterns used to select objects to exclude. Use an asterix (*) to match any characters and a question mark (?) to match a single character.
renames	Object specifying the old to new object renames. On export the rename controls the item written to the package. On import the rename applies to objects in the package and controls the name imported into the system.

In an export task the include and exclude patterns are matched against items in the database. In an import task the patterns are matched against the items in the package file.

Examples:

To include all projects except "Temp1" and those starting with "Test", and to include all non-standard reference data:

```
{ ...
  "projects": {
    "include": ["*"],
    "exclude": ["Temp1", "Test*"]
  },
  "referencedata": {
    "include": ["*"],
    "exclude": ["\\**"]
  },
  ...
}
```

Note the use of the escape character backslash (\) to prevent the first asterix (*) from being treated as a wild card. To conform to JSON syntax, you need to use two backslash (\) characters.

To include everything:

```
{ ...

  "projects": {
    "include": ["*"]
  },

  "referencedata": {
    "include": ["*"]
  },

  "datastores": {
    "include": ["*"]
  },

  "publishedprocessors": {
    "include": ["*"]
  },

  "storedcredentials": {
    "include": ["*"]
  },

  "casemanagement": {

    "sources": {
      "include": ["*"]
    },

    "workflows": {
      "include": ["*"]
    },

    "permissions": {
      "include": ["*"]
    },
    "filters": {
      "include": ["*"]
    }
  }
}
```

8.2.3 Packaging Task Status Result Format

The result of the GET status call is an object in which the attributes are the currently retained execution IDs. The value of each attribute is a status object with these attributes:

Attribute	Description
label	The label copied from the packaging request payload.
type	Whether "import" or "export".
complete	Set to **true** when the packaging process has completed.
failed	Set to **true** if the packaging process failed with an errors.
start	Packaging process start timestamp.
end	Packaging process end timestamp. Set when the process is complete.
status	Current status of packaging process. Cleared when process is compete.
error	Error message if `failed` is **true**.
manifest	Manifest object defining contents of the export package or the items imported. Set on successful completion.

The manifest for an export task is stored as the first entry in the package ZIP file. The **manifest** object contains these attributes:

Attribute	Description
version	Manifest object version. Always 1 in this version.
appversion	Application version, such as "12.2.1.4.4" or "14.1.2.0.0".
encrypted	Set to **true** if the package was generated with a password.
projects	Array of names of projects included in the export or matched during import.
referencedata	Array of names of global reference data included in the export or matched during import.
datastores	Array of names of global data stores included in the export or matched during import.
publishedprocessors	Array of names of global published processors included in the export or matched during import.
storedcredentials	Array of names of stored credentials included in the export or matched during import.
casemanagement	Case management items included in the export or matched during import. sources - Case source names workflows - Case workflow names permissions - Case permission keys

Example status result

An export that is run with this payload:

```
{ ...

{ "label": "Export task1",

  "password": "medusa",

  "destination": {
    "location": "https://objectstorage.us-phoenix-1.oraclecloud.com/n/
devbigdata/b/rde.bucket1/o/example.zip",
    "credentials": "OCI 1"
  },

  "projects": {
    "include": ["adb", "cases", "s*"],
    "exclude": ["snowflake"],
    "renames": {
      "adb": "Oracle ADB"
    }
  },

  "referencedata": {
    "include": ["*Country*"]
  },

  "storedcredentials": {
    "include": ["OCI*", "aws*"],
    "exclude": ["aws?"]
  },
  "casemanagement": {

    "sources": {
      "include": ["CS2", "Issue Remediation"]
    },

    "workflows": {
      "include": ["Issue Remediation*"]
    },

    "permissions": {
      "include": ["Permission?"]
    }
  }
}
```

would return this status after completion:

```
{ "d154aee7-9af4-46ab-af89-84ed6b12109a": {
    "start": "2023-07-13T16:44:17.809Z",
    "label": "Export task1",
    "type": "export",
    "end": "2023-07-13T16:44:26.871Z",
    "manifest": {
      "appversion": "14.1.2.0.0",
      "encrypted": true,
      "projects": [
        "Oracle ADB",
        "cases",
        "scannerbug",
        "scripts",
        "services",
        "sqlserver",
        "srvr"
      ],
      "referencedata": [
        "Country from City",
        "Nationality to Standard Country",
        "Standardize Country Names"
      ],
      "datastores": [],
      "publishedprocessors": [],
      "storedcredentials": [
        "OCI 1",
        "OCI edqtest",
        "OCI edqtest2",
        "aws - oracle",
        "aws rde",
        "aws s3"
      ],
      "casemanagement": {
        "permissions": [
          "Permission1",
          "Permission2"
        ],
        "workflows": [
          "Issue Remediation Alerts",
          "Issue Remediation Cases"
        ],
        "sources": [
          "CS2",
          "Issue Remediation"
        ]
      },
      "version": 1
    },
    "complete": true
  }
}
```

8.2.3.1 Packaging REST API Triggers

The packaging APIs run triggers with these paths:

- `/package/export/start`
- `/package/export/end`
- `/package/import/start`
- `/package/import/end`

The arguments to each trigger call are the task label (from the export/import payload) and the packaging status JSON as a string. The status is passed as a string so that it can be sent easily to a logging or streaming framework without additional parsing.

The following is an example that notifies administrators using a web push when an export is complete:

```
addLibrary("webpush")

function getPath() {
  return "/package/export/end"
}

function run(path, id, env, label, status) {
  if (path.endsWith("/end")) {
    var push = WebPush.create("Export " + label + " complete")

    push.title      = "Export notification"
    push.icon       = "images/logo.png"

    push.groupnames = ["Administrators"]
    push.push()
  }
}
```

8.3 REST API for Security and Encryption

Certain information stored in the EDQ configuration is encrypted for additional security. The encryption and decryption are handled by a pluggable security module. If the security module is replaced or reconfigured in an existing system and there is existing encrypted data, you can define the new security module using the following REST API interfaces:

```
POST to /etc/admin/security/rotateencryption
```

```
POST to /edq/admin/security/migrateencryption
```

For more information, see Configuring EDQ Encryption.

8.4 REST API for Web Push Notifications

Administrators can use the following REST API to notify users of important events such as an impending shutdown:

```
POST http://server/edq/admin/web/push
```

For more information, see REST API for Web Push Notifications.

8.5 REST Interface for Launchpad Applications

Calls in this category are used to list and update the applications that are shown on the launchpad for all users. The calling user must have the **Access Server Administration** and **Set User Application Access** permissions for both calls.

You can perform the following tasks:

- **Get applications**
  To return the current list of published applications, and a list of all applications known to the system, use the following interface:

  ```
  GET http://server:port/edq/admin/web/launchpad/applications
  ```

  The result object contains these attributes:

Attribute	Type	Description
published	Array of strings	List of applications that are currently shown on the launchpad, in display order.
allapplications	Array of application objects	List of all the applications which are known to the system. Each object contains the following values: – label - Display string for application – identifier - Internal identifier Note that the `allapplications` list includes applications that are permission controlled by group membership (such as Case Management) and also applications that are links to simple content, such as Watchlist Screening Help.

- **Update launchpad applications**
  To update launchpad applications, use the following interface:

  ```
  POST http://server:port/edq/admin/web/launchpad/applications
  ```

  The payload is an array of strings listing the identifiers of the applications that should appear on the launchpad, in order.

For example:

```
[ "opsui",
  "director",
  "casemanager",
  "casemanageradmin"
]
```

The result for all POST calls is an object containing these attributes:

Attribute	Description
ok	Success flag. Possible values are `true` or `false`.
message	Error message, present if `ok` is `false`.

8.6 Using REST APIs for User Group Management

EDQ includes a set of REST-based interfaces to manage user groups and external group mappings.

The result for all POST and DELETE calls is an object containing these attributes:

Attribute	Description
ok	Success flag. Possible values are `true` or `false`.
message	Error message, present if `ok` is `false`.

This chapter includes the following topics:

- REST Interface for Creating and Updating Groups
- REST Interface for External Group Mappings

8.6.1 REST Interface for External Group Mappings

Groups in external realms (LDAP etc) are mapped to EDQ internal group so that the permissions and applications available to a user can be determined on login. Calls in this category return the currently defined mappings and update mappings.

Each mapping is represented by an object with these attributes:

Attribute	Description
realm	The realm name as defined in `login.properties`. In an update call this attribute may be omitted if exactly one external realm is defined.
externalgroup	The name of group in the external realm. Always required.
internalgroups	Array of EDQ internal group names. In an update call, existing mappings are removed if this attribute is omitted or empty.

You can perform the following tasks:

- **Get current mappings**
  To retrieve information about the current external group mappings, use the following interface:

```
GET http://server:port/edq/useradmin/externalgroups
```

The result is an array of mapping objects.

- **Update mappings**
  To update information about the current external group mappings, use the following interface:

  ```
  POST http://server:port/edq/useradmin/externalgroups
  ```

  The payload is an array of mapping objects. The calling user must have the **Modify External Group Permissions** functional permission.

  Example:

  ```
  [
    { "realm": "EXAMPLE.COM",
      "externalgroup": "edqgroup1",
      "internalgroups": ["Data Analysts"]
    },
    { "realm": "EXAMPLE.COM",
      "externalgroup": "edqgroup2",
      "internalgroups": ["Match Reviewers", "Executives"]
    }
  ]
  ```

8.6.2 REST Interface for Creating and Updating Groups

The calling user must have the **Access User Administration** functional permission for all user administration calls.

The group object used in requests and responses has these attributes:

Attribute	Type	Description
name	String	The name of the group. Always required.
permissions	Array of strings	Functional permissions associated with group. Values are the internal identifiers representing permissions. In create or update requests, all the permissions in a category can be selected using *prefix:*. For example ***ops:**** or ***user_admin:****.
applications	Array of strings	Applications available to the group. Values are the internal identifiers for launchpad applications.

You can perform the following tasks:

- **Get all groups**
  To return an array of group objects, use the following interface:

  ```
  GET http://server:port/edq/useradmin/groups
  ```

- **Get named group**
  To get information on a single named group, use the following interface:

  ```
  GET http://server:port/edq/useradmin/group/name
  ```

  The result is a group object. If the group does not exist, a 404 response is returned.

- **Create or update single group**

To create or update a single group, use the following interface:

```
POST http://server:port/edq/useradmin/group
```

The payload is a single group object. If the `permissions` or `applications` attributes are not specified, the corresponding values in the group are not changed on update. If the group does not exist, the calling user must have the **Add Group** permission, otherwise the user must have the **Modify Group** permission.

- **Create or update multiple groups**
  To create or update multiple groups, use the following interface:

  ```
  POST http://server:port/edq/useradmin/groups
  ```

  The payload is an array of group objects.

- **Delete named group**
  To delete a named group, use the following interface:

  ```
  DELETE http://server:port/edq/useradmin/group/name
  ```

  The user must have the **Delete Group** permission.

- **Update group**
  To update a group, use the following interface:

  ```
  POST http://server:port/edq/useradmin/updategroup
  ```

  This request can be used to add or remove permissions or applications from an existing group. The payload is a JSON object with these attributes:

Attribute	Description
name	The name of the group. Always required.
permissions	Permission additions and removals.
applications	Application additions and removals.

  The `permissions` and `applications` attributes are object with these attributes:

Attribute	Description
add	Array of values to add to the group.
remove	Array of values to remove from the group.

  Example:

```
{ "name": "Test Group",
  "permissions": {
    "add": [ "user_admin:*", "ops:dnviewallevents"],
    "remove": [ "server_admin:accessserveradmin"]
   },
   "applications": {
     "add": ["opsui" ]
   }
}
```

- **Get lists of supported applications and functional permissions**
  Use the following interface:

  ```
  GET http://server:port/edq/useradmin/permissionsinfo
  ```

Group objects list applications and functional permissions using internal identifiers. The call returns lists of all supported applications and permissions including both the display strings used in the administration web UI and the associated identifiers. Permissions are grouped by the categories shown in the group administration web UI.

The result object contains these attributes:

Attribute	Description
applications	List of supported applications.
functionalpermissions	List of permission categories. Each element contains these attributes: – **categorylabel** - Display string for the category. For example "CM.Static" or "Director". – **prefix** - Prefix associated with the category. For example "data:" or "user_admin:". – **permissions** - List of permissions in the category.

Individual application and permission objects contain these attributes:

Attribute	Description
label	Display string for application or permission.
identifier	Internal identifier.

Display strings are returned in the language implied by the web request.

8.7 REST Interface for Creating and Updating Users

The user object used in requests and responses has these attributes:

Attribute	Type	Description
username	String	The name of the user. Always required.
password	String	The user password. Required when creating a new user.
blocked	Boolean	Set to `true` if the user has been blocked permanently.
fullname	String	The full name of user. Required when creating a new user.
email	String	The e-mail address of user.
organization	String	The name of the user's organization.
telephone	String	The user's telephone number.
forcepasswordchange	Boolean	If `true` forces the user to change their password on the next login.
passwordpolicy	Integer	User password expiration policy. Possible values are: • 0 - Password expiry defined by system configuration • 1 - Password never expires • 2 - Password expires after system defined period

Attribute	Type	Description
lockoutpolicy	Integer	User lockout policy after bad logins. Possible values are: • 0 - Policy defined by system configuration • 1 - No action • 2 - Lockout for 2 minutes • 3 - Lockout for 5 minutes • 4 - Lockout for 15 minutes • 5 - Lockout for 30 minutes • 6 - Lockout for 1 hour • 7 - Lockout for 24 hours • 8 - Lockout permanently
groups	Array of strings	Groups to associate with the user.

The `passwordpolicy` and `lockoutpolicy` attributes correspond to the Password Policy and Invalid Attempts fields in the user definition web UI:

You can perform the following tasks:

- **Get all users**
  To return an array of user objects, use the following interface:

  ```
  GET http://server:port/edq/useradmin/users
  ```

- **Get named user**
  To get information on a single named user, use the following interface:

  ```
  GET http://server:port/edq/useradmin/user/username
  ```

  The result is a user object. If the user does not exist a 404 response is returned.

- **Create or update single user**
  To create or update a single user, use the following interface:

  ```
  POST http://server:port/edq/useradmin/user
  ```

  The payload is a single user object. On update, attributes that are not included in the payload object are not modified in the target user. An attribute which is set to **null** in the payload is cleared in the target user object.

  For example to update a user and clear the e-mail address, use this payload:

  ```
  { "username": "user23",
    "email":    null
  }
  ```

  To update the password for a user, but force a change on next logon, use the following:

  ```
  { "username": "user153",
    "password": "newtemppassword",
    "forcepasswordchange" : true
  }
  ```

  To create a new user, the caller must have the **Add User** permission. To modify a user, the required permissions depend on the detailed changes. The following table lists the permissions you need corresponding to the information you want to update:

Action	Required Permission
Change password	**Change/Reset User Passwords**
Block user permanently	**Block User**
Unblock user	**Unblock User**
Change full name, e-mail address, organization name or telephone number	**Modify User Details**
Change password policy, lockout policy or force change option	**Modify Account Security Options**
Change group membership	**Modify User Group Permissions**

- **Create or update multiple users**
  To create or update multiple users, use the following interface:

  ```
  POST http://server:port/edq/useradmin/users
  ```

  The payload is an array of user objects. To modify users, the required permissions depend on the detailed changes as listed in the table above.

- **Delete named user**
  To delete a user, the user must have the **Delete User** permission. Use the following interface:

  ```
  DELETE http://server:port/edq/useradmin/user/username
  ```

The result for all POST and DELETE calls is an object containing these attributes:

Attribute	Description
ok	Success flag. Possible values are `true` or `false`.
message	Error message, present if `ok` is `false`.

9 Using the Java Messaging Service (JMS) with EDQ

This document describes how to get started using Java Messaging Service (JMS) technology with Oracle Enterprise Data Quality (EDQ). This documentation is intended for system administrators responsible for installing and maintaining EDQ applications.

This chapter includes the following sections:

- Understanding the Message Queue Architecture
- Uses of JMS with EDQ
- Configuring EDQ to Read and Write JMS Messages
- Defining the Interface Files
- Illustrations

9.1 Understanding the Message Queue Architecture

JMS is a flexible technology that allows EDQ to integrate with a variety of different messaging systems and technologies, including WebLogic Server Messaging, WebSphere MQ, Oracle Advanced Queueing (OAQ), and many more.

The EDQ server acts as a ‘client’ of a message queue. The server component may be installed on the same physical server as EDQ, but more commonly the server (or servers) will be remote to it. EDQ is shipped with the client jars needed to connect to either ActiveMQ or Artemis message queue providers. If EDQ is deployed on WebLogic Server, it will also be able to connect natively to WebLogic JMS. Otherwise, for example if EDQ needs to act as a client to an alternative message queue (MQ) system, such as WebSphere MQ, it will need additional client jars to be installed. In this case, the administrator of the MQ system should be consulted to determine which files are needed for a Java application such as EDQ to act as a client.

9.2 Uses of JMS with EDQ

There are two main uses of JMS with EDQ, as listed below: We will focus here on the first use of JMS with EDQ.

- **Consuming and Providing Messages**: You can configure EDQ to read messages from JMS queues, and write messages to them. This can be beneficial where several EDQ servers need to read from a single stream of messages that needs to be persistent to ensure no loss of messages. In this mode, each EDQ server will consume messages from a queue and only ‘commit’ when it has finished processing each message. Note that JMS is best used for Asynchronous communication, where EDQ is not expected to return a response to a calling application for a specific message.
- **Using JMS in Triggers**: EDQ may send JMS messages to other systems, or may consume JMS messages to start triggers (for example to use a JMS queue to distribute batch jobs amongst several EDQ servers;). For more details, see the Using JMS in Triggers.

9.3 Configuring EDQ to Read and Write JMS Messages

JMS interfaces to and from EDQ are configured using XML interface files that define:

- The path to the queue of messages
- Properties that define how to work with the specific JMS technology
- How to decode the message payload into a format understood by an EDQ process (for Message Providers – where EDQ reads messages from a queue), or convert messages to a format expected by an external process (for Message Consumers – where EDQ writes messages to a queue).

The XML files are located in the EDQ Local Home directory (formerly known as the config directory), in the following paths:

- buckets/realtime/providers (for interfaces 'in' to EDQ)
- buckets/realtime/consumers (for interfaces 'out' from EDQ)

Once the XML files have been configured, Message Provider interfaces are available in Reader processors in EDQ and to map to Data Interfaces as 'inputs' to a process, and Message Consumer interfaces are available in Writer processors, and to map from Data Interfaces as 'outputs' from a process as shown in Figures 1 and 2 below:

9.4 Defining the Interface Files

An interface file in EDQ consists of three sections, as follows:

- The `<attributes>` section, defining the shape of the interface as understood by EDQ
- The `<messengerconfig>` section, defining how to connect to the JMS queue or topic
- The `<messagebody>` section, defining how to extract contents of a message (e.g. from XML) into attribute data readable by EDQ (for inbound interfaces), or how to convert attribute data from EDQ to message data (e.g. in XML).

9.4.1 Knowing the <attributes> section

The <attributes> section defines the shape of the interface. It constitutes the attributes that are available in EDQ when configuring a Reader or Writer. For example the following excerpt from the beginning of an interface file configures three string attributes that can be used in EDQ, and their names are:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="jms">
 <attributes>
<attribute type="string" name="messageID"/>
<attribute type="string" name="name"/>
<attribute type="string" name="AccountNumber"/>
</attributes>

[file continues]...
```

EDQ supports all the standard attribute types and they are:

- string
- number
- date
- stringarray
- numberarray
- datearray

9.4.2 Knowing the <messengerconfig> Section

The `<messengerconfig>` section of the interface file configures the settings needed to connect to a given JMS queue or topic on a particular type of JMS technology. The text in `<messengerconfig>` is parsed as a Java properties file and is merged with the set of properties from the `realtime.properties` file in the EDQ configuration path, with settings in the `<messengerconfig>` section overriding any matching settings in`realtime.properties`. The `realtime.properties` file therefore allows the configuration of a number of global properties that do not need to be stated in every JMS interface file.

Note:

- As with all property files stored in the EDQ Local Home directory, properties are themselves merged with a set of 'base' properties in the EDQ Home directory which should not be modified, with any property stated in a file in the Local Home directory used in preference to any property in the EDQ Home directory. For example the`realtime.properties` file in the EDQ Local Home directory will be merged with the file of the same name in the EDQ Home directory to form the final set of properties used.
- The `realtime.properties` values may also be used to set properties for Web Services in EDQ. A prefix of jms. is therefore used (in this file only, not in the `<messengerconfig>` section of a JMS interface file) to set properties for JMS (ws. is used for Web Services).

The messenger configuration properties, specified either in the `<messengerconfig>` section of a JMS interface file, or inherited from `realtime.properties`, are used:

- To configure a JNDI name store to look up queues
- To specify the name of the queue/topic and connection factory
- To supply authentication information to the MQ server, if required

JNDI Setup Properties

- `java.naming.factory.initial` - It denotes the class name of the JNDI initial context factory, used to bootstrap JNDI naming
- `java.naming.provider.url` - the JNDI URL used to connect to the JNDI server or local store

In some cases, authentication is required to connect to the JNDI service. In this case the following properties must be added:

- `java.naming.security.principal` - JNDI user name

- `java.naming.security.credentials` - JNDI password

These JNDI properties are a standard Java feature and for more details, see JNDI Documentation.

JMS Names
The other properties are:

- `connectionfactory` - the JNDI name of the JMS ‘connection factory’. The default for this (if the property is not set) is ‘ConnectionFactory’ which is correct for several MQ servers.
- `destination` - the JNDI name of the JMS queue or topic. There is no default for this and it is always required.

Authentication
If the MQ server requires authentication when making connections, add the below properties:

```
username: connection user name
password: connection password
```

> **Note:**
>
> These properties are used to authenticate against the MQ server and are used to connect to JNDI. In rare cases, both sets may be required.

Notes for specific MQ servers

1. ActiveMQ -
   To connect to an ActiveMQ server, use a property setting as listed below:

   This can be set in the `<messengerconfig>` section to apply to a single interface, or can be set in `realtime.properties` (prefixed with jms.) to set the default for all interfaces.

   ```
   java.naming.provider.url=tcp://host:port
   ```
   ActiveMQ servers generally require connection authentication, so the username and password properties will be required.

   To use ActiveMQ on an EDQ server installed on WebLogic, the property settings above need to be entered into the realtime.properties in the EDQ Local Home directory. (The settings are present, but commented out, in the file in the Home directory, as it is assumed that WebLogic JMS will normally be used where EDQ is installed on WebLogic.)
2. WebLogic JMS -
   When running in a full JavaEE application server, such as WebLogic, Glassfish or WebSphere, the JNDI settings for local JMS are configured automatically and the JNDI factory and url information do not need to be specified in any EDQ configuration file.

   The recommended approach in these cases is to define the JMS configuration within the application server and then just specify the JNDI connection factory and destination names in `<messengerconfig>`.

   WebLogic supports the concept of ‘foreign’ JMS servers. In this case, you define the connection information in the WebLogic Console and expose the destination(s) and connection factory as names in the native WebLogic JNDI store. This works well with Oracle Advanced Queueing (AQ). See below for a snippet of the configuration of a ‘foreign JMS server’ pointing at an AQ schema.

When JMS is configured like this, a typical `<messengerconfig>` section might be:

```
<messengerconfig>
   connectionfactory = jms/cf1
   destination       = jms/queue1
</messengerconfig>
```

Here `jms/cf1and jms/queue1` are JNDI names defined in WebLogic.

9.4.3 Knowing the <messagebody> section

This section uses JavaScript to parse message payloads into attributes that EDQ can use for inbound interfaces, and perform the reverse operation (convert EDQ attribute data into message payload data) for outbound interfaces. A function named 'extract' is used to extract data from XML into attribute data for inbound interfaces, and a function named 'build' is used to build XML data from attribute data.

For more details, refer to Illustrations , wherein the scripts are best illustrated using examples.

9.5 Illustrations

Example 1 – Simple Provider File

The following XML is a simple example of a provider interface file, reading messages from a queue in the path `'dynamicQueues/InputQueue'`.

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="jms">
 <attributes>
<attribute type="string" name="messageID"/>
<attribute type="string" name="name"/>
<attribute type="string" name="AccountNumber"/>
<attribute type="string" name="AccountName"/>
         <attribute type="string" name="Country"/>
 </attributes>
<messengerconfig> destination = dynamicQueues/InputQueue </messengerconfig>
 <incoming>
<messagebody>
<script>
<![CDATA[ function extract(str) { var screeningRequest = new XML(str); var
rec = new Record(); rec.messageID = screeningRequest.individual.messageID;
rec.name = screeningRequest.individual.name; rec.AccountNumber =
screeningRequest.individual.AccountNumber; rec.AccountName =
screeningRequest.individual.AccountName; rec.Country =
screeningRequest.individual.Country; return [rec]; } ]]>
 </script>
 </messagebody>
<eof>
<messageheader name="JMSType" value="t1eof"/>
</eof>
 </incoming>
</realtimedata>
```

Example 2 – Simple Consumer File

The following XML is a simple example of a consumer file representing similar data to the provide file listed above, but with additional attributes, which can be added by an EDQ process:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="jms">
 <attributes>
<attribute type="string" name="messageID"/>
<attribute type="string" name="AccountNumber"/>
<attribute type="string" name="AccountName"/>
<attribute type="string" name="Country"/>
<attribute type="string" name="AccountType"/>
</attributes>
<messengerconfig> destination = dynamicQueues/OutputQueue </messengerconfig>
<outgoing>
<messagebody>
<script>
<![CDATA[ function build(recs) { var rec = recs[0]; var xml = <Request>
<individual> <messageID>{checkNull(rec.messageID)}</messageID>
<AccountNumber>{checkNull(rec.AccountNumber)}</AccountNumber>
<AccountName>{checkNull(rec.AccountName)}</AccountName>
<Country>{checkNull(rec.Country)}</Country>
<AccountType>{checkNull(rec.AccountType)}</AccountType> </individual> </
Request>; return xml.toXMLString(); } function checkNull(value) { return
value != null ? value : ""; } ]]>
     </script>
</messagebody>
</outgoing>
</realtimedata>
```

Example 3 – Date Parsing and Formatting

The following example file shows how to define a DATE attribute type, and include date parsing and formatting when decoding the message payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="jms">
<attributes>
<attribute type="date" name="processingDate"/>
</attributes>
<messengerconfig> destination = dynamicQueues/HoldQueue </messengerconfig>
<incoming>
<messagebody>
 <script>
<![CDATA[ var df = Formatter.newDateFormatter("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
function extract(str) { var screeningRequest = new XML(str); var rec = new
Record();rec.processingDate = date(screeningRequest.processingDate; return
[rec]; } function date(x) { var s = x.text().toString(); return s == "" ?
null : df.parse(s); } ]]>
 </script>
 </messagebody>
 <eof>
<messageheader name="JMSType" value="t1eof"/>
```

```
  </eof>
</incoming>
  </realtimedata>
```

10 Using Apache Kafka with EDQ

This document describes how to get started using the Apache Kafka streaming platform with Oracle Enterprise Data Quality (EDQ). This documentation is intended for system administrators responsible for installing and maintaining EDQ applications. In-depth understanding of Kafka concepts and configuration is assumed.

> **Note:**
>
> This feature is applicable only for EDQ 12.2.1.4.1 release.

This chapter includes the following sections:

10.1 Introduction to Kafka and EDQ

Apache Kafka is a highly performant distributed streaming platform.

EDQ can use the Kafka Consumer API to subscribe to one or more topics and process records as they are published, and can use the Kafka Producer API to publish a stream of records to a topic.

Kafka records contain a value and an optional key. EDQ supports only text values for record values and keys.

10.2 Configuring EDQ to Read and Write Kafka Records

Kafka interfaces to and from EDQ are configured using XML interface files that define:

- The EDQ attributes produced or consumed by the interface
- Properties that define how to configure the Kafka consumer or producer API
- How to decode the record value into EDQ attributes (for Message Providers – where EDQ reads records from a topic), or convert attributes to a value for a Kafka record (for Message Consumers – where EDQ writes messages to a topic).

The XML files are located in the EDQ Local Home directory, in the following paths:

- buckets/realtime/providers (for interfaces 'in' to EDQ)
- buckets/realtime/consumers (for interfaces 'out' from EDQ)

Once the XML files have been configured, Message Provider interfaces are available in Reader processors in EDQ and to map to Data Interfaces as 'inputs' to a process, and

Message Consumer interfaces are available in Writer processors and to map from Data Interfaces as 'outputs' from a process, in the same way as Web Service inputs/outputs and JMS providers/consumers.

10.3 Defining the Interface Files

An interface file in EDQ consists of a `realtimedata` element which defines the message framework. For Kafka interfaces, use the following:

```
<realtimedata messenger=”kafka”>
  …
</realtimedata>
```

The `realtimedata` element contains three subsections:

- The `<attributes>` section, defining the shape of the interface as understood by EDQ
- The `<messengerconfig>` section, defining how to configure the Kafka API
- A message format section defining how a Kafka record is mapped from or to EDQ attributes. For provider interfaces, the element is `<incoming>`; for consumer interfaces, the element is `<outgoing>`.

10.3.1 Understanding the <attributes> section

The `<attributes>` section defines the shape of the interface. It configures the attributes that are available in EDQ when configuring a Reader or Writer. For example, the following excerpt from the beginning of an interface file configures string and number attributes that can be used in EDQ:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="kafka">
  <attributes>
    <attribute type="string" name="messageID"/>
    <attribute type="string" name="name"/>
    <attribute type="number" name="AccountNumber"/>
  </attributes>
…
```

EDQ supports all the standard attribute types. These are:

- string
- number
- date
- stringarray
- numberarray
- datearray

10.3.2 Understanding the <messengerconfig> section

The `<messengerconfig>` section of the interface file configures the Kafka API. The text in `<messengerconfig>` is parsed as a set of Java properties.

Properties prefixed with `conf.` are passed directly to the Kafka API after removing the `conf.` prefix from the key.

Other properties which may be placed in the `<messengerconfig>` section for Kafka are:

- `topic`: For provider interfaces, a comma or space separated list of Kafka topics to subscribe to is required. For consumer interfaces, a single topic name is required.
- `poll`: The interval between polls for new records in milliseconds. This is only applicable for provider interfaces. The default value is 500.
- `key.encoding` and `value.encoding`: The character sets used for converting record keys and values. The character sets must be recognized by `java.nio.charset.Charset.forName`. The default values are implementation-specific.

On a WebLogic installation, authentication information required in configuration properties may be stored in the OPSS credentials store, and used in properties with ${username} and ${password} substitution. Use the `cred.key` and `cred.map` properties to define the credentials store key and map names. The map name defaults to “edq” if omitted.

The following is an example of a complete configuration with credentials store use:

```
<messengerconfig>
  cred.key                      = kafka1
  topic                         = mytopic

  conf.bootstrap.servers        = kserver:9094
  confs.acks                    = all
  conf.max.block.ms             = 1000
  conf.security.protocol        = SASL_SSL
  conf.sasl.mechanism           = PLAIN
  conf.ssl.truststore.location = pathtokeystore.jks
  conf.ssl.truststore.password = pw
  conf.sasl.jaas.config         =
org.apache.kafka.common.security.plain.PlainLoginModule required
  username="${username}" password="${password}";
</messengerconfig>
```

The user name and password for the `jaas.config` property are read from the credentials store with key "kafka1".

Default properties may be defined in the `realtime.properties` file in the EDQ local home directory. Keys for Kafka interfaces in this file are prefixed with "kafka.". For example:

```
kafka.conf.security.protocol = SASL_SSL
```

10.3.3 Understanding the <incoming> or <outgoing> section

The `<incoming>` or `<outgoing>` section defines how record metadata and values are converted to/from EDQ attributes. It consists of the following two subsections:

- The `<messageheaders>` section
- The `<messagebody>` section

10.3.3.1 Understanding the <messageheaders> section

The `<messageheaders>` section is optional. It defines how data outside the record value is converted to/from EDQ attributes.

The format of this section is as follows:

```
<messageheaders>
   <header name="headername" attribute="attributename"/>
  …
</messageheaders>
```

The Kafka interface defines two header names as follows:

- **key**: For providers, the key value from a record is stored in the named EDQ attribute. For consumers, the value of the EDQ attribute is used as the record key on publish.
- **topic**: The name of the Kafka topic on which a record was received is stored in the EDQ attribute. It is only applicable for consumers. This is useful if the interface is defined to subscribe to a number of different topics.

10.3.3.2 Understanding the <messagebody> section

The `<messagebody>` section defines how the text value in a Kafka message is converted from/to EDQ attributes. The element is followed by a subsection defining the conversion mechanism. The following conversion mechanisms are supported:

- JSON Conversion
- Script Conversion

JSON Conversion

The record value is expected to be in JSON format. The nested elements define the mapping between JSON attributes and EDQ attributes.

The format is as follows:

```
<json [multirecord="true or false"] [defaultmappings="true or false"]>
  <mapping attribute="attributename" path="jsonattributename"/>
…
</json>
```

If the `defaultmappings` attribute is omitted or set to **true**, automatic mappings are created from the interface EDQ attributes to JSON attributes.

If the `multirecord` attribute is set to **true**, consumer interfaces expect the JSON input to be an array and provider interfaces to generate a JSON array.

Assuming the attributes shown in the `<attributes>` section description, here are some JSON conversion examples:

Example 1:

An example of a simple JSON conversion in which everything is automatic is as follows:

```
<json/>
```

A simple conversion with automatic mappings expects and generates values like:

```
{ "messageID": "x123",
  "name": "John Smith",
  "AccountNumber": 34567
}
```

Example 2:

An example of a multirecord JSON conversion with a mapping is as follows:

```
<json multirecord=”true”>
  <mapping attribute=”AccountNumber” path=”accno”/>
</json>
```

A multirecord conversion with a single attribute name mapping expects and generates values like:

```
[{ "messageID": "x123",
  "name": "John Smith",
  "accno": 34567
},
…
]
```

Script Conversion

To support more complex conversions, for example XML parsing, a JavaScript script can be provided to process the record value.

In provider interfaces, the script must define a function named "extract" which takes the string record value as an argument. The script should return an array of **Record** objects with attribute names matching EDQ attributes.

The following is an example parsing some XML, using the E4X XML processing API in Rhino JavaScript:

```
<script>
<![CDATA[
  function extract(str) {
    var r = new Record()
    var x = new XML(XMLTransformer.purifyXML(str));

    r.messageID     = x.ID
    r.name          = x.Accname
    r.accountNumber = parseInt(x.Accnumber)
    return [r];
  }
 ]]>
</script>
```

In consumer interfaces, the script must define a function named "build" which takes an array of **Record** objects and returns the text value.

The following is an example generating some XML:

```
<script>
  <![CDATA[
  function build(recs, mtags) {
    var rec = recs[0];

    var xml =
      <response xmlns="http://www.datanomic.com/ws">
       <sum>{rec.sum}</sum>
      </response>;

    return xml.toXMLString();
  }
  ]]>
</script>
```

In a multi-record response, the default behaviour is to call the script for each record. If `<script multirecord=”true”>` is used, the build function is called once with all the records in the message.

For more details, refer to Illustrations, which provides an example of a provider interface file using default JSON conversion.

10.4 Illustrations

The following XML is a simple example of a provider interface file, using default JSON conversion:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="kafka">

  <attributes>
    <attribute type="string" name="messageID"/>
    <attribute type="string" name="name"/>
    <attribute type="string" name="AccountNumber"/>
    <attribute type="string" name="AccountName"/>
    <attribute type="string" name="Country"/>
  </attributes>

  <messengerconfig>
    cred.key                      = kafka1
    topic                         = mytopic

    conf.bootstrap.servers        = kserver:9094
    confs.acks                    = all
    conf.max.block.ms             = 1000
    conf.security.protocol        = SASL_SSL
    conf.sasl.mechanism           = PLAIN
    conf.ssl.truststore.location = mykeystore,jks
    conf.ssl.truststore.password = pw
    conf.sasl.jaas.config        =
org.apache.kafka.common.security.plain.PlainLoginModule required
    username="${username}" password="${password}";
  </messengerconfig>

  <incoming>
    <messagebody>
      <json/>
    </messagebody>
  </incoming>
</realtimedata>
</realtimedata>
```

11

Using Amazon Simple Queue Service (Amazon SQS) with EDQ

This document describes how to get started using the Amazon Simple Queue Service (Amazon SQS) technology with Oracle Enterprise Data Quality (EDQ). This documentation is intended for system administrators responsible for installing and maintaining EDQ applications.

This chapter includes the following sections:

11.1 Introduction to Amazon SQS and EDQ

EDQ realtime provider and consumer buckets can be configured to use Amazon SQS queues for reading and publishing.

11.2 Configuring EDQ to Read and Write Amazon SQS Messages

Amazon SQS interfaces to and from EDQ are configured using XML interface files that define:

- The path to the queue of messages
- Properties that define how to work with the specific Amazon SQS technology
- How to decode the message payload into a format understood by an EDQ process (for Message Providers – where EDQ reads messages from a queue), or convert messages to a format expected by an external process (for Message Consumers – where EDQ writes messages to a queue).

The XML files are located in the EDQ Local Home directory (formerly known as the config directory), in the following paths:

- buckets/realtime/providers (for interfaces ‘in’ to EDQ)
- buckets/realtime/consumers (for interfaces ‘out’ from EDQ)

Once the XML files have been configured, Message Provider interfaces are available in Reader processors in EDQ and to map to Data Interfaces as ‘inputs’ to a process, and Message Consumer interfaces are available in Writer processors, and to map from Data Interfaces as ‘outputs’ from a process.

11.3 Defining the Interface Files

An interface file in EDQ consists of a `realtimedata` element which defines the message framework. For Amazon SQS interfaces, use the following:

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

<realtimedata messenger="sqs">
    ...
</realtimedata>
```

The `realtimedata` element contains three subsections:

- The `<attributes>` section, defining the shape of the interface as understood by EDQ
- The `<messengerconfig>` section, defining how to connect to the Amazon SQS queue.
- A message format section defining how to extract contents of a message (e.g. from XML) into attribute data readable by EDQ (for inbound interfaces), or how to convert attribute data from EDQ to message data (e.g. in XML). For provider interfaces, the element is `<incoming>`; for consumer interfaces, the element is `<outgoing>`.

11.3.1 Understanding the <attributes> section

The <attributes> section defines the shape of the interface. It constitutes the attributes that are available in EDQ when configuring a Reader or Writer. For example, the following excerpt from the beginning of an interface file configures string and number attributes that can be used in EDQ:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="sqs">
<attributes>
   <attribute type="string" name="messageID"/>
   <attribute type="string" name="name"/>
   <attribute type="number" name="AccountNumber"/>
</attributes>

[file continues]...
```

EDQ supports all the standard attribute types and they are:

- string
- number
- date
- stringarray
- numberarray
- datearray

11.3.2 Understanding the <messengerconfig> section

The following properties can be set in the `<messengerconfig>` section:

Property	Description
queue	SQS queue URL (required).
credentials	Stored credentials name used to connect to stream; if omitted, platform (instance) authentication is used.
interval	Interval in milliseconds between polls for message reception. The default is 0 (long polls are used).
proxy	host:port proxy server for HTTPS calls.
deletemode	Controls message deletion after reception. Valid values are: • **reception**: Each message is deleted immediately on reception. This is the default. • **completion**: Each message is deleted when it has completed traversing the processes in the job. • **off**: Messages are not deleted automatically. They must be deleted manually, perhaps using the reception handle in a web service call.
MessageDeduplicationId	Deduplication ID for FIFO queues. May be overridden by header attribute. Two special values are supported: • **$hash**: The deduplication ID is computed as the SHA-256 hash of the message body. • **$uuid**: A random UUID is used as the deduplication ID.
MessageGroupId	Group ID for FIFO queues. May be overridden by header attribute.
DelaySeconds	Sending delay in seconds. May be overridden by header attribute.
MaxNumberOfMessages	Maximum number of messages to receive in one call. The default is 10.
VisibilityTimeout	The duration (in seconds) that the received messages are hidden from subsequent receive requests. The default is set in the queue definition.
WaitTimeSeconds	Wait time in seconds for long poll receive requests. The default is 20s.

Defaults can be set in `realtime.properties` with prefix "sqs.".

11.3.3 Understanding the <incoming> or <outgoing> section

The `<incoming>` or `<outgoing>` section defines how message metadata and values are converted to/from EDQ attributes. It consists of the following two subsections:

- The `<messageheaders>` section
- The `<messagebody>` section

11.3.3.1 Understanding the <messageheaders> section

The <messageheaders> section allows attributes to be mapped to additional sending properties for transmission, and attributes to be set from message metadata on reception.

Refer to the SQS Documentation for more details on each value. A non empty message header value will override a <messengerconfig> property with the same name.

The following standard headers are available:

Header name	Settable	Readable	Type	Description
DelaySeconds	yes	no	number	Sending delay in seconds
MessageDeduplicationId	yes	yes	string	Deduplication ID for FIFO queues
MessageGroupId	yes	yes	string	Group ID for FIFO queues
ApproximateFirstReceiveTimestamp	no	yes	number	Timestamp for first receive from queue
ApproximateReceiveCount	no	yes	number	Number of times message has been received
AWSTraceHeader	no	yes	string	X-Ray trace header
SenderId	no	yes	string	IAM user or role ID
SentTimestamp	no	yes	number	Sending timestamp
SequenceNumber	no	yes	number	Sequence number from SQS
MessageId	no	yes	string	Internal message ID
ReceiptHandle	no	yes	string	Receipt handle, required for manual deletion of messages

Custom message attributes

In addition to the standard headers defined above, the <messageheaders> section can also define custom attributes. Custom attribute names are prefixed with "`messageattribute:`" to distinguish them from standard headers.

Example custom attribute

```
<messageheaders>
  <header name="messageattribute:tel" attribute="telephone" type="string"/>
</messageheaders>
```

11.3.3.2 Understanding the <messagebody> section

This section uses JavaScript to parse message payloads into attributes that EDQ can use for inbound interfaces, and perform the reverse operation (convert EDQ attribute data into message payload data) for outbound interfaces. A function named ‘extract’ is used to extract data from XML into attribute data for inbound interfaces, and a function named ‘build’ is used to build XML data from attribute data.

For more details, refer to Illustrations, which provides an example of a complete provider bucket which can receive JSON messages from a case management filter reporting trigger.

11.4 Illustrations

The following XML is a simple example of a complete provider bucket which can receive JSON messages from a case management filter reporting trigger. The sender ID and message group ID are also returned, along with two custom message attributes, **attr1** and **attr2**.

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

<realtimedata messenger="sqs">
  <attributes>
    <attribute name="filter"      type="string"/>
    <attribute name="type"        type="string"/>
    <attribute name="xaxis"       type="string"/>
    <attribute name="yaxis"       type="string"/>
    <attribute name="server"      type="string"/>
    <attribute name="userid"      type="number"/>
    <attribute name="user"        type="string"/>
    <attribute name="userdisplay" type="string"/>
    <attribute name="start"       type="date" format="iso"/>
    <attribute name="duration"    type="number"/>
    <attribute name="status"      type="string"/>
    <attribute name="sql"         type="string"/>
    <attribute name="arg.type"    type="stringarray"/>
    <attribute name="arg.value"   type="stringarray"/>
    <attribute name="senderid"    type="string"/>
    <attribute name="attr1"       type="string"/>
    <attribute name="attr2"       type="number"/>
    <attribute name="mgid"        type="string"/>
  </attributes>

  <messengerconfig>
    queue       = https://sqs.eu-west-1.amazonaws.com/458503484332/queue1
    credentials = aws1
    deletemode  = completion
  </messengerconfig>

  <incoming>

    <messageheaders>
      <header name="SenderId"                attribute="senderid"/>
      <header name="MessageGroupId"          attribute="mgid"/>
      <header name="messageattribute:attr1" attribute="attr1" type="string"/>
      <header name="messageattribute:attr3" attribute="attr2" type="number"/>
    </messageheaders>

    <messagebody>
      <script>
         <![CDATA[
           var simple = ["filter", "type", "xaxis", "yaxis", "server",
"userid", "user", "userdisplay", "duration", "status", "sql"]

           function extract(str) {
             var obj = JSON.parse(str)
             var rec = new Record()
             for (let x of simple) {
               rec[x] = obj[x]
             }

             rec.start = obj.start && new Date(obj.start)

             if (obj.args) {
               rec['arg.type']  = obj.args.map(a => a.type)
               rec['arg.value'] = obj.args.map(a => a.value &&
a.value.toString())
             }

             return [rec];
           }
         ]]>
      </script>
    </messagebody>

  </incoming>

</realtimedata>
```

12 Using Oracle Cloud Infrastructure (OCI) Queue with EDQ

This document describes how to get started using the Oracle Cloud Infrastructure (OCI) Queue service with Oracle Enterprise Data Quality (EDQ). This documentation is intended for system administrators responsible for installing and maintaining EDQ applications.

This chapter includes the following sections:

12.1 Introduction to OCI Queue and EDQ

EDQ realtime provider and consumer buckets can be configured to use OCI Queue service for reading and publishing. This queue service is a much better fit for realtime processing as compared to OCI streams.

12.2 Configuring EDQ to Read and Write OCI Queue Messages

OCI Queue interfaces to and from EDQ are configured using XML interface files that define:

- The path to the queue of messages
- Properties that define how to work with the specific OCI Queue technology
- How to decode the message payload into a format understood by an EDQ process (for Message Providers – where EDQ reads messages from a queue), or convert messages to a format expected by an external process (for Message Consumers – where EDQ writes messages to a queue).

The XML files are located in the EDQ Local Home directory (formerly known as the config directory), in the following paths:

- buckets/realtime/providers (for interfaces 'in' to EDQ)
- buckets/realtime/consumers (for interfaces 'out' from EDQ)

Once the XML files have been configured, Message Provider interfaces are available in Reader processors in EDQ and to map to Data Interfaces as 'inputs' to a process, and Message Consumer interfaces are available in Writer processors, and to map from Data Interfaces as 'outputs' from a process.

12.3 Defining the Interface Files

An interface file in EDQ consists of a `realtimedata` element which defines the message framework. For OCI Queue interfaces, use the following:

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

<realtimedata messenger="ociqueues">
    ...
</realtimedata>
```

The `realtimedata` element contains three subsections:

- The `<attributes>` section, defining the shape of the interface as understood by EDQ
- The `<messengerconfig>` section, defining how to connect to OCI Queue.
- A message format section defining how to extract contents of a message (e.g. from XML) into attribute data readable by EDQ (for inbound interfaces), or how to convert attribute data from EDQ to message data (e.g. in XML). For provider interfaces, the element is `<incoming>`; for consumer interfaces, the element is `<outgoing>`.

12.3.1 Understanding the <attributes> section

The <attributes> section defines the shape of the interface. It constitutes the attributes that are available in EDQ when configuring a Reader or Writer. For example, the following excerpt from the beginning of an interface file configures string and number attributes that can be used in EDQ:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="ociqueues">
<attributes>
   <attribute type="string" name="messageID"/>
   <attribute type="string" name="name"/>
   <attribute type="number" name="AccountNumber"/>
</attributes>

[file continues]...
```

EDQ supports all the standard attribute types and they are:

- string
- number
- date
- stringarray
- numberarray
- datearray

12.3.2 Understanding the <messengerconfig> section

The following properties can be set in the `<messengerconfig>` section:

Property	Description
queue	The queue OCID (required).
credentials	Stored credentials name used to connect to stream; if omitted, platform (instance) authentication is used.
interval	Interval in milliseconds between polls for message reception. The default is 0 (long polls are used).
proxy	host:port proxy server for HTTPS calls.
deletemode	Controls message deletion after reception. Valid values are: • **reception**: Each message is deleted immediately on reception. This is the default. • **completion**: Each message is deleted when it has completed traversing the processes in the job. • **off**: Messages are not deleted automatically. They must be deleted manually, perhaps using the reception handle in a web service call.
limit	Maximum number of messages to receive in one call. The default is 20.
visibilityInSeconds	The duration (in seconds) that the received messages are hidden from subsequent receive requests. The default is set in the queue definition.
timeoutInSeconds	Wait time in seconds for long poll receive requests. If omitted the OCI default of 30s is used.

Defaults can be set in `realtime.properties` with the prefix "ociqueues.".

12.3.3 Understanding the <incoming> or <outgoing> section

The `<incoming>` or `<outgoing>` section defines how message metadata and values are converted to/from EDQ attributes. It consists of the following two subsections:

- The `<messageheaders>` section
- The `<messagebody>` section

12.3.3.1 Understanding the <messageheaders> section

The `<messageheaders>` section allows attributes to be mapped to additional sending properties for transmission, and attributes to be set from message metadata on reception. Refer to the OCI Queue Documentation for more details on each value. A non empty message header value will override a `<messengerconfig>` property with the same name.

The following standard headers are available:

Header name	Settable	Readable	Type	Description
deliveryCount	no	yes	number	The number of times the message has been delivered to a consumer.
expireAfter	no	yes	date	The time after which the message will be automatically deleted.
id	no	yes	string	The internal message ID.
receipt	no	yes	string	The Receipt token that is required for manual deletion of messages.

Header name	Settable	Readable	Type	Description
visibleAfter	no	yes	date	The time after which the message will be visible to other consumers.

12.3.3.2 Understanding the <messagebody> section

This section uses JavaScript to parse message payloads into attributes that EDQ can use for inbound interfaces, and perform the reverse operation (convert EDQ attribute data into message payload data) for outbound interfaces. A function named ‘extract’ is used to extract data from XML into attribute data for inbound interfaces, and a function named ‘build’ is used to build XML data from attribute data.

For more details, refer to Illustrations, which provides an example of a complete provider bucket which can receive JSON messages from a case management filter reporting trigger.

12.3.4 Illustrations

The following XML is a simple example of a complete provider bucket which can receive JSON messages from a case management filter reporting trigger. The internal message ID is also returned.

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

<realtimedata messenger="ociqueues">
  <attributes>
    <attribute name="filter"      type="string"/>
    <attribute name="type"        type="string"/>
    <attribute name="xaxis"       type="string"/>
    <attribute name="yaxis"       type="string"/>
    <attribute name="server"      type="string"/>
    <attribute name="userid"      type="number"/>
    <attribute name="user"        type="string"/>
    <attribute name="userdisplay" type="string"/>
    <attribute name="start"       type="date" format="iso"/>
    <attribute name="duration"    type="number"/>
    <attribute name="status"      type="string"/>
    <attribute name="sql"         type="string"/>
    <attribute name="arg.type"    type="stringarray"/>
    <attribute name="arg.value"   type="stringarray"/>
    <attribute name="mgid"        type="string"/>
  </attributes>

  <messengerconfig>
    queue        =
ocid1.queue.oc1.phx.amaaaaaa7u6obfiaotsz7yuj2zcbc5uhc45evvv7wfwedgertw3543we
     credentials = OCI 1
  </messengerconfig>

  <incoming>

    <messageheaders>
      <header name="id" attribute="mgid"/>
    </messageheaders>
    <messagebody>
      <script>
         <![CDATA[
           var simple = ["filter", "type", "xaxis", "yaxis", "server",
"userid", "user", "userdisplay", "duration", "status", "sql"]

           function extract(str) {
             var obj = JSON.parse(str)
             var rec = new Record()

             for (let x of simple) {
               rec[x] = obj[x]
             }

             rec.start = obj.start && new Date(obj.start)

             if (obj.args) {
               rec['arg.type']  = obj.args.map(a => a.type)
               rec['arg.value'] = obj.args.map(a => a.value &&
a.value.toString())
             }

             return [rec];
           }
         ]]>
      </script>
    </messagebody>

  </incoming>

</realtimedata>
```

13

Using Scripted Global Web Services with EDQ

EDQ 12.2.1.4.4 restores limited support for global web services. This documentation is intended for system administrators responsible for installing and maintaining EDQ applications. This chapter includes the following topics:

13.1 Introduction to Global Web Services and EDQ

EDQ 12.2.1.4.0 and earlier supported "global" SOAP web services, which were installed as jars in the local webservices directory. Support for global web services was removed in EDQ 12.2.1.4.1.

EDQ 12.2.1.4.4 onwards you can configure realtime provider and consumer buckets to use global web services for reading and publishing. These web services bucket files use scripts to decode incoming text payloads and generate outgoing text results. The supported payload formats are JSON (application/json), XML (text/xml), and plain text (text/plain).

Note:

SOAP is not supported.

13.2 Configuring EDQ to Read and Write Web Service Requests

Web service interfaces to and from EDQ are configured using XML interface files that define:

- The URL path for the web service
- How to decode the message payload into a format understood by an EDQ process (for Message Providers – where EDQ reads messages), or convert messages to a format expected by an external process (for Message Consumers – where EDQ writes messages).

The XML files are located in the EDQ Local Home directory (formerly known as the config directory), in the following paths:

- buckets/realtime/providers (for interfaces ‘in’ to EDQ)
- buckets/realtime/consumers (for interfaces ‘out’ from EDQ)

Once the XML files have been configured, Message Provider interfaces are available in Reader processors in EDQ and to map to Data Interfaces as ‘inputs’ to a process, and Message Consumer interfaces are available in Writer processors, and to map from Data Interfaces as ‘outputs’ from a process.

13.3 Defining the Interface Files

An interface file in EDQ consists of a `realtimedata` element which defines the message framework. A web service that returns a result is defined by provider and consumer buckets. The configuration in both must be marked as synchronous. For web services, use the following:

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

<realtimedata messenger="ws" synchronous="true">
    ...
</realtimedata>
```

The `realtimedata` element contains three subsections:

- The `<attributes>` section, defining the shape of the interface as understood by EDQ
- The `<messengerconfig>` section, defining how to connect to the web service endpoint.
- A message format section defining how to extract contents of a message (e.g. from XML) into attribute data readable by EDQ (for inbound interfaces), or how to convert attribute data from EDQ to message data (e.g. in XML). For provider interfaces, the element is `<incoming>`; for consumer interfaces, the element is `<outgoing>`.

13.3.1 Understanding the <attributes> section

The <attributes> section defines the shape of the interface. It constitutes the attributes that are available in EDQ when configuring a Reader or Writer.

Provider scripts generate `record` object from the incoming payload and consumer scripts convert record contents into the outgoing payload. In a provider bucket script, create a record using new `Record()` or `newRecord()`. The second form should be used in Groovy scripts. Then set the attributes with assignment like:

```
rec.id = 23
rec.name = "John Smith"
```

Record objects contain the attributes defined in the bucket's attributes section. For example, the following excerpt from the beginning of an interface file defines the bucket's **attributes**, which generate record objects that contain **id**, **name** and **email** attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="ws">
<attributes>
  <attribute name="id"    id="1" type="number"/>
  <attribute name="name"  id="2" type="string"/>
  <attribute name="email" id="3" type="string"/>
</attributes>

[file continues]...
```

EDQ supports all the standard attribute types and they are:

- string
- number
- date
- stringarray
- numberarray
- datearray

13.3.2 Understanding the <messengerconfig> section

The `messengerconfig` section must include a path property. The value is appended to `http://server/edq/restws/` to form the endpoint for the web service.

The path can contain letters, digits, and underscore (_) characters.

13.3.3 Understanding the <incoming> or <outgoing> section

The `<incoming>` or `<outgoing>` section defines how message metadata and values are converted to/from EDQ attributes. It consists of the following two subsections:

- Understanding the <messageheaders> section
- Understanding the <messagebody> section

The provider and consumer buckets are linked by adding a key attribute to the incoming and outgoing elements:

```
<incoming key="addup1">
<outgoing key="addup1">
```

The same key value must be used in the provider and consumer web service buckets for a single service. Each service should use different key values.

13.3.3.1 Understanding the <messageheaders> section

The `<messageheaders>` section is optional. It defines how data outside the record value is converted to/from EDQ attributes. The format of this section is as follows:

```
<messageheaders>
   <header name=”headername” attribute=”attributename”/>
  …
</messageheaders>
```

13.3.3.2 Understanding the <messagebody> section

This section uses JavaScript to parse message payloads into attributes that EDQ can use for inbound interfaces, and perform the reverse operation (convert EDQ attribute data into message payload data) for outbound interfaces. A function named ‘extract’ is used to extract data from XML into attribute data for inbound interfaces, and a function named ‘build’ is used to build XML data from attribute data.

Provider Definition

The provider bucket XML file uses a script decoder to create attributes from the input payload:

```
<messagebody>
  <script content="...">
   <![CDATA[
     function extract(payload, type, mtags, env) {
       ...
     }
   ]]>
  </script>
</messagebody>
```

The **content** attribute is a comma-separated list of the input types that the provider supports:

- **json**: JSON text with content type *application/json*.
- **dom**: XML text with content type *text/xml*. The XML is parsed to DOM by the web services framework. To decode the DOM in the script use `new XML(payload)`.
- **text**: Plain text with content type *text/plain*.

A web service call with a content type not supported by the script will fail with a 415 error. The actual type used is available in `env.contenttype;`. If the script supports more than one content type, you can used this to determine how to parse the payload.

For more details, refer to Illustrations, which provides examples of a provider bucket XML file.

Consumer Definition

The consumer bucket XML file uses a script encoder to create the result text from input attributes:

```
<messagebody>
  <script content="...">
   <![CDATA[
     function build(recs, mtags, ctype) {
       ...
     }
   ]]>
  </script>
</messagebody>
```

The **content** attribute is a comma-separated list of the output types that the consumer supports:

- **recs** is an array of record objects.
- **ctype** is the content type (json, dom, or text) of the incoming request. If the service can accept more than one content type, use this to create the output in the correct format.

Set multirecord="true" if a request can contain multiple records:

```
<script multirecord="true">
```

For more details, refer to Illustrations, which provides examples of a consumer bucket XML file.

13.4 Illustrations

This is an example of a simple global web service called *addup*, which accepts two numbers and generates the sum. Supported input formats are JSON, XML, and plain text. The path for the service is *addup* and the web service endpoint is:

```
http://server:port/edq/restws/addup
```

The following table illustrates how the input and output formats work based on this example service.

Supported Format	Example Input	Example Ouput
JSON	`{ "n1" : 100, "n2" : 235 }`	`{ "inputs": [ 100, 235 ], "sum": 335 }`
XML	`<request> <n1>100</n1> <n2>223</n2> </request>`	`<response> <inputs> <v>100</v> <v>223</v> </inputs> <sum>323</sum> </response>`
Plain text	`234,129`	`234 + 129 = 363`

Example 1 – Provider File

The following XML is an example of a provider bucket XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="ws" synchronous="true">

  <!-- ================================================= -->
  <!-- The attributes which are created by this provider -->
  <!-- ================================================= -->

  <attributes>
    <attribute name="n1" id="1" type="number"/>
    <attribute name="n2" id="2" type="number"/>
  </attributes>

  <!-- ============================= -->
  <!-- The web service configuration -->
  <!-- ============================= -->

  <messengerconfig>
```

```
    path = addup
  </messengerconfig>

  <!-- ==================================== -->
  <!-- The script which unpacks the payload -->
  <!-- ==================================== -->

  <incoming key="addup1">
    <messagebody>
      <script content="json,text,dom">
       <![CDATA[
         function extract(str, type, mtags, env) {
           if (env.contenttype == "json") {
             var x = JSON.parse(str)
             var r = new Record();

             r.n1 = x.n1
             r.n2 = x.n2

             return [r];
           } else if (env.contenttype == "dom") {
             var x = new XML(str)
             var r = new Record();

             r.n1 = parseInt(x.n1)
             r.n2 = parseInt(x.n2)

             return [r];
           } else {
             var a = str.split(",")
             var r = new Record();

             r.n1 = parseInt(a[0])
             r.n2 = parseInt(a[1])

             return [r];
           }
         }
       ]]>
      </script>
    </messagebody>
  </incoming>

</realtimedata>
```

Example 2 – Consumer File

The following XML is an example of a consumer bucket XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<realtimedata messenger="ws" synchronous="true">

  <!-- ================================================== -->
  <!-- The attributes which are accepted by this consumer -->
  <!-- ================================================= -->
  <attributes>
    <attribute name="n1"  type="number"/>
    <attribute name="n2"  type="number"/>
    <attribute name="sum" type="number"/>
  </attributes>

  <!-- ============================= -->
  <!-- The web service configuration -->
  <!-- ============================= -->

  <messengerconfig>
    path = addup
  </messengerconfig>

  <!-- =================================== -->
  <!-- The script which creates the output -->
  <!-- =================================== -->

  <outgoing key="addup1">
    <messagebody>
      <script multirecord="true">
        <![CDATA[
          function build(recs, mtags, ctype) {
            var rec = recs[0]

            if (ctype == 'json') {
              return JSON.stringify({ inputs: [rec.n1, rec.n2], sum: 
rec.sum }, null, 2)
            } else if (ctype == "dom") {
              return  <response><inputs><v>{rec.n1}</v><v>{rec.n2}</v></
inputs><sum>{rec.sum}</sum></response>.toXMLString()
            } else {
              return rec.n1 + " + " + rec.n2 + " = " + rec.sum
            }
          }
        ]]>
      </script>
    </messagebody>
  </outgoing>

</realtimedata>
```