Oracle® Fusion Middleware

Administering Oracle Enterprise Data Quality

14c (14.1.2.0.0)
G10151-01
December 2024

Oracle Fusion Middleware Administering Oracle Enterprise Data Quality, 14c (14.1.2.0.0)

G10151-01

Contents

Preface

Audience viii

Documentation Accessibility viii

Related Documents viii

Conventions viii

1 Using Autorun to Execute Startup Tasks

1.1 Understanding Autorun 1-1

1.2 Using the Autorun Chores 1-1

1.3 Using the Autorun Scripts 1-2

1.3.1 Examples 1-2

1.4 Understanding the Chore and Rules Schemas 1-5

1.4.1 Understanding the Chores Schema 1-5

1.4.2 Understanding the Rules Schema 1-9

2 Configuring EDQ Email Notifications

2.1 Using SMTP to Send Email Notifications 2-1

2.2 Configuring Email Sessions in WebLogic Administration Console 2-1

2.3 Ensuring that Email is Configured 2-2

3 Configuring EDQ Web Push Notifications

3.1 Setting Up and Registering for Web Push Notifications 3-1

3.2 Generating Web Push Notifications 3-2

3.3 Example Trigger Script 3-3

3.4 REST API for Web Push Notifications 3-4

4 Configuring EDQ Case Management

4.1 Understanding and Adding Extended Attributes 4-1

4.1.1 Default Extended Attributes 4-1

4.1.2 Adding New Extended Attributes 4-2

4.2 Configuring Data Entry Validation 4-2
4.2.1 Checking Predefined List Restrictions 4-3
4.2.2 Checking Regular Expression Restriction 4-3
4.3 Understanding Case Management Configuration Properties 4-4
4.4 Reporting for Case Management Filter Execution 4-5
4.5 Configuring External Storage for Case Management Attachments 4-7
4.5.1 Configure Storage in the File System 4-7
4.5.2 Configure Storage in Cloud Storage Providers 4-8
4.5.3 REST API for External Attachment Storage 4-8
4.5.4 Performance Considerations during Bulk Delete 4-9

5 Tuning EDQ Performance

5.1 Understanding the Properties File 5-1
5.2 Tuning for Batch Processing 5-2
5.3 Tuning for Real-Time Processing 5-2
5.3.1 Tuning Batch Processing On Real-Time Systems 5-2
5.3.2 Tuning Real-Time Thread Numbers 5-2
5.3.3 Tuning I/O Heavy Real-Time Processes 5-3
5.3.4 Example of Tuning Real-Time Processes 5-3
5.4 Tuning JVM Parameters 5-3
5.4.1 Setting the Maximum Heap Memory 5-3
5.5 Tuning Database Parameters 5-4
5.6 Adjusting the Client Heap Size 5-4
5.7 Designing Fast Jobs: General Performance Options 5-5
5.7.1 Streaming Data and Disabling Staging 5-5
5.7.2 Minimized Results Writing 5-9
5.7.3 Disabling Sorting and Filtering 5-10
5.7.4 Resource-Intensive Processors 5-11
5.8 Performance Tuning for Parsing and Matching 5-12
5.8.1 Place Parse and Match processors in their own Processes 5-12
5.8.2 Parsing performance options 5-12
5.8.3 Matching performance options 5-13
5.8.3.1 Optimized Clustering 5-13
5.8.3.2 Disabling Sort/Filter options in Match processors 5-14
5.8.3.3 Minimizing Output 5-15
5.8.3.4 Streaming Inputs 5-16
5.9 Performance Tuning for Address Verification 5-16
5.10 What Makes Processes Slow? Common Pitfalls 5-17
5.10.1 Poor Matching Processor Configuration 5-17
5.10.2 Unnecessary Merge Data Streams Processors 5-17
5.10.3 Doing Too Much in a Single Process 5-17

5.10.4 Using the Script Processor when You Could Use a Core Processor 5-17
5.10.5 Using Matching Processors Unnecessarily 5-17
5.11 Tuning EDQ's Platform 5-18
5.11.1 The Application Server and the Database Repository 5-18
5.11.1.1 Relative Importance of the Application Server and the Database Repository 5-18
5.11.1.2 Database Tuning 5-19
5.11.2 Processor Cores and Process Threads 5-19
5.11.2.1 Process Threads 5-19

6 Using JMX Extensions to Monitor EDQ

6.1 Understanding JMX Binding 6-1
6.2 Understanding JMX Bean Naming 6-1
6.2.1 Reviewing the Example 6-2
6.3 Monitoring Real-Time Processes 6-2
6.3.1 Monitoring the Real-Time Web Service MBeans 6-2
6.3.2 Monitoring the Real-Time MBeans 6-3
6.4 Using REST APIs to Access and Manage MBeans 6-4
6.4.1 Web Enabled MBeans 6-4
6.4.2 REST API for MBeans 6-6

7 Using Triggers

7.1 Overview of the Triggers Functionality 7-1
7.1.1 About Predefined Triggers 7-1
7.1.2 About Custom Triggers 7-1
7.2 Required Skills to Use Triggers 7-2
7.3 Storing Triggers 7-2
7.4 Configuring Triggers Using the Script Trigger API 7-2
7.5 Extending the Configuration of Triggers Using Properties Files 7-3
7.6 Understanding EDQ Trigger Points 7-4
7.7 Understanding TriggerInfo Methods 7-8
7.8 Setting Trigger Levels 7-9
7.9 Using JMS in Triggers 7-10
7.10 Exposing Triggers in a Job Configuration 7-10
7.11 Trigger Examples 7-11

8 Using Case Management Scripting

8.1 Overview of the Case Management Script Library 8-1
8.1.1 casemanager object properties 8-1
8.1.2 case bean properties 8-3

8.1.3 case history bean properties 8-5
8.1.4 Source Data Object properties 8-5
8.2 Case Management Trigger Environment 8-7

9 Using Scripting for User Cache Queries

10 Accessing EDQ Files Remotely

10.1 Using HTTP and HTPS Server to Access EDQ Files 10-1
10.2 Using SFTP Server to Access EDQ Files 10-3

11 Defining Housekeeping Rules

11.1 For the Event Log Table 11-1
11.2 For the Task Status Table 11-1

12 Third-Party License Attributions

13 Limits in EDQ

14 Backing Up and Restoring EDQ Server

15 Configuring Schema Password Expiry Warnings and Wallet Refresh

15.1 Configuring Schema Password Expiry Warnings 15-1
15.2 Configuring Schema Password Reset 15-3
15.3 Configuring Automatic Wallet Refresh 15-5

16 Using the User Garbage Collection Tool to Manage Deleted Users

17 Updating Database Passwords using setpws.jar

18 Using the Local Web Content Directory

18.1 Location of the Local Web Content Directory 18-1

18.2 Populating the Local Web Content Directory 18-1

18.3 Examples 18-2

19 Configuring Secure Script Execution

Preface

This document describes how to administer and configure Oracle Enterprise Data Quality. You can perform a variety of administration tasks to extend the default EDQ configuration.

Audience

This document is intended for system administrators or application developers who are installing the Oracle Enterprise Data Quality. It is assumed that you have a basic understanding of core EDQ concepts, application server and web technology and have a general understanding of Linux, UNIX, and Windows platforms.

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`.

Access to Oracle Support

Oracle customers that 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 about EDQ, see the Oracle Enterprise Data Quality documentation set.

EDQ Documentation Library

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.
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

Using Autorun to Execute Startup Tasks

This chapter provides an introduction to the EDQ autorun functionality, which allows EDQ to load projects and run jobs when the application server starts up. It explains how the autorun functionality is configured, introduces the chore types that can be performed by using the autorun facility and provides examples of autorun scripts.
This chapter includes the following sections:

- Understanding Autorun
- Using the Autorun Chores
- Using the Autorun Scripts
- Understanding the Chore and Rules Schemas

1.1 Understanding Autorun

EDQ can be configured to do the following automatically at startup:

- Perform a range of tasks when the application server starts up. Each task, which is composed of chores, can be configured to be performed every time the application server is started, or just once the next time the application server is started.
- Load and apply purge rules that override the purge settings that are stored in the EDQ server.

To use autorun processing, you place autorun scripts, written in XML, that specify tasks in one of two specific directories in the EDQ installation:

- `startup` directory: Scripts in the `startup` directory are processed every time the EDQ application server starts up.
- `onceonly` directory: Scripts in the `onceonly` directory are processed when the EDQ application server next starts up, and are then moved to the `complete` subdirectory within `onceonly`. Scripts in the `complete` directory are not processed on subsequent start ups.

When the application server starts up, EDQ checks the `onceonly` and `startup` directories for autorun scripts and processes any that are present.

The `startup` and `onceonly` directories are located in the EDQ `autorun` directory in the local configuration directory of the application server, `oedq.local.home`.

1.2 Using the Autorun Chores

Various kinds of autorun chores are available in EDQ, each with a set of XML attributes specific to its function. The chore types and their available attributes are defined by the autorun file XML schema, see Understanding the Chore and Rules Schemas. The chores available are listed in the following table:

Chore Type	What the Chore Does
`httpget`	Downloads files from a web server.

Chore Type	What the Chore Does
`package`	Loads a project from a `.dxi` file into the server, or saves a project on the server into a `.dxi` file. If no nodes are specified then the contents of the whole file, including system level components, are loaded into the server.
`load`	Loads a file, for example a purge rules configuration file. This chore is valid only in the `startup` directory. See the Examples section, to know how to use the `load` chore with the `Rules` schema to load purge rules.
`import`	(EDQ 14.1.2 and later) Imports a package file that is created using the packaging export REST API. The chore XML includes the package import definition, as inline JSON or as a file name. JSON is assumed if the contents of the **manifest** element starts with "{" after the leading and trailing whitespace is removed. File names that are not absolute are located relative to the EDQ local configuration directory. See the Examples section, to know how to use the `import` chore to import package files.
`runjob`	Runs an existing job from Director. Any run labels in a run profile specified in this chore are ignored. (Use `runopsjob` to run a job based on a run label.)
`runopsjob`	Runs an existing job from the EDQ Server Console and requires a run label to be set, either in the run profile or with the `runlabel` attribute.
`dbscript`	Runs a database script against the Director database. This kind of chore must only be used with extreme care, as inappropriately applied scripts may corrupt the underlying database.
`sleep`	Waits for a specified interval before proceeding.

1.3 Using the Autorun Scripts

Autorun scripts are files that contain XML code. The main part of an autorun script consists of a list of chores, each bounded by `<chores>` tags. Each chore is of one of the autorun chore types listed in Using the Autorun Chores and includes a set of attributes that specify the chore to be performed. The attributes available depend on the chore type selected.

The XML schema that is used to structure autorun scripts is shown in full in Understanding the Chore and Rules Schemas.

1.3.1 Examples

This section shows some examples of autorun scripts.

Example 1
The following XML code shows a sample autorun script that instructs EDQ to:

- Download the `23People.dxi` file, overwriting any existing file with the same name.
- Import the `23People` project from the `23People.dxi` file, overwriting any existing project with the same name.
- Run the `23People Excel.23People` job with the `rp1` run profile. Any run label specified in the profile will be ignored, because this is not a `runopsjob` chore.

```
<?xml version="1.0" encoding="UTF-8"?>
 <chores version="1">
  <!-- Get the dxi file -->
    <httpget overwrite="true" todir="dxiland" tofile="23People.dxi">
      <url>http://svn/repos/dev/trunk/benchmark/ benchmark/dxis/23People.dxi</url>
  </httpget>
  <!-- Import the project from the dxi -->
    <package direction="in" dir="dxiland" file="23People.dxi" overwrite="true">
      <node type="project" name="23People"/>
    </package>
    <!-- Run the jobs -->
    <runjob project="23People" job="23People Excel.23People" runprofile="rp1"
      waitforcompletion="true"/>
 </chores>
```

Example 2
The following XML code shows a sample autorun script that shows four different ways to use a `runjob` or `runopsjob` chore to run a job.

```
<?xml version="1.0" encoding="UTF-8"?>
<chores version="1">
  <!-- runs a director job with no runlabel -->
  <runjob project="merge" job="tester" waitforlocks="false"
    waitforcompletion="false" runprofile="x"/>
  <!-- runs an ops job with the runlabel from the runprofile -->
  <runopsjob project="merge" job="tester" waitforlocks="false"
    waitforcompletion="false" runprofile="x" />
  <!-- runs an ops job with the runlabel from the runlabel attribute-->
  <runopsjob project="merge" job="tester" waitforlocks="false"
     waitforcompletion="false" runprofile="x" runlabel="chooseme" />
  <!-- runs an ops job with the runlabel from the runlabel attribute-->
  <runopsjob project="merge" job="tester" waitforlocks="false"
     waitforcompletion="false" runlabel="onlychoice" />
</chores>
```

Example 3
The following XML code shows how to use a load chore to load purge rules.

```
<?xml version="1.0" encoding="UTF-8" ?>
<chores version="1">
  <load file="purgerules.xml" dir="autorun" type="purgeRules" />
</chores>
```

The following are the purge rules in the `purgerules.xml` file that is loaded in the chore specification:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <rules>
  - <rule displayName="testa" enabled="true">
      <purgePeriod period="1" unit="HOURS" />
      <project>aa</project>
      <job>12345</job>
      <runlabelMatcher regex="false" runlabel="ABCD" />
  </rule>
  - <rule displayName="testb" enabled="true">
      <purgePeriod period="1" unit="HOURS" />
      <project>aa</project>
      <job>ABCD</job>
      <runlabelMatcher regex="true" runlabel="^\d{5}$" />
  </rule>
- <rule displayName="testc" enabled="true">
  <purgePeriod period="2" unit="HOURS" />
      <project />
      <job />
      <runlabelMatcher regex="true" runlabel="TEST" />
  </rule>
```

```
- <rule displayName="testd" enabled="true">
      <purgePeriod period="3" unit="WEEKS" />
      <project />
      <job />
      <runlabelMatcher regex="true" runlabel="TEST" />
  </rule>
  - <rule displayName="teste" enabled="false">
      <purgePeriod period="999" unit="MONTHS" />
      <project />
      <job />
      <runlabelMatcher regex="true" runlabel="^\d{5}$" />
  </rule>
  - <rule displayName="testf" enabled="true">
      <purgePeriod period="1" unit="HOURS" />
      <project />
      <job />
      <runlabelMatcher regex="true" runlabel="^\d{5}$" />
  </rule>
  - <rule displayName="testg" enabled="true">
      <purgePeriod period="1" unit="DAYS" />
      <project />
      <job />
      <runlabelMatcher regex="false" runlabel="ABCD" />
  </rule>
</rules>
```

Example 4

The following XML code shows a sample autorun script that shows import definition provided using absolute file name to import a package file.

```
<?xml version="1.0" encoding="UTF-8"?>
<chores version="1">
  <import>
    <manifest>/opt/pkg/pkg2.json</manifest>
  </import>
</chores>
```

Example 5

The following XML code shows a sample autorun script that shows import definition provided as inline JSON to import a package file.

```
<?xml version="1.0" encoding="UTF-8"?>
<chores version="1">
  <import>
    <manifest>
      { "label": "Import project imptest",
        "password": "pass2",
        "destination": {
          "location": "/opt/stuff/testdata/imptest.zip"
        },
        "projects": {
          "include": ["imptest"]
        }
      }
    </manifest>
  </import>
</chores>
```

Example 6

The following XML code shows a sample autorun script that shows import definition provided using the relative file name to import a package file.

```
<?xml version="1.0" encoding="UTF-8"?>
<chores version="1">
  <import>
    <manifest>imports/pkg2.json</manifest>
  </import>
</chores>
```

1.4 Understanding the Chore and Rules Schemas

This section shows the Chores and Rules XML schemas.

1.4.1 Understanding the Chores Schema

This schema explains the chores listed in Using the Autorun Chores.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Chores -->
  <xs:element name="chores">
    <xs:complexType>

      <!--
      List of chores that need to be performed.  The chores will be performed
      in the order
      specified in the xml file
      -->
      <xs:choice minOccurs="0" maxOccurs="unbounded">

        <xs:element name="httpget"     type="httpgetType"/>
        <xs:element name="package"     type="packageType"/>
        <xs:element name="runjob"      type="runjobType"/>
        <xs:element name="runopsjob"   type="runopsjobType"/>
        <xs:element name="dumpdb"      type="dumpdbType"/>
        <xs:element name="dbscript"    type="dbScriptType"/>
        <xs:element name="sleep"       type="sleepType"/>
        <xs:element name="load"        type="loadType"/>
      </xs:choice>

      <!-- Schema version number -->
      <xs:attribute name="version" type="xs:positiveInteger" use="required"/>

    </xs:complexType>
  </xs:element>

  <!-- Base type for chores -->
  <xs:complexType name="choreType">

    <!-- Flag indicating whether we should wait for completion before moving
    on to the next chore. -->
    <xs:attribute name="waitforcompletion" type="xs:boolean"
    use="optional" default="true"/>
  </xs:complexType>

  <!-- HTTP Get chore.  Download the specified urls. -->
  <xs:complexType name="httpgetType">

    <xs:complexContent>
      <xs:extension base="choreType">
        <xs:sequence minOccurs="1" maxOccurs="1">
          <!-- URL to download. -->
          <xs:element name="url" type="xs:string"/>
        </xs:sequence>

        <!-- Filename to download to. -->
        <xs:attribute name="tofile" type="xs:string" use="required"/>

        <!--
        Directory to download the files to.
          - relative path is relative to the config dir
          - absolute path is used as is
          - no path indicates the config dir
        -->
        <xs:attribute name="todir" type="xs:string" use="optional"/>

        <!-- If true existing files are overwritten, otherwise download is
        not performed. -->
        <xs:attribute name="overwrite" type="xs:boolean" use="optional"
        default="true"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- dxi file control chore.  Import or export to/from a dxi file. -->
  <xs:complexType name="packageType">

    <xs:complexContent>
      <xs:extension base="choreType">

        <!-- List of root level nodes to import/export.
        An empty list indicates 'all'. -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="node" type="packageNodeType"/>
        </xs:sequence>

        <!-- dxi filename. -->
        <xs:attribute name="file" type="xs:string" use="required"/>

        <!--
        Directory that the dxi is in.
          - relative path is relative to the config dir
          - absolute path is used as is
          - no path indicates the config dir
        -->
        <xs:attribute name="dir" type="xs:string" use="optional"/>

        <!-- If true existing files/nodes are overwritten,
        otherwise no operation. -->
        <xs:attribute name="overwrite" type="xs:boolean"
        use="optional" default="true"/>

        <!-- Direction: in=import out=export -->
        <xs:attribute name="direction" type="packageDirectionEnum"
        use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Package node for import or export from/to a dxi. -->
  <xs:complexType name="packageNodeType">
```

```
  <!-- the type of the node to process -->
  <xs:attribute name="type" type="nodeTypeEnum" use="required"/>

  <!-- the name of the node to process -->
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<!-- db script control chore.  Runs db script against the configuration database. -->
<xs:complexType name="dbScriptType">

  <xs:complexContent>
    <xs:extension base="choreType">

      <!-- db script filename. -->
      <xs:attribute name="file" type="xs:string" use="required"/>

      <!--
      Directory that the db script is in.
        - relative path is relative to the config dir
        - absolute path is used as is
        - no path indicates the config dir
      -->
      <xs:attribute name="dir" type="xs:string" use="optional"/>

      <!-- The database to run the script against -->
      <xs:attribute name="database" type="databaseEnum" use="required"/>

    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Invoke named job chore.  Run a named job -->
<xs:complexType name="runjobType">
  <xs:complexContent>
    <xs:extension base="choreType">

      <!-- Project name -->
      <xs:attribute name="project" type="xs:string" use="required"/>

      <!-- Job name -->
      <xs:attribute name="job" type="xs:string" use="required"/>

      <!-- Wait for locks flag - default to true -->
      <xs:attribute name="waitforlocks" type="xs:boolean"
      use="optional" default="true"/>

      <!-- Optional run profile -->
      <xs:attribute name="runprofile" type="xs:string" use="optional"/>

    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="runopsjobType">
  <xs:complexContent>
    <xs:extension base="runjobType">

      <!-- Optional run label (will override run profile run label if set) -->
      <xs:attribute name="runlabel" type="xs:string" use="optional"/>

    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexType>

<!--
  Dump the database.
-->
<xs:complexType name="dumpdbType">
  <xs:complexContent>
    <xs:extension base="choreType">

      <!-- Output JMP file for config database -->
      <xs:attribute name="configout" type="xs:string" use="required"/>

      <!-- Output JMP file for results database -->
      <xs:attribute name="resultsout" type="xs:string" use="required"/>

      <!--
      Directory that the JMP files are written to
        - relative path is relative to the config dir
        - absolute path is used as is
        - no path indicates the config dir
      -->
      <xs:attribute name="dir" type="xs:string" use="optional"/>

      <!--
      TODO: Add some filtering to allow dumping of categories of data
      e.g. staged data, results data, case management data, etc.
      -->
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- Load a certain file to do a certain thing. Eg change purge rules. -->
<xs:complexType name="loadType">
  <xs:complexContent>
    <xs:extension base="choreType">
      <!-- type of action to run with file -->
      <xs:attribute name="type" type="loadTypeEnum" use="required"/>

      <!-- filename -->
      <xs:attribute name="file" type="xs:string" use="required"/>

      <!--
      Directory that the file is in.
        - relative path is relative to the config dir
        - absolute path is used as is
        - no path indicates the config dir
      -->
      <xs:attribute name="dir" type="xs:string" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!--  Enumeration of databases  -->
<xs:simpleType name="databaseEnum">
  <xs:restriction base="xs:string">
      <xs:enumeration value="director"/>
      <xs:enumeration value="results"/>
  </xs:restriction>
</xs:simpleType>

<!-- Enumeration of valid node types -->
<xs:simpleType name="nodeTypeEnum">
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="project"/>
      <!-- Probably need to do these sometime
      <xs:enumeration value="resource"/>
      <xs:enumeration value="datastore"/>
      -->
    </xs:restriction>
  </xs:simpleType>

  <!-- Enumeration of packaging direction. -->
  <xs:simpleType name="packageDirectionEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="in"/>
      <xs:enumeration value="out"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- Enumeration of types of things that can be loaded. -->
  <xs:simpleType name="loadTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="purgeRules"/>
      <!-- <xs:enumeration value="schedule"/> -->
    </xs:restriction>
  </xs:simpleType>

    <!-- Sleep chore.  Wait for a while before doing other autorun stuff -->
  <xs:complexType name="sleepType">

    <xs:complexContent>
      <xs:extension base="choreType">

        <!-- seconds to wait. -->
        <xs:attribute name="time" type="xs:integer" use="required"/>

      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

1.4.2 Understanding the Rules Schema

This section describes the Rules schema, which provides the basis for structuring an XML script that specifies EDQ server purge rules. Use the `load` chore to load the script at EDQ startup.

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Common types -->
  <!-- ============ -->
  <xs:include schemaLocation="urn:commontypes.xsd"/>

  <xs:element name="rules" type="rulesType">
    <!-- Rule name must be unique -->
    <xs:key name="rule.name">
      <xs:selector xpath="rules/rule"/>
      <xs:field    xpath="@name"/>
    </xs:key>
  </xs:element>
  <!--  Rules  -->

  <xs:complexType name="rulesType">
    <xs:sequence>
      <xs:element name="rule" type="ruleType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>

    <xs:attribute name="schemaversion" type="xs:positiveInteger"
       use="optional" default="1"/>
  </xs:complexType>

  <xs:complexType name="ruleType">
    <xs:sequence>
      <xs:element name="purgePeriod"     type="periodType"   minOccurs="1"
         maxOccurs="1"/>
      <xs:element name="project"         type="xs:string"    minOccurs="0"
         maxOccurs="1"/>
      <xs:element name="job"             type="xs:string"    minOccurs="0"
         maxOccurs="1"/>
      <xs:element name="runlabelMatcher" type="runlabelType" minOccurs="0"
         maxOccurs="1"/>
    </xs:sequence>

    <!-- name -->
    <xs:attribute name="displayName"    type="xs:string"  use="required"/>
    <!-- whether this rule should be applied -->
    <xs:attribute name="enabled" type="xs:boolean" use="required"/>
  </xs:complexType>

  <!-- Runlabel -->

  <xs:complexType name="runlabelType">
    <xs:attribute name="regex"    type="xs:boolean" use="required"/>
    <xs:attribute name="runlabel" type="xs:string"  use="required"/>
  </xs:complexType>

  <!-- Purge Period -->

  <xs:complexType name="periodType">
    <xs:attribute name="period" type="xs:int" use="optional"/>
    <xs:attribute name="unit" type="periodUnitType" use="required"/>
  </xs:complexType>

  <!-- Purge Unit types -->

  <xs:simpleType name="periodUnitType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="IMMEDIATE"/>
      <xs:enumeration value="HOURS"/>
      <xs:enumeration value="DAYS"/>
      <xs:enumeration value="WEEKS"/>
      <xs:enumeration value="MONTHS"/>
      <xs:enumeration value="NEVER"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

2

Configuring EDQ Email Notifications

This chapter describes how to configure to produce email notifications in a number of situations.

- Using SMTP to Send Email Notifications
- Configuring Email Sessions in WebLogic Administration Console
- Ensuring that Email is Configured

Emails can be sent to EDQ users when relevant issues are created or changed, when relevant cases or alerts in Case Management are added or modified, or when relevant jobs are finished running.

2.1 Using SMTP to Send Email Notifications

To send email notifications, the Simple Mail Transfer Protocol (SMTP) information for your EDQ installation must be entered in the `mail.properties` file. Email authentication from EDQ works with an SMTP server requiring authentication is now supported via WebLogic credentials store. Set the SMTP user name and password as the session user name and password to the property list. This `mail.properties` file is stored in `/oedq_home/notification/smtp`.

1. Copy the `mail.properties` file from its installed location of `edq_home/notification/smtp` to the `notification/smtp` sub-directory of the local configuration directory (`oedq_local_home` by default).

   /*oedq_local_home*/notification/smtp

   This file is in the standard Java `mail.properties` file format, as documented at the JavaMail API documentation website found at `https://javamail.java.net/nonav/docs/api/`.

2. Edit the `mail.properties` file as follows, supplying the name of your SMTP host at the site.

```
enabled = true
mail.transport.protocol = smtp
mail.host = smtp.fully qualified domain name of mail host
auth.username = username
auth.password = password
from.address = edqserver@example.com
```

2.2 Configuring Email Sessions in WebLogic Administration Console

You can also send email notifications by creating the configuration for an email session in the WebLogic console. You can then refer to this by the JNDI name in the mail.properties file. To configure a JNDI-accessible session, see http://docs.oracle.com/middleware/1221/wls/WLACH/taskhelp/mail/CreateMailSessions.html.

```
session = JNDI name of session
from.address = edqserver@example.com
enabled = true
```

Note:

For email notifications to work correctly, you must ensure that the `from.address` property is set to a valid email format for your site. You must also ensure that each of your users who will be receiving email notifications has an email address configured in their profile.

2.3 Ensuring that Email is Configured

To check that email notifications are working correctly, create a test issue in Director and assign it to a user with a configured email address. The user should receive an email with a link to the issue.

3

Configuring EDQ Web Push Notifications

EDQ 12.2.1.4.4 includes support for web notifications. This chapter describes how to configure EDQ to trigger web push notifications.
This chapter includes the following sections:

- Setting Up and Registering for Web Push Notifications
- Generating Web Push Notifications
- Example Trigger Script
- REST API for Web Push Notifications

3.1 Setting Up and Registering for Web Push Notifications

Notifications are generated by triggers and therefore can report on job start/stop/error and other events. These notifications should be used for infrequent events, which may demand immediate attention from users. Note that these notifications are not meant to replace existing methods like emails, which you should use for more frequent events such as case management assignment.

To enable support for web push notifications, add the following setting to *director.properties* and restart the server:

```
web.push.enabled = true
web.push.sub     = mailto:systemadminemailaddress
```

The email address in the `web.push.sub` property identifies an administrative contact for the EDQ installation. It is used only if the browser provider identifies a problem. For example:

```
web.push.sub     = mailto:systemadmins@example.com
```

When you set this property, a new Web Interface system permission called **Register for web push notifications** is available.

You should enable this permission for any user who can register for notifications. For users with this permission a new option called **Enable Notifications** appears in the menu under the user name in the top right of the Launchpad. Make sure that EDQ uses HTTPS connections with a valid SSL certificate.

Selecting **Enable Notifications** will request a subscription from the browser. If this is the first time for the site the browser will ask for confirmation. After the confirmation is granted the subscription information is sent to the EDQ server and persisted in the database.

To generate a notification, the EDQ server must make an HTTPS call to an external endpoint. If a proxy is required to reach the external location, ensure that the Java proxy settings are specified. You can set the proxy settings either using command line settings or in *jvm.properties*. For example:

```
https.proxyHost = proxy.example.com
https.proxyPort = 80
http.nonProxyHosts = *.example.com|localhost
```

3.2 Generating Web Push Notifications

A new webpush library is available for use in script triggers. The basic usage is:

```
addLibrary("webpush")

function getPath() {
  return "/job/(start|end)"
}

function run(path, id, env, ...) {
  var push = WebPush.create("message")
  WebPush.push(push)
}
```

The notification object has these attributes:

Attribute	Description	Default Value
title	Notification title.	Oracle Enterprise Data Quality.
message	Notification body.	
icon	Notification icon, relative to EDQ web root.	images/logo64.png
image	Top image for notification, relative to EDQ web root. This attribute is not supported in Mozilla Firefox and Safari browsers.	
requireInteraction	If set to `true`, the notification remains open and must be closed manually. This attribute is not supported in Mozilla Firefox and Safari browsers.	`false`
usernames	User name filter.	
userids	User ID filter.	
groupnames	Group name filter.	
groupids	Group ID filter.	
projectID	Project ID filter.	

Example:

```
function run(path, id, env, ...) {
  var push = WebPush.create("a message")

  push.title = "System notification"
  push.icon  = "local/images/logo.png"

  WebPush.push(push)
}
```

Normally the notifications are sent to all registered subscriptions. You can filter subscriptions by user, group, and project ID. To apply filtering, set options on the notification before sending. For example:

```
function run(path, id, env, ...) {
  var push = WebPush.create("a message")

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

  push.usernames  = ["username"]
  push.userids    = [1]
  push.groupnames = ["Administrators"]
  push.groupids   = [1]
  push.projectID  = 1

  push.push()
}
```

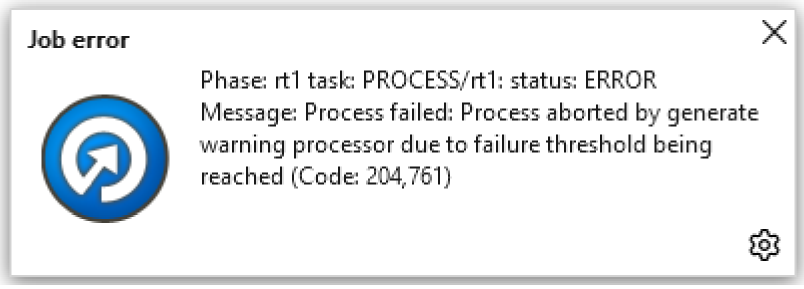
The push will proceed if the user matches any of the user names or IDs, and matches any of the group names or IDs, and has access to the project. Note that the filtering applies to the user who created the subscription originally and not the user, if any, that is currently logged into the site from the browser. The notification image is displayed above the title and message.

3.3 Example Trigger Script

Here's a trigger script that runs on job completion and sends notifications on error.

```
addLibrary("jobNotification");
addLibrary("webpush")

function getPath() {
  return "/job/end";
}

function run(path, id, env, missionbean, map) {

  // Report badness

  if (map.mission.currentStatus != 'FINISHED') {
    var str       = ""
    var sections  = map.sections;
    var tsections = missionbean.taskSections;
    var nsecs     = tsections.length


    for (var k = 0; k < nsecs; k++) {
      var sec = tsections[k]
      var tasks = sec.tasks;
      var xs   = sections['section' + k]

      for (var t = 0; t < tasks.length; t++) {
        var task = tasks[t]
        var tcx  = xs[task.taskId.taskDetail + ":" + task.taskId.version]
        if (tcx != null) {
          var tc = tcx.taskcontext

          str += "Phase: " + sec.sectionName + " task: " +
task.taskId.taskType + "/" + task.taskId.taskDetail + ": status: " +
tc.currentStatus + "\n"

          if (tc.currentStatusMessage) {
            str += "Message: " + tc.currentStatusMessage.getMessage() + "\n"
          }

          str += "\n"
        }
      }
    }

    var push = WebPush.create(str)

    push.title     = "Job error"
    push.projectID = missionbean.projectID

    WebPush.push(push)
  }
}
```

Here's an example notification generated by the script:



Job error

Phase: rt1 task: PROCESS/rt1: status: ERROR
Message: Process failed: Process aborted by generate warning processor due to failure threshold being reached (Code: 204,761)

3.4 REST API for Web Push Notifications

You can use a system administration REST API to trigger push notifications. Administrators can use this to notify users of important events such as an impending shutdown. To trigger push notifications, use the following interface:

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

The user must have the administer system permission. The payload is a JSON object with the same attributes as the script push object described in Generating Web Push Notifications.

Example:

```
{ "title"              : "Sys admin",
  "message"            : "System Admin notification",
  "image"              : "local/agamemnon.jpg",
  "requireInteraction" : true
}
```

4
Configuring EDQ Case Management

This chapter describes how to configure to use Case Management.
This chapter includes the following sections:

- Understanding and Adding Extended Attributes
- Configuring Data Entry Validation
- Understanding Case Management Configuration Properties

Case Management supports the manual investigation of results from data quality processes. Using Case Management, privileged users can manage and review matching results using highly configurable workflows.

The complete set of Case Management extended attributes that are used on an server are configured in the `flags.xml` file in the `oedq_local_home/casemanagement` directory. This file must be modified to add new extended attributes, and to define rules for how these attributes are populated.

An additional property file named `flags.properties` accompanies the base `flags.xml` file and specifies the labels for the extended attributes as they will appear in the graphical user interface (GUI). The settings in this file may be overridden for a specific client language by the creation of additional property files with an ISO 639-1 language code, such as `flags_en.properties` (for English) or `flags_de.properties` (for German). This language code is described at the ISO website found at `http://www.iso.org/iso/home/standards/language_codes.htm`.

If Oracle Watchlist Screening is installed, these files may already exist.

To ensure that Case Management publication works correctly, the `flags.xml` file is overwritten whenever a Case Source is imported using the Case Management Administration application. This is because Case Sources have a dependency on the format of the `flags.xml` file and requires the flags to be indexed and specified in the same way as on the server where the Case Source was defined. Oracle recommends that you back up the file before importing a Case Source in case there are any existing extended attributes in the `flags.xml` file on the server that need to be re-added once the import is complete.

4.1 Understanding and Adding Extended Attributes

This section describes the different types of extended attributes and how to add them for use in Case Management.

4.1.1 Default Extended Attributes

In an initial installation, the `flags.xml` file contains the following two extended attribute (`flag`) example definitions:

```
<f:flag index="1" label="%escalation" type="boolean" default="false"
notnull="true"/>
```

```
<f:flag index="2" label="%priority.score" type="number" readonly="true"/>
```

> **Note:**
>
> The order in which these properties appear in each line may not match this example. The order of properties is immaterial. Also, if Oracle Watchlist Screening is installed, the contents of the `flags.xml` file is different.

4.1.2 Adding New Extended Attributes

To add a new extended attribute, add a line immediately after the existing attribute definitions in the `flags.xml` file, following the same syntax as the existing lines and using the following notes for each property:

Property	Allowed Values	Notes
`index`	Integer	Must be unique for each entry in the file
`label`	Any	The% character is used to indicate that the label for the UI should be retrieved from the `flags.properties` file for the client locale. If the% character is not used, the label will always be exactly as stated (in all languages).
`type`	`number`, `boolean`, or `string`	Controls the data type of the column.
`readonly`	`true` or `false`	Controls whether or not privileged users can edit the value of the extended attribute when editing a Case or Alert
`notnull`	`true` or `false`	Controls whether or not Null values are allowed in the extended attribute. If this is undefined, Null values are allowed (the same as the 'false' setting).
`default`	Any permissible value	Sets the default value of the extended attribute if not set to a specific value.

There is a character limit of 80 characters for extended attributes with a type of 'string'. Values longer than this cannot be inserted as values.

4.2 Configuring Data Entry Validation

You can restrict the format of user-specified data for an extended attribute. The restriction is checked when users edit extended attributes in the Case Management GUI, and when defining possible values to set for an extended attribute in the Workflow editor in Case Management Administration.

The restriction is not checked when cases and alerts are written to Case Management from a process, so it is possible to write invalid values into an extended attribute. The invalid values will appear in error, with an appropriate error message. This designed behavior protects the system against unnecessary job failure.

Restrictions are defined as part of the `flags.xml` file. There are two types of possible restrictions:

- **Predefined list** means that the data to be written is checked against a predefined list of allowed values.
- **Regular expression** means that the data to be written is checked against a regular expression.

4.2.1 Checking Predefined List Restrictions

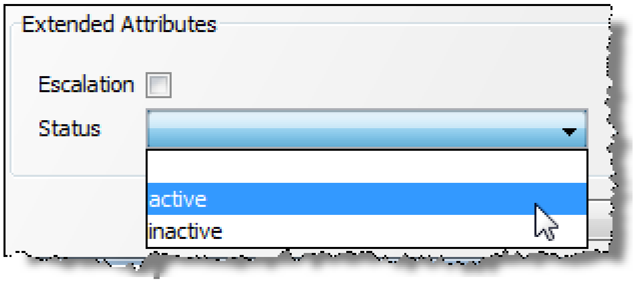
To check that the data being entered into the extended attribute matches a predefined list of possible values, add XML elements in the following format after the definition of the extended attribute (`flag`):

```
<f:restrictions>
<f:predefined>
<f:value>first value</f:value>
<f:value>second value</f:value>
<f:value>third value</f:value>
</f:predefined>
</f:restrictions>
</f:flag>
```

For example, the following XML fragment defines a custom 'Status' extended attribute that allows only the values 'active' and 'inactive':

```
<f:flag index="6" label="Status" type="string" readonly="false">
<f:restrictions>
<f:predefined>
<f:value>active</f:value>
<f:value>inactive</f:value>
</f:predefined>
</f:restrictions>
</f:flag>
```

The extended attribute appears with a list of the valid values in the Case Management Edit Case (or Edit Alert) dialog:



Tip:

In this case, the user can specify a Null value for the `Status` field (as a 'notnull' condition was not set).

4.2.2 Checking Regular Expression Restriction

To check that data being entered into the extended attribute matches a regular expression, add XML elements in the following format after the definition of the extended attribute (`flag`):

```
<f:restrictions>
<f:regex ignorecase="false" matchby="w">
<f:value></f:value>
```

```
</f:regex>
</f:restrictions>
```

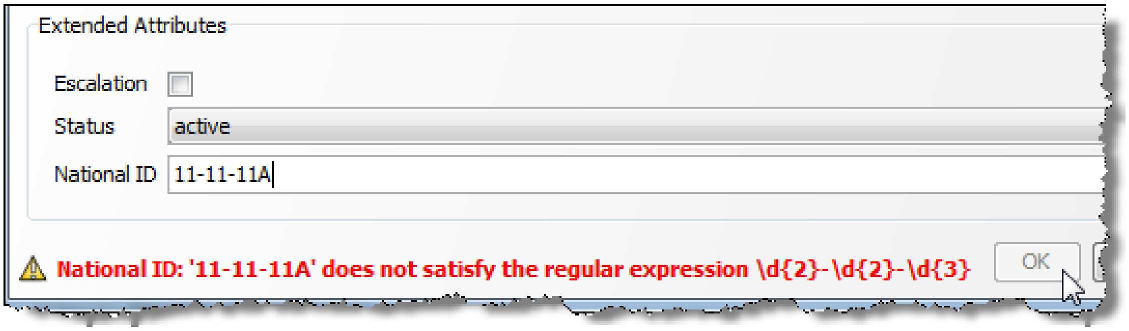
Where: the *`value`* property defines the regular expression, and the `ignorecase` and `matchby` properties defines how it is matched. The possible values for the `matchby` condition are as follows:

Value	Description
`w`	WHOLE - The whole value must match the Regular Expression.
`s`	STARTS - The beginning of the value must match the Regular Expression.
`e`	ENDS - The end of the value must match the Regular Expression.
`c`	CONTAINS - The value must contain a string that matches the Regular Expression.

For example, the following XML fragment defines a custom 'National ID' extended attribute that allows only values in the format `NN-NN-NNN` (2 digits, hyphen, 2 digits, hyphen, 3 digits):

```
<f:flag index="7" label="National ID" type="string" readonly="false" notnull="true">
<f:restrictions>
<f:regex ignorecase="false" matchby="w">
<f:value>\d{2}-\d{2}-\d{3}</f:value>
</f:regex>
</f:restrictions>
</f:flag>
```

The following shows the error message displayed when a user attempts to add a value that does not match the regular expression:



Status: active

National ID: 11-11-11A

⚠ National ID: '11-11-11A' does not satisfy the regular expression \d{2}-\d{2}-\d{3}

It is also possible to customize this error message with the `errormessage` attribute. Either enter a simple text string to be displayed as the error message, or begin the string with a percent (%) symbol to direct the application to look in the `flags.properties` file for a localized value.

For example, the following XML fragment causes the `e1.message` error message to be retrieved from the `flags.properties` file when an error occurs:

```
<f:restrictions><f:regex ignorecase="false" matchby="w"
errormessage="%e1.message"><f:value>\d{3}-\d{2}-\d{4}</f:value></f:regex></
f:restrictions>
```

4.3 Understanding Case Management Configuration Properties

This section lists the main parameters in `director.properties` that are used to configure Case Management.

Parameter	Description0
case.management.fail.on.long.flags	This property controls the Case Management behavior when flag values that are longer than 80 characters are generated. If this property is set to true, the process will generate an error and will stop. If it is set to false, long flag values will be truncated and a warning will be written to the log file. This property is set to false by default.
cm.index.queue.limit	This property controls the maximum size of the index queue limit.
index.directory	This property allows an absolute path for the Lucene index directories to be configured. By default, the index directories are always created within the localhome directory. In some circumstances, these directories can become very large, and storing them in a separate location may facilitate better management of disk space.

4.4 Reporting for Case Management Filter Execution

This appendix describes how to extract information about Case Management filter execution in Oracle Enterprise Data Quality.

EDQ includes trigger points that are called at the start and end of each filter execution. Information about the filter execution is passed to the triggers, which can then be sent to a streaming service such as Java Message Service (JMS), Oracle Cloud Infrastructure Streaming service, and Amazon Kinesis Data Streams.

The trigger paths are as follows:

- /casemanagement/filter/start
- /casemanagement/filter/end

Run functions in triggers accepting these paths are defined as shown below:

```
function run(path, id, env, json) {
 ...
}
```

The **json** argument is a stringified JSON object containing the following attributes:

Attribute	Description
id	Internal filter execution ID
filter	String representation of the filter
type	Possible values are search_filter, report, bulk_update, bulk_delete and export
xaxis	String representation of the report X axis (present for reports only)
yaxis	String representation of the report Y axis (present for reports only)
server	Name of the server where the filter was executed
userid	Internal numeric ID of the user running the filter
user	Name of the user running the filter
userdisplay	Display name of the user running the filter

Attribute	Description
start	Timestamp of filter start
duration	Filter execution time in milliseconds (present for end calls only)
status	Indicates the reason if the filter did not complete successfully
sql	SQL statement used for the filter (SQL execution mode only)
args	Array of SQL bind arguments

The **args** attribute in a report for a SQL execution contains the bind values which replace the '?' placeholders in the SQL text. Each entry in the array contains the following:

type	string, number or date
value	argument value

Example

Suppose a trigger sends filter execution end reports to JMS, an OCI notification service topic, an OCI stream and a Kafka topic.

```
addLibrary("oci");
addLibrary("jms");
addLibrary("kafka");

var oci    = OCI.create("OCI 1")
var topic  = oci.topic("ocid1.onstopic.oc1.phx.aaaaaaaa....")
var stream = oci.stream("ocid1.stream.oc1.phx.aaaaaaaa....")

var props  = config.loadTriggerProperties("jms", "jms\\.properties");
var jms    = JMS.open(Object.assign({}, props, config.loadCredentials("",
props)))
var kprops = config.loadTriggerProperties("kafka", "kafka\\.properties");
var kprod  = KAFKA.producer(kprops.topic, kprops)

function getPath() {
  return "/casemanagement/filter/(start|end)";
}

function run(path, id, env, json) {
  if (path.endsWith("end")) {
    topic.publish("filter done", json)
    stream.publish(null, json)
    jms.send(json)
    kprod.publish(null, json)
  }
}
```

Note that the script trigger framework uses one instance for general queries and additional instances for execution. The first instance will never be triggered and thus resources such as

JMS connections are not needed. The framework creates a built in variable runnable which is false for the single configuration instance. To optimise resources, this can be used as shown below:

```
var jms = runnable && JMS.open(Object.assign({}, props,
config.loadCredentials("", props)))
...
function run(path, id, env, json) {
  ...
  jms && jms.send(json)
}
```

4.5 Configuring External Storage for Case Management Attachments

In EDQ 14.1.2 there is a new option to store Case Management attachment data in the file system or cloud storage. New attachments are stored externally, while existing attachment data remains in the database table. You can also use REST APIs to migrate data to external storage or move external data back into the database.

4.5.1 Configure Storage in the File System

To configure case management storage in the file system, add the following to *director.properties*:

```
case.management.attachment.storage.directory = pathtodirectory
```

If the directory path is not an absolute path, the directory is relative to the "local home" configuration directory. EDQ uses the internal ID of the attachment record in the name of the stored file. You can find the internal ID in the **attachment_id** column in the **dn_caseattachments** record. If you want to store the file using a different name, set the following:

```
case.management.attachment.storage.entryname = filename
```

The *`filename`* value can contain these substitutions:

Substitution	Value
${id}	The internal ID of the attachment. This is always required.
${source}	The source name. The source name must not contain any characters that are not valid in file names.
${caseid}	The internal ID of the associated **dn_case** record.

For example:

```
case.management.attachment.storage.directory = cmatts
```

```
case.management.attachment.storage.entryname = ${source}/${id}
```

Here the attachment data are stored in the directory named *cmatts* in the "local home" configuration directory, with file names constructed from the source name and internal ID.

4.5.2 Configure Storage in Cloud Storage Providers

To configure case management storage in the file system, add the following to *director.properties*:

```
case.management.attachment.storage.url = baseurl
```

EDQ constructs the full attachment URL by appending the entry name to the base URL. The entry name is formed from the **case.management.attachment.storage.entryname** value.

You can set the credentials for upload and downloads to and from the cloud by using any of the following:

- `case.management.attachment.storage.credentials = stored credentials name`
- `case.management.attachment.storage.platform.credentials = true`
  If **case.management.attachment.storage.platform.credentials** is set to *true,* EDQ uses the credentials that are associated with the compute instance. EDQ supports platform credentials for AWS, OCI, and GCP.

Example for OCI Object Storage

```
case.management.attachment.storage.url         = https://
mytenancy.objectstorage.us-phoenix-1.oci.customer-oci.com/n/mytenancy/b/
cmbucket/o/
case.management.attachment.storage.credentials = OCI 1
case.management.attachment.storage.entryname   = cm/${source}/${id}
```

Example for Amazon S3

```
case.management.attachment.storage.url         = https://cmbucket.s3.eu-
west-1.amazonaws.com/cm/
case.management.attachment.storage.credentials = aws1
```

GCP storage has different rules for download and upload URLs. To configure external storage on GCP:

```
case.management.attachment.storage.url    = https://storage.googleapis.com/
storage/v1/b/bucketname/o/
case.management.attachment.storage.cloud  = gcp
```

4.5.3 REST API for External Attachment Storage

When you enable external attachment storage, the existing attachment data remains in the database table. You can use REST APIs that are provided as part of the Case Management Administration endpoint to migrate existing records to external storage, or move extermal data back into the database.

Migrate to/from external storage

POST http://server:port/edq/cmadmin/migrateattachments

POST http://server:port/edq/cmadmin/unmigrateattachments

The payload is a JSON object containing these attributes:

Attribute	Description	Default Value
start	The internal ID of the first attachment to process.	0
count	The number of records to process.	unlimited
batch	The number of records to process in a single database query.	10000

The migration runs asynchronously. The result to the call is a JSON object containing a key corresponding to the execution. The key may be used to poll for execution completion:

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

Get conversion status

```
GET http://server:port/edq/cmadmin/conversionstatus/EXECUTIONKEY
```

This call gets the status of an asynchronous conversion. The EXECUTIONKEY component of the URL is the key that is returned from a previous migrate call.

The result is a JSON object with these attributes:

Attribute	Type	Description
complete	boolean	**true** if conversion is complete.
failed	boolean	**true** if an error occurred during conversion.
start	string	The timestamp when the conversion started.
end	string	The timestamp when the conversion completed.
error	string	Error message, or **null** for a successful conversion.
lastid	integer	The ID of the last attachment that is processed during the conversion.
processed	integer	The number of records processed. If this is zero, it means that all records have been processed.
converted	integer	The number of records that are converted.

The **start** and **count** attributes can be used to perform an incremental migration of a large system over a number of idle periods. Set **start** = 0 for the first call and **lastid+1** for subsequent calls and choose a **count** that allows the process to run in a reasonable time.

4.5.4 Performance Considerations during Bulk Delete

The use of external storage should not affect user actions with attachments. However, when you delete cases and alerts in bulk you must delete the associated external items, whether they are files or cloud storage objects, individually. This could impact the overall execution time of the deletion.

Cloud storage objects are deleted by performing multiple calls simultaneously. The maximum number of deletions executed in a single batch is 10. You can change this by setting the following:

```
case.management.attachment.storage.deletebatch = batchsize
```

5 Tuning EDQ Performance

This chapter describes the server properties that can be used to optimize the performance of the system and how these properties should be configured in various circumstances.
This chapter includes the following topics:

- Understanding the Properties File
- Tuning for Batch Processing
- Tuning for Real-Time Processing
- Tuning JVM Parameters
- Tuning Database Parameters
- Adjusting the Client Heap Size
- Designing Fast Jobs: General Performance Options
- Performance Tuning for Parsing and Matching
- Performance Tuning for Address Verification
- What Makes Processes Slow? Common Pitfalls
- Tuning EDQ's Platform

has a large number of properties that are used to configure various aspects of the system. A relatively small number of these are used to control the performance characteristics of the system.

Performance tuning in is often discussed in terms of **CPU cores**. In this chapter, this refers to the number of CPUs reported by the Java Virtual Machine as returned by a call to the `Runtime.availableProcessors()` method.

5.1 Understanding the Properties File

The tuning controls are exposed as properties in the `director.properties` file. This file is found in the `oedq_local_home` configuration directory.

The available tuning properties are as follows:

Properties	Description
`runtime.threads`	This property determines the number of threads that will be used for each batch job which is invoked. The default value of this property is zero, meaning that the system should start one thread for each CPU core that is available. You can specify an explicit number of threads by supplying a positive, non-zero integer as the value of this property. For example, if you know that you want to start a total of four threads for each batch process, set `runtime.threads` to four.
`runtime.intervalthreads`	This property determines the number of threads that will be used by each process when running in interval mode. This will also define the number of requests that can be processed simultaneously. The default behavior is to run a single thread for each process running in interval mode.

Properties	Description
`workunitexecutor.outputThreads`	This property determines the number of threads that will be used to write data to the results database. These threads service the queue of results and output data for the whole system, and so are shared by all the processes which are running on the system. The default value of this property is zero, meaning that the system should use one output thread for each CPU core that is available. You can specify an explicit number of output threads by supplying a positive, non-zero integer as the value of this property. For example, if you know that you want to use a total of four threads for each batch process, set `workunitexecutor.outputThreads` to 4.

5.2 Tuning for Batch Processing

The default tuning settings provided with are appropriate for most systems that are primarily used for batch processing. Enough threads are started when running a job to use all available cores. If multiple jobs are started, the operating system can schedule the work for efficient sharing between the cores. It is best practice to allow the operating system to perform the scheduling of these kinds of workloads.

5.3 Tuning for Real-Time Processing

When a production system is being used for a significant amount of real time processing, it should not be used for simultaneous batch and real time processing unless the real time response is not critical. Run batch processing only to process data that is required by the real time processes.

5.3.1 Tuning Batch Processing On Real-Time Systems

If batch processing must be run on a system that is being used for real time processing, it is best practice to run the batch work when the real time processes are stopped, such as during a scheduled maintenance window. In this case, the default setting of `runtime.threads` is appropriate.

If it is necessary to run batch processing while real time services are running, set `runtime.threads` to a value that is less than the total number of cores. By reducing the number of threads started for the batch processes, you prevent those processes from placing a load on all of the available cores when they run. Real time service requests that arrive when the batch is running will not be competing with it for CPU time.

5.3.2 Tuning Real-Time Thread Numbers

For most production systems the default value of one for `runtime.intervalthreads` is not appropriate. The default setting implies that, for any given real-time service handled by a process running in interval mode, all requests will be processed sequentially. If four requests for the same service arrive simultaneously, and the average time to process a request is 100 ms, then the first message will be processed after 100 ms, the second after 200 ms, and so on. In addition, all the work will be performed by a single core, meaning that on a four-core machine three of the cores are idle. It is best practice to set `runtime.intervalthreads` to the same as the number of available cores. This configuration allows incoming requests to be processed simultaneously, resulting in a more efficient use of resources and a much faster turnaround speed. The default setting for `runtime.intervalthreads` is adequate for development environments.

5.3.3 Tuning I/O Heavy Real-Time Processes

If a process performs significant I/O, you can try increasing the value of `runtime.intervalthreads` above the number of available cores. When a process performs intensive I/O, there will be times when all the threads are waiting for disk activity to complete, leaving one or more cores idle. By using more active threads than there are cores, you ensure that when one thread stalls for I/O, another thread can utilize the core that the thread was using.

5.3.4 Example of Tuning Real-Time Processes

In this example of how to tune real-time processes, a four-core Intel server is being used to support four different web services. The web services are CPU-intensive and perform minimal amounts of I/O. Some data used by the web services must be updated on a daily basis, which includes running a data preparation process in a batch mode. The web services receive intermittent sets of simultaneous requests. Overnight, the web services are stopped for maintenance and data preparation.

In this scenario, it is appropriate to leave the `runtime.threads` property set to its default value of one thread per CPU core: in this case, four threads. With the goal of performing data preparation in the quickest possible time, and assuming the process is not likely to become I/O bound, you can set the `runtime.intervalthreads` property to four. Using the same number of threads as processes ensures that the maximum number of requests are processed at the same time.

> **Note:**
>
> Increasing the value of `runtime.intervalthreads` means that there will be a significant increase in the memory requirement, particularly at interval turnover.

5.4 Tuning JVM Parameters

JVM parameters should be configured during the installation of EDQ. For more information, see Setting Server Parameters to Support Enterprise Data Quality section present in *Installing and Configuring Oracle Enterprise Data Quality* guide. If it becomes necessary to tune these parameters post-installation to improve performance, follow the instructions in this section.

> **Note:**
>
> All of the recommendations in this section are based on EDQ installations using the Java HotSpot Virtual Machine. Depending on the nature of the implementations, these recommendations may also apply to other JVMs.

5.4.1 Setting the Maximum Heap Memory

If an `OutOfMemory` error message is generated in the log file, it may be necessary to increase the maximum heap space parameter, -`Xmx`. For most use cases, a setting of 8GB is sufficient.

However, large installations may require a higher max heap size, and therefore setting the -Xmx parameter to a value half that of the server memory is the normal recommendation.

5.5 Tuning Database Parameters

The most significant database tuning parameter with respect to performance tuning within is `workunitexecutor.outputThreads`. This parameter determines the number of threads, and hence the number of database connections, that will be used to write results and staged data to the database. All processes that are running on the application server share this pool of threads, so there is a risk of processing becoming I/O bound in some circumstances. If there are processes that are particularly I/O intensive relative to their CPU usage, and the database machine is more powerful than the machine hosting the application server, it may be worth increasing the value of `workunitexecutor.outputThreads`. The additional database threads would use more connections to the database and put more load on the database.

5.6 Adjusting the Client Heap Size

Under certain conditions, client heap size issues can occur; for example, when:

- attempting to export a large amount of data to a client-side Excel file, or
- opening up Match Review when there are many groups.

allows the client heap size to be adjusted using a property in the `blueprints.properties` file.

To double the default maximum client heap space for *all* Java Web Start client applications, create (or edit if it exists) the file `blueprints.properties` in the *local* configuration directory of the server. For more information about the EDQ configuration directories, see "EDQ Directory Requirements" in *Installing Oracle Enterprise Data Quality*.

Add the line:

```
*.jvm.memory = 512m
```

> **Note:**
>
> Increasing this value will cause all connecting clients to change their heap sizes to 512MB. This could have a corresponding impact on client performance if other applications are in use.

To adjust the heap size for a specific application, replace the asterisk, `*`, with the blueprint name of the client application from the following list:

- `director` - (Director)
- `matchreviewoverview` - (Match Review)
- `casemanager` - (Case Management)
- `casemanageradmin` - (Case Management Administration)
- `opsui` - (Server Console)
- `diff` - (Configuration Analysis)
- `issues` - (Issue Manager)

> **Note:**
>
> Dashboard is not a Java Web Start application, and therefore cannot be controlled using this property.

For example, to double the maximum client heap space for Director, add the following line:

```
director.jvm.memory = 512m
```

When doubling the client heap space for more than one application, simply repeat the property; for example, for Director and Match Review:

```
director.jvm.memory = 512m
```

```
matchreviewoverview.jvm.memory = 512m
```

5.7 Designing Fast Jobs: General Performance Options

You can use four general techniques to maximize the performance.

See below for more information.

5.7.1 Streaming Data and Disabling Staging

You can develop jobs that stream imported data directly into processes instead of, or as well as, staging the imported data in the EDQ repository database. Where only a small number of threads are available to a job, streaming data into that job may enable it to process the data more quickly. This is because bypassing the staging of imported data reduces a job's I/O load. Depending on your job's technical and business requirements, and the resources available to it, you may be able to stream data into it, or stage the data and stream it in, to improve performance. Note, however, that where a large number of threads are available to a job, it may run more quickly if you snapshot the data, so that it is all available from the outset. For the avoidance of doubt: you can stream data into a job with or without staging it. However, you cannot disable the staging of imported data unless you are streaming data. A job that streams imported data directly into and out of a process or chain of processes without staging it acts as a pipe, reading records directly from a data store and writing records to a data target.

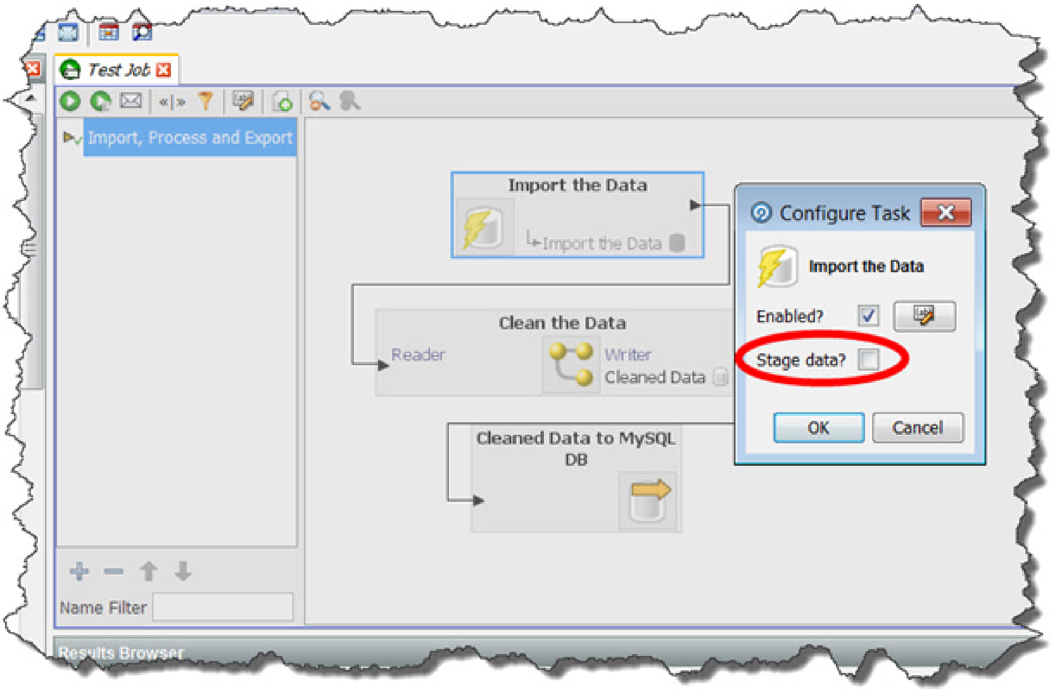
Configuration

To stream data into a process:

- Create a job.
- Add both the snapshot and the process as tasks within the same phase of the job, ensuring that the snapshot is directly connected to the process.

To additionally disable the staging of imported data in the EDQ repository:

- Right-click the snapshot within the job and select **Configure Task**… or **Configure Connector...**
- Within the Configure Task dialog box, de-select the **Stage data?** check box.



Import the Data

Enabled? ☑

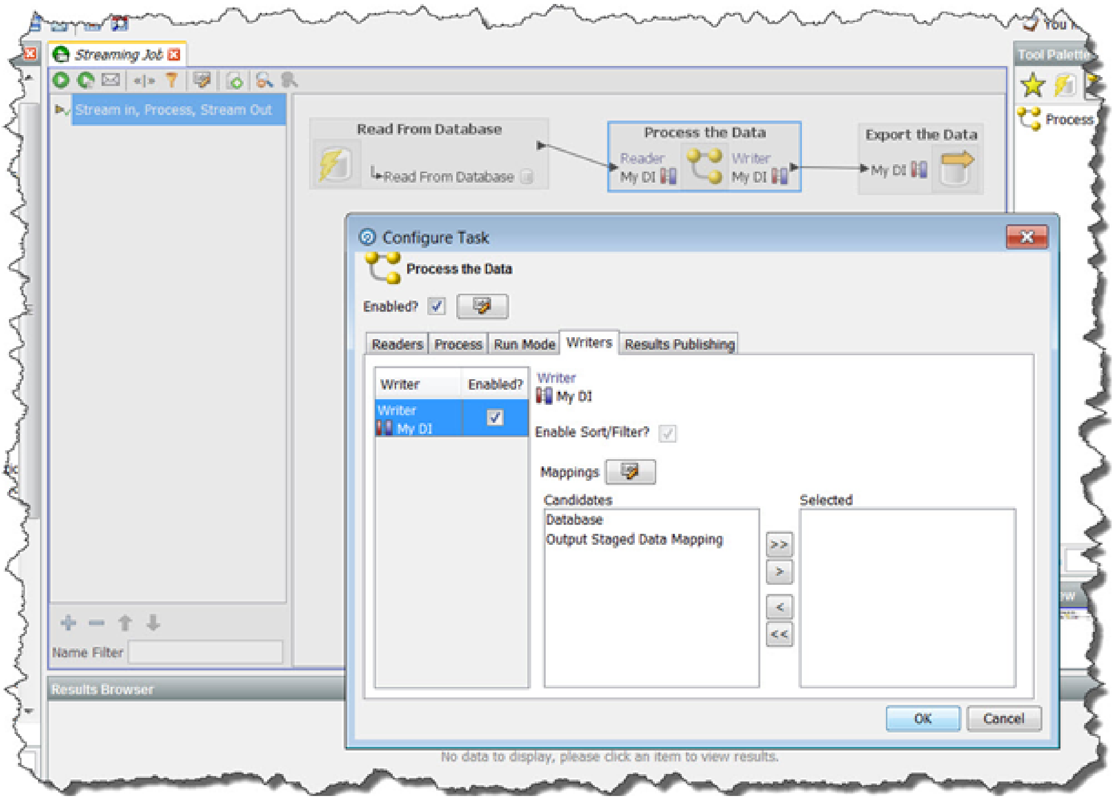
Stage data? □

> **Note:**
>
> Any record selection criteria (snapshot filtering or sampling options) will still apply when streaming data.

To stream an export:

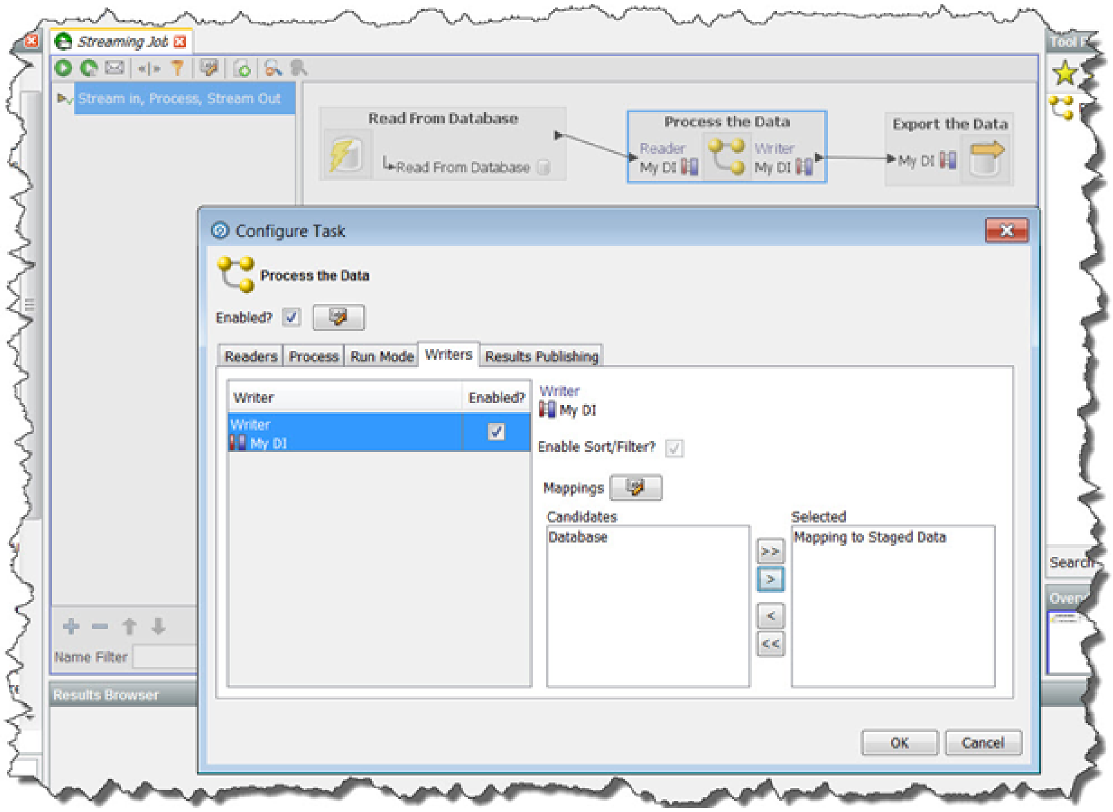
- Create a Process that finishes with a Writer that writes to a Data Interface.
- Create an Export that reads from the same Data Interface.
- Within a Job, add the Process and the Export as tasks in the same phase of the Job.
- Ensure that the Process that writes to your Data Interface is directly connected to the Export.

If you have configured EDQ as outlined above, then, by default, data will not be staged in the repository. This is because you have not selected a Data Interface Output Mapping that points at a set of staged data.



If you want to enable staging of the data that is to be exported:

- Create a Data Interface Mapping that points to a set of staged data.
- Right-click the Process within your Job and select **Configure Task…** or **Configure Connector...**
- In the Configure Task dialog, navigate to the Writers tab.
- Ensure that the Enabled? Check-box beside the writer is ticked (it should be ticked by default).
- Select the Data Interface Mapping that points to a set of staged data.



Enabled? ☑

Writer	Enabled?
Writer My DI	☑

Writer: My DI

Enable Sort/Filter? ☑

Candidates: Database

Selected: Mapping to Staged Data

When to Stage Data, and When to Disable Staging

Whilst designing a process, you will often run it against data that has been staged in the EDQ repository via a snapshot. Streaming data into a job without staging it may be appropriate when:

- You are dealing with a production environment.
- You have a large number of records to process.
- You always want to use the latest records from the source system.

However, streaming a snapshot without staging it is not always the quickest or best option. If you need to run several processes on the same set of data, or if you need to lookup on staged data, it may be more efficient to stage the data via a snapshot as the first task of a job, and then run the dependent processes. If your job has a large number of threads available to it, it may run more quickly if all of the data is staged at the outset. Additionally, if the source system for the snapshot is live, it may be best to run the snapshot in a phase on its own so that the impact on the source system is minimized. In this case, the data will not be streamed into a process, since the snapshot and process need to be directly connected to each other within the same job phase for streaming to occur.

For the avoidance of doubt: if you connect a process directly to a snapshot, then the data will always be streamed into that process, regardless of whether it is also staged in the repository (which is determined by the Stage data? check box). Streaming the data into EDQ and also staging it may, in some cases, be an efficient approach - for example, if the data is used again later in the job.

Streaming an Export

When an export of a set of staged data is configured to run in the same job after the process that writes the staged data, the export will always write records as they are processed, regardless of whether records are also staged in the repository. However, it is possible to realize a small performance gain by disabling staging so that data is only streamed to its target.

You may choose to disable staging of output data:

- For deployed data cleansing jobs.
- If you are writing to an external staging database that is shared between applications. (For example when running a data quality job as part of a larger ETL process, and using an external staging database to pass the data between EDQ and the ETL tool.)

5.7.2 Minimized Results Writing

Minimizing results writing reduces the amount of Results Drilldown data that EDQ writes to the repository from processes, and so saves on I/O.

Each process in EDQ runs in one of three Results Drilldown modes:

- **All** (all records in the process are written in the drilldowns)
- **Sample** (a sample of records are written at each level of drilldown)
- **None** (metrics only are written - no drilldowns will be available)

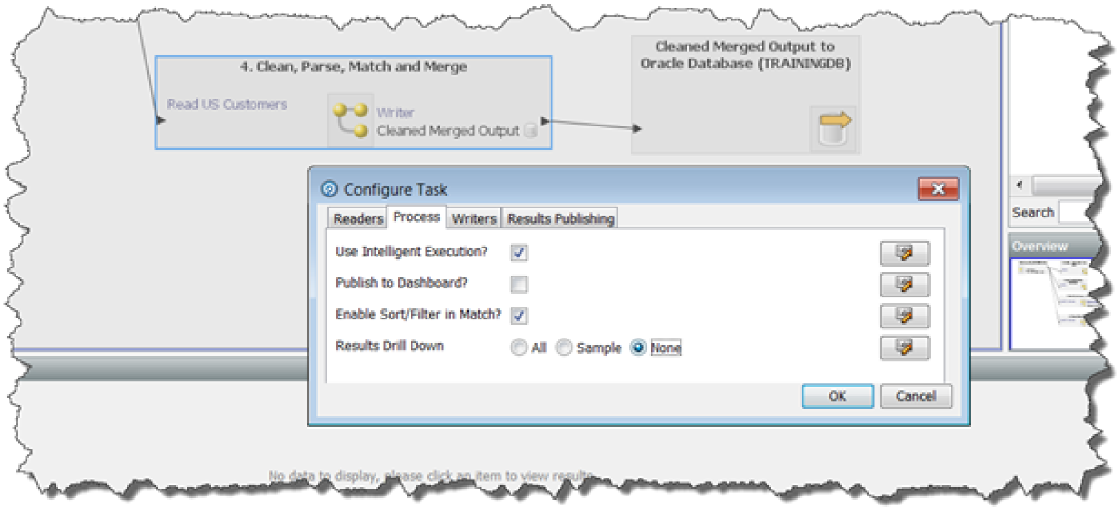
  All mode should be used only on small volumes of data, to ensure that all records can be fully tracked in the process at every processing point. This mode is useful when processing small data sets, or when debugging a complex process using a small number of records.

  Sample mode is suitable for high volumes of data, ensuring that a limited number of records are written for each drilldown. The System Administrator can set the number of records to write per drilldown; by default this is 1000 records. Sample mode is the default when running processes interactively from the Director User Interface.

  None mode should be used to maximize the performance of tested processes that are running in production, and where users will not need to interact with results. None is the default when processes are run within jobs.

  To change the Results Drilldown mode when executing a process, use the Run Preferences screen, or create a Job and double click the process task to configure it.

  For example, the following process is configured so that it does not write drilldown results when it is deployed in production via a job (this is the default when a process is run within a job):


4. Clean, Parse, Match and Merge
Read US Customers
Writer
Cleaned Merged Output
Cleaned Merged Output to Oracle Database (TRAININGDB)
Configure Task
Readers
Process
Writers
Results Publishing
OK
Cancel
Search
Overview
No data to display, please click an item to view results

Use Intelligent Execution? ☑

Publish to Dashboard? □

Enable Sort/Filter in Match? ☑

Results Drill Down ○ All ○ Sample ● None

The Effect of Run Labels

Note that jobs that are run with run labels from either the Server Console user interface or the command line do not generate results drill-downs.

5.7.3 Disabling Sorting and Filtering

When working with large data volumes, it can take a long time to index snapshots and staged data in order to enable users to sort and filter the data in the Results Browser. In many cases, this sorting and filtering capability will not be needed, or will only be needed when working with smaller samples of the data.

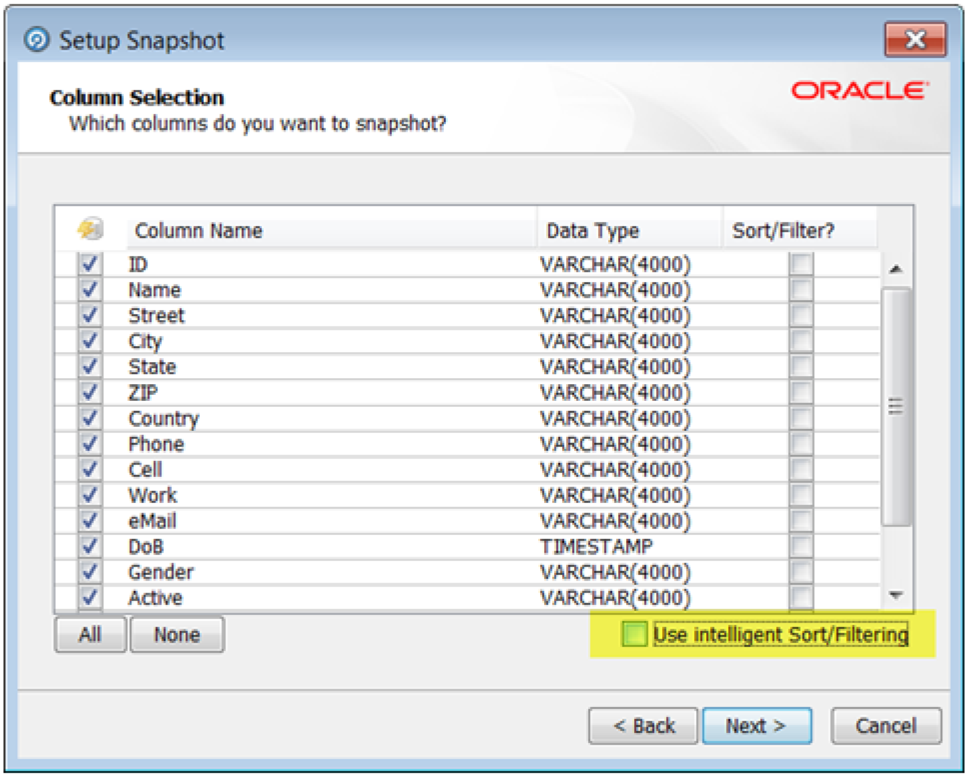
The system applies intelligent sorting and filtering, where it will enable sorting and filtering when working with smaller data sets, but will disable sorting and filtering for large data sets. However, you can choose to override these settings - for example to achieve maximum throughput when working with a number of small data sets.

Snapshot Sort/Filter options

When a snapshot is created, the default setting is to 'Use intelligent Sort/Filtering options', so that the system will decide whether or not to enable sorting and filtering based on the size of the snapshot.

However, if you know that no users will need to sort or filter results that are based on a snapshot in the Results Browser, or if you only want to enable sorting or filtering at the point when the user needs to do it, you can disable sorting and filtering on the snapshot when adding or editing it.

To do this, edit the snapshot, and on the third screen (Column Selection), uncheck the option to Use intelligent Sort/Filtering, and leave all columns unchecked in the Sort/Filter column:



Setup Snapshot

Column Selection
Which columns do you want to snapshot?

	Column Name	Data Type	Sort/Filter?
☑	ID	VARCHAR(4000)	□
☑	Name	VARCHAR(4000)	□
☑	Street	VARCHAR(4000)	□
☑	City	VARCHAR(4000)	□
☑	State	VARCHAR(4000)	□
☑	ZIP	VARCHAR(4000)	□
☑	Country	VARCHAR(4000)	□
☑	Phone	VARCHAR(4000)	□
☑	Cell	VARCHAR(4000)	□
☑	Work	VARCHAR(4000)	□
☑	eMail	VARCHAR(4000)	□
☑	DoB	TIMESTAMP	□
☑	Gender	VARCHAR(4000)	□
☑	Active	VARCHAR(4000)	□

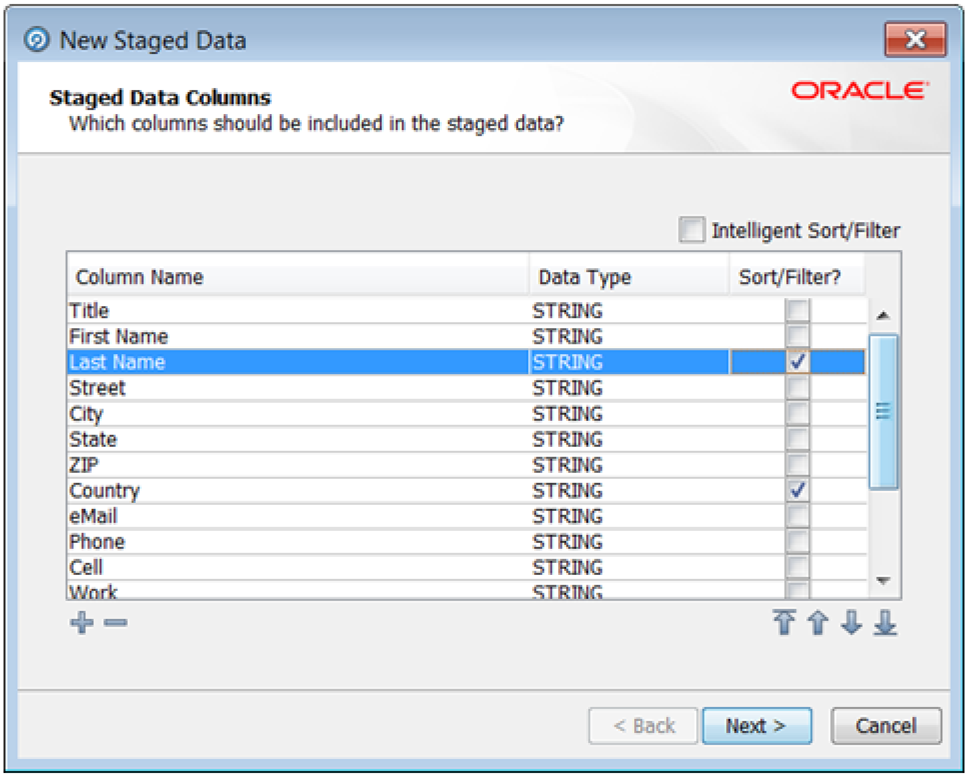
□ Use intelligent Sort/Filtering

Alternatively, if you know that sorting and filtering will only be needed on a sub-selection of the available columns, use the tick boxes to select the relevant columns. Note that any columns that are used as lookup columns by a Lookup and Return processor should be indexed to boost performance.Disabling sorting and filtering means that the total processing time of the snapshot will be less as the additional task to enable sorting and filtering will be skipped.Note that if a user attempts to sort or filter results based on a column that has not been enabled, the user will be presented with an option to enable it at that point.

Staged Data Sort/Filter options

When staged data is written by a process, the server does not enable sorting or filtering of the data by default. The default setting is therefore maximized for performance.

If you need to enable sorting or filtering on written staged data - for example, because the written staged data is being read by another process which requires interactive data drilldowns - you can enable this by editing the staged data definition, either to apply intelligent sort/filtering options (varying whether or not to enable sorting and filtering based on the size of the staged data table), or to enable it on selected columns (as below):



New Staged Data

Staged Data Columns

Which columns should be included in the staged data?

☐ Intelligent Sort/Filter

Column Name	Data Type	Sort/Filter?
Title	STRING	☐
First Name	STRING	☐
Last Name	STRING	☑
Street	STRING	☐
City	STRING	☐
State	STRING	☐
ZIP	STRING	☐
Country	STRING	☑
eMail	STRING	☐
Phone	STRING	☐
Cell	STRING	☐
Work	STRING	☐

Match Processor Sort/Filter options

It is possible to set sort/filter enablement options for the outputs of matching.

> **Note:**
>
> This should only be enabled if you wish to review the results of match processing using the Match Review UI.

5.7.4 Resource-Intensive Processors

The following processors are highly resource intensive because they need to write all of the data they process to the EDQ repository before they work on it:

- Quickstats Profiler

- Record Duplication Profiler
- Duplicate Check
- All match processors
- Group and Merge
- Phrase Profiler
- Merge Data Streams

> **Note:**
>
> This processor should only be used to merge records from separate readers; it is NOT necessary to use it to connect up multiple paths from the same reader.

The following processors work on a record-by-record basis, but are also highly resource intensive:

- Parse

  > **Note:**
  >
  > This Parse processor's performance is highly dependent upon its configuration, it can be fast or slow.

- Address Verification

Clearly, there are situations in which you will need to use one or more of these resource-intensive processors. For example, a de-duplication process requires a match processor. However, when optimal performance is required, you should avoid their use where possible. See below for specific guidance on how to tune the matching, Parse and Address Verification processors.

5.8 Performance Tuning for Parsing and Matching

In the case of Parsing and Matching, a large amount of work is performed by an individual processor, as each processor has many stages of processing. In these cases, options are available to optimize performance at the processor level.

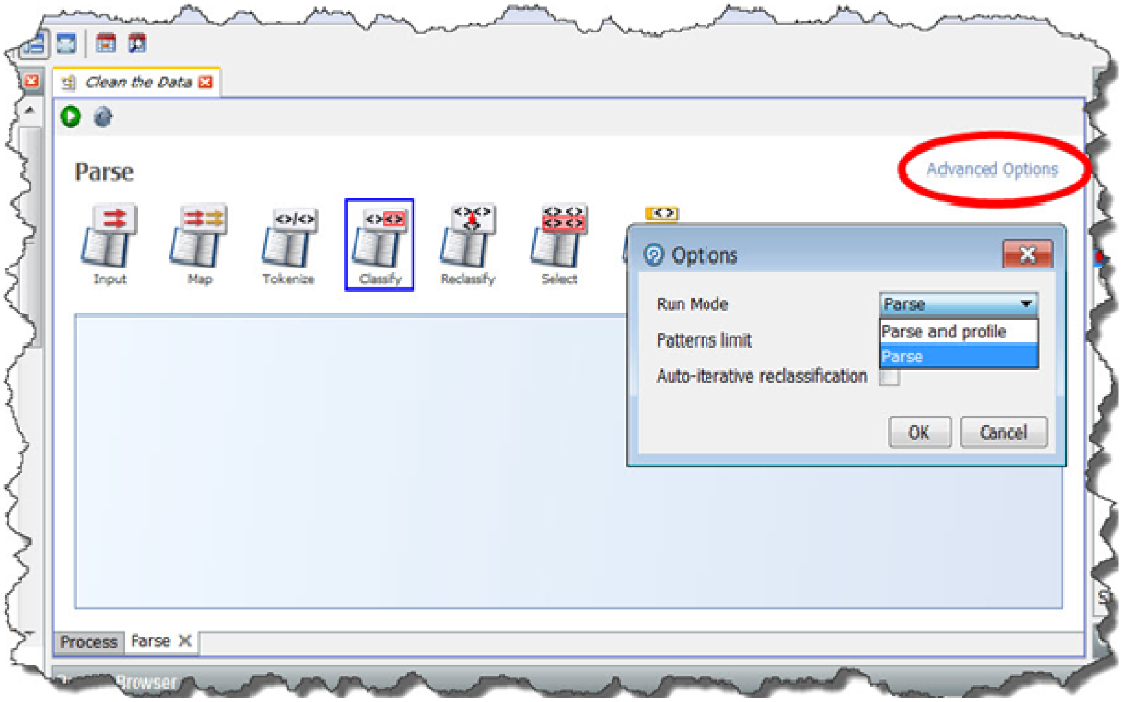
See below for more information on how to maximize performance when parsing or matching data:

5.8.1 Place Parse and Match processors in their own Processes

Both parsing and matching are inherently resource-intensive, and can take time to run. For this reason, it is advisable to place parse and match processors in processes on their own (or with only a small number of other processors). This will enable you to isolate and therefore accurately measure their performance, which should in turn make it easier to tune them.

5.8.2 Parsing performance options

By default, the Parse processor works in Parse and Profile mode. This is useful during configuration, as the parser will output the Token Checks and Unclassified Tokens results views. These will help you to define parsing rules. In production, however, when maximum performance is required from a Parse processor, it should be run in Parse mode, rather than Parse and Profile mode. To change the Parser's run mode, click its Advanced Options link, and then set the run mode in the Options dialog box.



Run Mode: Parse
Patterns limit
Auto-iterative reclassification □

For even better performance where only metrics and data output are required from a Parse processor, the process that includes the parser may be run with no drilldowns - see Minimized results writing above.

When designing a Parse configuration iteratively, where fast drilldowns are required, it is generally best to work with small volumes of data. If a parse processor has configuration that drives it to generate a number of different patterns for a given input record, for example it has many classification and reclassification rules, it may be possible to improve performance by reducing the number of patterns produced using the Patterns limit option (for example to 8) without altering results. If changing this option, parsing results should be tested for changes before and after making the change.

5.8.3 Matching performance options

The following techniques may be used to maximize matching performance:

5.8.3.1 Optimized Clustering

Matching performance may vary greatly depending on the configuration of the match processor, which in turn depends on the characteristics of the data involved in the matching process. The most important aspect of configuration to get right is the configuration of clustering in a match processor.

In general, there is a balance to be struck between ensuring that as many potential matches as possible are found and ensuring that redundant comparisons (between records that are not likely to match) are not performed. Finding the right balance may involve some trial and error - for example, assessment of the difference in match statistics when clusters are widened (perhaps by using fewer characters of an identifier in the cluster key) or narrowed (perhaps by using more characters of an identifier in a cluster key), or when a cluster is added or removed.

The following two general guidelines may be useful:

- If you are working with data with a large number of well-populated identifiers, such as customer data with address and other contact details such as e-mail addresses and phone numbers, you should aim for clusters with a maximum size of 20 for every million records, and counter sparseness in some identifiers by using multiple clusters rather than widening a single cluster.
- If you are working with data with a small number of identifiers, for example, where you can only match individuals or entities based on name and approximate location, wider clusters may be inevitable. In this case, you should aim to standardize, enhance and correct the input data in the identifiers you do have as much as possible so that your clusters can be tight using the data available. In this case, you should still aim for clusters with a maximum size of around 500 records if possible (bearing in mind that every record in the cluster will need to be compared with every other record in the cluster - so for a single cluster of 500 records, there will be 500 x 499 = 249500 comparisons performed).

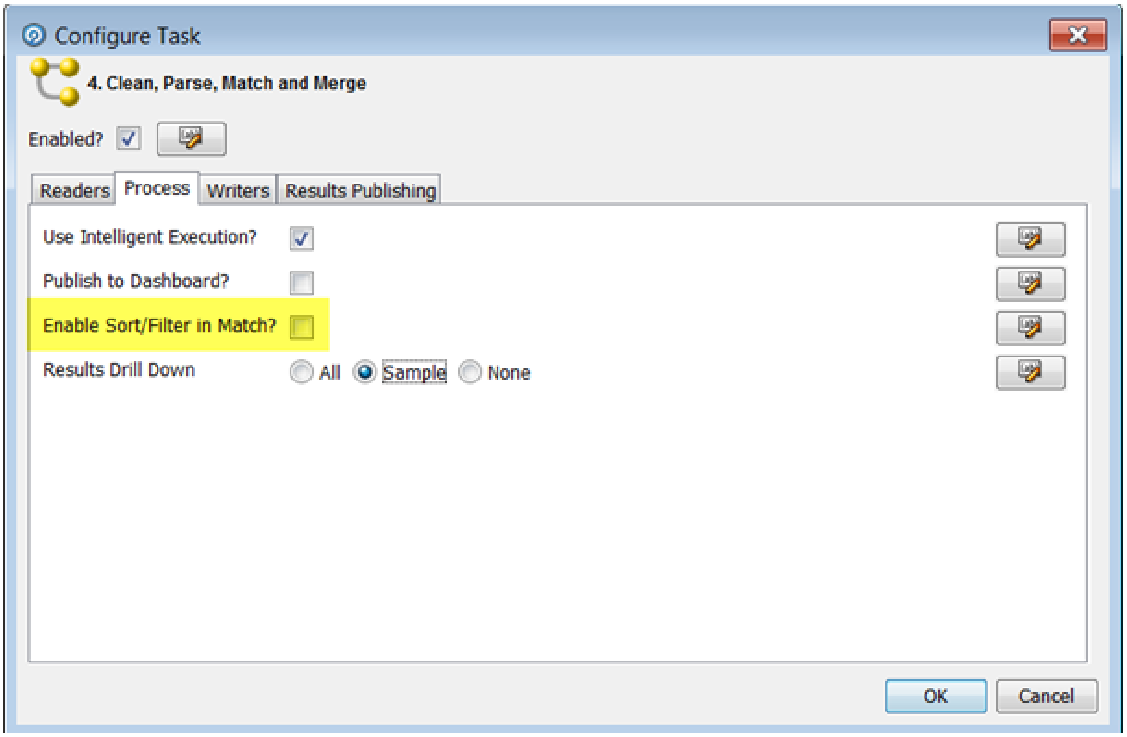
5.8.3.2 Disabling Sort/Filter options in Match processors

By default, sorting, filtering and searching are enabled on all match results to ensure that they are available for user review. However, with large data sets, the indexing process required to enable sorting, filtering and searching may be very time-consuming, and in some cases, may not be required.

If you do not require the ability to review the results of matching using the Match Review Application, and you do not need to be able to sort or filter the outputs of matching in the Results Browser, you should disable sorting and filtering to improve performance. For example, the results of matching may be written and reviewed externally, or matching may be fully automated when deployed in production.

The setting to enable or disable sorting and filtering is available both on the individual match processor level, available from the Advanced Options of the processor (see Sort/Filter options for match processors for details), and as a process or job level override.

To override the individual settings on all match processors in a process, and disable the sorting, filtering and review of match results, deselect the option to **Enable Sort/Filter in Match** processors in a job configuration, or process execution preferences:



Enabled? ☑

Use Intelligent Execution? ☑

Publish to Dashboard? □

Enable Sort/Filter in Match? □

Results Drill Down ○ All ● Sample ○ None

Note:

Sort / Filter in Match is disabled by default when processes are included in jobs.

5.8.3.3 Minimizing Output

Match processors may write out up to three types of output:

- Match (or Alert) Groups (records organized into sets of matching records, as determined by the match processor. If the match processor uses Match Review, it will produce Match Groups, whereas if uses Case Management, it will produce Alert Groups.)
- Relationships (links between matching records)
- Merged Output (a merged master record from each set of matching records)

By default, all available output types are written. (Merged Output cannot be written from a Link processor.)

However, not all the available outputs may be needed in your process. For example you should disable Merged Output if you only want to identify sets of matching records.

Note that disabling any of the outputs will not affect the ability of users to review the results of a match processor.

To disable Match (or Alert) Groups output:

- Open the match processor on the canvas and open the Match sub-processor.
- Select the Match (or Alert) Groups tab at the top.
- Un-check the option to Generate Match Groups report, or to Generate Alert Groups report.

Or, if you know you only want to output the groups of related or unrelated records, use the other tick boxes on the same part of the screen.

To disable Relationships output:

- Open the match processor on the canvas and open the Match sub-processor.
- Select the Relationships tab at the top.
- Un-check the option to Generate Relationships report.

  Or, if you know you only want to output some of the relationships (such as only Review relationships, or only relationships generated by certain rules), use the other tick boxes on the same part of the screen.

To disable Merged Output:

- Open the match processor on the canvas and open the Merge sub-processor.
- Un-check the option to Generate Merged Output.

  Or, if you know you only want to output the merged output records from related records, or only the unrelated records, use the other tick boxes on the same part of the screen.

5.8.3.4 Streaming Inputs

Batch matching processes require a copy of the data in the EDQ repository in order to compare records efficiently.

As data may be transformed between the Reader and the match processor in a process, and in order to preserve the capability to review match results if a snapshot used in a matching process is refreshed, match processors always generate their own snapshots of data (except from real time inputs) to work from. For large data sets, this can take some time.

Where you want to use the latest source data in a matching process, therefore, it may be advisable to stream the snapshot rather than running it first and then feeding the data into a match processor, which will generate its own internal snapshot (effectively copying the data twice). See Streaming a Snapshot above.

5.9 Performance Tuning for Address Verification

EDQ's Address Verification processor is a conduit to the Enterprise Data Quality Address Verification Server (EDQ AV). EDQ AV attempts to match each input record against all of the addresses that exist for that country in its Global Knowledge Repository. This operation is inherently resource-intensive, and it does take time to run. For this reason, it is advisable to place the Address Verification processor in a process on its own, or with only a small number of other processors. This will enable you to isolate and therefore accurately measure its performance, which should in turn make it easier to tune. The EDQ Address Verification server requires a substantial amount of memory outside of the EDQ Application Server's Java Heap. Address Verification's performance may suffer if insufficient memory is available. See Application Server Tuning for more information about tuning the EDQ Application Server's Java Heap. You can adjust Address Verification performance by tuning its caching options. You can control these parameters using the Address Verification Processor's Additional Options field, which is available from the processor's Options tab. Two parameters that it may be beneficial to adjust are:

- ReferenceDatasetCacheSize
- ReferencePageCacheSize

Full information on the available options is available on Loqate's support site: Setting options.

You should seek advice from Loqate Support before adjusting these parameters.

In addition to adjusting AV's caching parameters, Address Verification performance can be significantly improved by creating a different EDQ process for each country that you want to screen addresses from.

5.10 What Makes Processes Slow? Common Pitfalls

See below for more information.

5.10.1 Poor Matching Processor Configuration

EDQ Match processors are inherently very efficient, and feature many automatic optimizations. However, performance can be severely compromised by poor configuration. If a matching process takes a long time to run, this may be caused by its cluster configuration, as too many large clusters often result in too many comparisons, which will slow matching down.

5.10.2 Unnecessary Merge Data Streams Processors

A common misconception is that the Merge Data Stream processor is required to join up multiple paths from the same reader. It is not. Whilst the Merge Data Stream processor should be used to join up genuinely different data streams from different readers, any regular EDQ processor can join multiple paths from the same reader, and will simply work with all the distinct records from all of the joined paths.

5.10.3 Doing Too Much in a Single Process

It can be very difficult to identify the cause of a performance issue in a very large, complex process. Instead, you should create distinct modular processes for distinct operations, chaining them together with Data Interfaces. If you take this approach, you can easily see how long each process takes to run, which makes diagnoses much easier. An added benefit is that smaller processes are easier to understand and maintain.

5.10.4 Using the Script Processor when You Could Use a Core Processor

It is sometimes necessary to use the Script processor, but in nearly all cases this will result in slower performance than running a core processor, which uses compiled Java code. Don't use the script processor unless you really have to.

5.10.5 Using Matching Processors Unnecessarily

EDQ's audit and transformation processors work on a single record at a time. They can do all of their processing in memory, and can scale to as much CPU power as the application server has at its disposal. EDQ match processors, on the other hand, operate on sets of data. (This is also true of some profiling processors - see the list of Resource Intensive processors, above, for details.) In order to assess the similarity of the records in the data set, match processors write these data sets to the EDQ repository database. This I/O overhead is a requirement of match processors (and also of the Record Duplication Profiler, the Quick Stats Profiler, the Record Duplication Check processor and the Phrase profiler). There are a number of scenarios in which match processors are absolutely necessary. For example, you should use match processors:

- To identify fuzzy matches in large sets of data.

- To identify matches using multiple fields.
- Where you need to review possible matches.

However, if you simply need to return records in which a single field matches exactly, the Lookup and Return processor is likely to run more quickly than a match processor.

5.11 Tuning EDQ's Platform

Beyond designing efficient processes, EDQ itself does not require extensive tuning. There are only a few parameters you can usefully alter, and in most cases you can simply leave these set to their default values. (See the 'Oracle Fusion Middleware Administering Oracle Enterprise Data Quality' guide for more information). However, EDQ exists within an ecosystem. Aside from the physical hardware and network infrastructure, the most critical aspect of this ecosystem is the platform that EDQ runs on: specifically its application server and its database repository. Most performance issues are caused by sub-optimal process and job configuration, and it is not necessary to tune the platform to resolve them. However, in some cases, a few simple platform optimization steps can provide a performance boost. Before we discuss tuning the platform, let's just note that, when configuring a new EDQ installation, you should run a realistic load of test data through your system and observe the results before you finalize your settings.

5.11.1 The Application Server and the Database Repository

See below for more information.

5.11.1.1 Relative Importance of the Application Server and the Database Repository

Tuning the application server's maximum Java heap size can provide performance benefits. When tuning the maximum Java Heap size, please bear the following points in mind:

- The more processor cores (and therefore threads) available, the more memory you should allocate to the Java Heap. We recommend that, for optimal performance, you should allocate 2GB of memory for each runtime thread. (Note that EDQ will employ a runtime thread for every logical CPU that it detects).
- For most use cases, a setting of 8 GB is sufficient.

> **Note:**
>
> EDQ Customer Data Services Pack (CDS) has intensive memory requirements, and may require more than 8 GB.

- Note that allocating too much of your server's overall memory to the Java Heap can cause performance problems, as there may not be enough spare memory left to run applications that require non-Java Heap memory (an example is the EDQ Address Verification Server). Each thread also requires non-Java Heap memory, and so where EDQ has access to many threads, this will also increase the non-Java Heap memory requirement. As a rule of thumb, you should not allocate more than two thirds of your server's overall memory to the Java Heap.

For more information about tuning the application server, see the *Oracle Fusion Middleware Administering Oracle Enterprise Data Quality* guide.

5.11.1.2 Database Tuning

Whilst EDQ does require optimal database I/O to perform certain operations, such as matching, efficiently, how to tune your database depends on your specific use case and circumstances. It is, however, possible to offer some general advice. Database Administrators should:

- Ensure that they allocate sufficient Tablespace for EDQ. (You can find guidelines about Tablespaces sizes and other Database settings in the 'Oracle Fusion Middleware Installing and Configuring Oracle Enterprise Data Quality' guide.)
- Adopt an experiential approach to tuning the database's I/O performance.
- Set important settings such as PGA, SGA, Processes and Sessions as specified in Configuring Oracle Database to Support EDQ section in *EDQ Installation Guide*.

  It also may be worth noting that on WebLogic, the configured Data Sources control the maximum number of connections to the database. On large systems running many threads, it may be necessary to adjust the maximum connections to the Results database (where data is written and read by EDQ processes) from the default value of 200. A value of 500 is sufficient for nearly all use cases.
- Archive (redo) logging is resource expensive. In some cases, where all EDQ processing on a server is entirely stateless and the server can be re-provisioned automatically with no loss of service, it may be appropriate to turn off archive logging in the database for better performance. Or, if this is not possible, you may be able to mount redo log files on separate disks to improve performance.

5.11.2 Processor Cores and Process Threads

EDQ will create a runtime thread for each logical CPU (or 'core') that it detects. It will, where possible, divide processing amongst parallel threads. In general, process run times decrease as cores are added. However, after a certain number of cores have been made available to the Java Virtual Machine (JVM), the decrease in processing time for each extra core tends to become marginal, as the increased processing power is offset by greater contention. This is the case even when other aspects of your system, such as reading and writing to files and databases, have been optimized. In typical batch processing, improvements in run-times for each core added to the JVM became marginal when the total number of cores used by the JVM exceeded 16.

5.11.2.1 Process Threads

The number of process threads used to execute jobs is automatically set to the number of available cores, and should not usually be changed. However, when EDQ is installed on servers that have more than 16 cores and which are running heavy batch processing workloads, you may want to manually set the number of threads used by EDQ to 16. This is because a higher number of threads may lead to contention for resources when the threads have finished their work.

In order to set the number of threads manually, amend the following parameters in the `director.properties` file, which should be located in your EDQ instance's `oedq.local.home` folder:

- runtime.threads = 16
- runtime.indexingthreads = 16
- workunitexecutor.outputThreads = 16

When EDQ is installed on servers with more than 16 cores, you can scale by adding additional managed servers. Every managed server should be placed within the same WebLogic Cluster, but each will run within a separate Java Virtual Machine. For more information about how to add additional managed servers see the *High Availability* section of the *Understanding Oracle Enterprise Data Quality* guide. A single EDQ batch job will always run on a single managed server, so adding managed servers will not necessarily enable individual jobs to run more quickly. The advantage of having multiple managed servers is that it enables different jobs to run on different managed servers concurrently.

Note that the database server may be remote from the application server where EDQ will run, but it must be on a fast network connection.

6 Using JMX Extensions to Monitor EDQ

This chapter describes the Java Management Extensions (JMX) interface that can be used to monitor and manage many details of its operation. JMX is a Java technology designed for remote administration and monitoring of Java components
This chapter includes the following topics:

- Understanding JMX Binding
- Understanding JMX Bean Naming
- Monitoring Real-Time Processes

6.1 Understanding JMX Binding

EDQ can use either an internal JMX server or one that is provided in the WebLogic or Tomcat application server. This topic explains how to control which JMX server is used.

- A default installation of EDQ on Apache Tomcat uses an internal JMX server.
- A default installation of EDQ on Oracle WebLogic Server uses the JMX tree in the WebLogic Server application server.

The default configuration contains a Remote Method Invocation (RMI) registry, which is used by the EDQ command line interface as well as by JMX clients. The RMI listening port number is specified by the `management.port` property, defined in the `director.properties` file. The default is 8090. This property controls access to both the internal JMX Server and the RMI API that is used by the command line tools.

You can change the JMX configuration as follows:

- If you do not want to use the command line interface, and you want to have JMX Beans appear in the Tomcat application server JMX tree (not the internal JMX server), change the management.port property to 0:

  ```
  management.port=0
  ```

  When `management.port` is set to zero, the RMI registry does not listen on any port. This means that the internal JMX Server will not be used *and* that the RMI API will also not be available. The command line tools will therefore not work if `management.port` is set to 0.

- If you are using Oracle WebLogic Server, and you want to use the command line interface as well as have JMX Beans appear in the WebLogic Server JMX tree, add the following property to the `director.properties` file in the configuration directory. Retain the setting of 8090 for `management.port` so that the RMI API can be used by the command line tools.

  ```
  management.jndiname=java:comp/env/jmx/runtime
  ```

6.2 Understanding JMX Bean Naming

The naming scheme used for the JMX Beans is designed to work well with Jconsole. However, other JMX Clients may require a modified naming scheme.

The names used for the JMX Beans can be customized by writing and placing an appropriate JavaScript or Groovy file in the configuration directory and setting the `management.namemaker.scriptfile` property in the `director.properties` to indicate its existence

6.2.1 Reviewing the Example

This example demonstrates how to modify the default JMX Bean naming scheme to add a type attribute to the end of the name. The type attribute will be based on the Java Bean class.

1. Create a file named `jmxnames.js` in the configuration directory and add the following JavaScript to it:

   ```
   /**
   * Adds a type attribute to the name of a JMX Beans.
   *
   * @param beanclass The bean class name
   * @param domain The domain name
   * @param names The name strings
   *
   * @return The name string
   */
   function objectNameFor(beanclass, domain, names)
   {
   var type = beanclass == null ? "*" : beanclass.substring(beanclass.lastIndexOf('.')
   + 1);
   var out;
   /*
   * The names array always has 2 elements.
   */
   out = domain + ":" + "component=" + escape(names[0]) + ",name=" +
   escape(names[1]);
   for (var i = 2; i < names.length; i++)
   {
   var index = i-1
   out += "," + "name" + index + "=" + escape(names[i]);
   }
   return out + ",type=" + type;
   }
   ```

2. Add the following line to the `director.properties` file:

   ```
   management.namemaker.scriptfile = jmxnames.js
   ```

3. Restart the EDQ application server.

   The JMX Beans will now include a type qualifier at the end of their names.

6.3 Monitoring Real-Time Processes

is provided with a built-in JMX server that can be used to monitor many aspects of its operation. Many of the objects and resources that make up the EDQ application provide MBeans to the JMX server, including the real-time Web services.

6.3.1 Monitoring the Real-Time Web Service MBeans

Each real-time Web service registers an MBean for its reader and one for its writer in the JMX tree.

Readers are registered at:

`Runtime/Data/Buckets/Realtime/Projects/`*`Project Name`*`/readers/`*`Web service name`*

Writers are registered at:

`Runtime/Data/Buckets/Realtime/Projects/`*`Project Name`*`/writers/`*`Web service name`*

In each case, the path to the MBean includes the name of the Web service that owns it and the project that contains the web service.

Global Web services (those deployed in a `.jar` file in the `oedq_local_home/webservices` directory) have a different path name. Simply replace `Projects/`*`Project Name`* in the path above with `Global`.

The port for the internal JMX server is controlled by the `management.port` property, defined in the `director.properties` file.

6.3.2 Monitoring the Real-Time MBeans

A general JMX console, such as JConsole, can be used to interact with MBeans. Each MBean exposes:

- Attributes, whose values can be read.
- Operations that can be invoked to perform some action with the MBean.
- An interface that allows clients to subscribe to notifications of events that occur on the MBean.

The EDQ real-time web service MBeans uses the following attributes:

Attributes	Description
`closetime`	The time at which the bucket was last closed.
`concurrent`	The current number of synchronous requests.
`maxConcurrent`	The maximum number of concurrent synchronous requests since the bucket was opened.
`maxConcurrentMax`	The maximum number of concurrent synchronous requests since startup.
`messages`	The number of messages processed since the bucket was opened.
`open`	Indicates whether the bucket is open or closed.
`openCount`	The number of times the bucket has been opened since startup.
`opentime`	The time when the bucket was last opened.
`processtime`	The time when the last message was processed.
`records`	The number of records processed since the bucket was opened.
`threads`	The number of threads that used the bucket when it was last opened.
`totalMessages`	The number of messages processed since startup.
`totalRecords`	The number of records processed since startup.

The EDQ real-time web service MBeans exposes the following operation:

Attribute	Description
`closedown`	Shutdown the reader or writer using this bucket.

6.4 Using REST APIs to Access and Manage MBeans

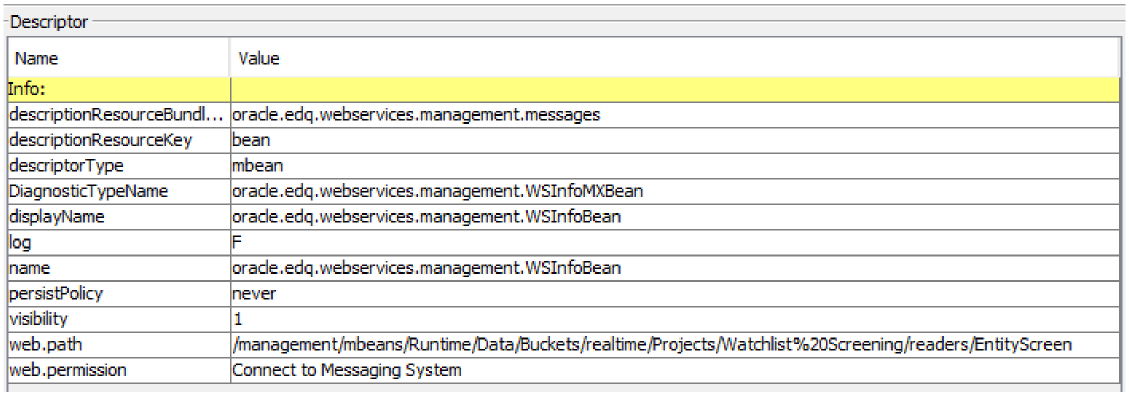
'EDQ defines a large number of JMX MBeans. A subset of these are commonly used for monitoring or changing key settings. This includes:

- "Bucket" MBeans that are used to monitor web service status.
- Logging MBeans that are used to control logging levels.
- SCCS MBean that has an operation to trigger rescan of source code control workspace, commonly used with Git integration.
- Runtime MBean that allows you to set runtime threads.

While you can use JConsole or scripts to access these MBeans, this is cumbersome and can cause problems with firewalls. EDQ 14.1.2.0.0 introduces a REST API that allows you to read and write MBean attributes and invoke MBean operations.

6.4.1 Web Enabled MBeans

An MBean that has a REST interface includes additional attributes in the MBean descriptor. These can be viewed in JConsole. Here's an example for a realtime reader MBean:



Descriptor

Name	Value
Info:	
descriptionResourceBundl...	oracle.edq.webservices.management.messages
descriptionResourceKey	bean
descriptorType	mbean
DiagnosticTypeName	oracle.edq.webservices.management.WSInfoMXBean
displayName	oracle.edq.webservices.management.WSInfoBean
log	F
name	oracle.edq.webservices.management.WSInfoBean
persistPolicy	never
visibility	1
web.path	/management/mbeans/Runtime/Data/Buckets/realtime/Projects/Watchlist%20Screening/readers/EntityScreen
web.permission	Connect to Messaging System

The **web.path** attribute is the path used after `http://server/edq` to access the MBean. The **web.permission** attribute displays the permission required to access the MBean.

The following MBeans are web-enabled in 14.1.2.0.0:

MBean	Description	Permission Required	Web path

Logger control	The logger control MBeans provide operations to query and set the logging level for each distinct logger registered in EDQ.	System Administration	The web path is:
			`/management/mbeans/Logging/`*`loggerpath`*
			For example,
			`/management/mbeans/Logging/OCI%20support`

MBean	Description	Permission Required	Web path
Runtime data nexus	The data nexus MBean provides operations to set and query runtime data "environment variables". The primary use case is in setting record counts in "randomator" buckets. For example, `<randomsource count="$randcount:10000" seed="$randseed">`	System Administration	The path to the MBean is `/management/mbeans/Runtime/Data/Nexus`
Bucket providers and consumers	These MBeans provide status information for realtime providers and consumers. Attributes include "open" and "active" states and record counts.	Connect to Messaging System and project access if relevant	The web paths are: - Global providers - `/management/mbeans/Runtime/Data/Buckets/realtime/Global/readers/providers/NAME` - Global consumers - `/management/mbeans/Runtime/Data/Buckets/realtime/Global/writers/consumers/NAME` - Project web service readers - `/management/mbeans/Runtime/Data/Buckets/realtime/Projects/PROJECTNAME/readers/WEBSERVICE` - Project web service writers - `/management/mbeans/Runtime/Data/Buckets/realtime/Projects/PROJECTNAME/writers/WEBSERVICE`
Runtime engine	The runtime engine MBean provides attributes to control the number of threads used in normal and interval mode. After update the values take effect the next time a job is started, but are not persisted over server start.	System Administration	The path to the MBean is `/management/mbeans/Runtime/Engine`
Source code control	The source code control MBean provides an operation to rescan the workspace location for changes. The primary use case is with Git integration after a pull operation. The web interface provides an alternative to the `scan.groovy` script that is shipped with EDQ.	System Administration	The path to the MBean is `/management/mbeans/SCCS/Controller`
File server control	This MBean can be used to configure web access to configuration and log files.	System Administration	See Using HTTP and HTPS Server to Access EDQ Files.

6.4.2 REST API for MBeans

The REST API for MBeans allows you to do the following:

- Read MBean attribute(s)
- Set MBean attribute
- Invoke MBean operation

Read MBean attribute(s)

```
GET http://server:port/edq/mbeanpath[?attributes=attr1,attr2]
```

The `mbeanpath` component is the MBean web path mentioned in Web Enabled MBeans. The optional `attributes` parameter lists the attributes to return. If omitted, all attributes are returned.

The result is a JSON object containing the attribute values.

For example, to get the **active** and **open** attributes for a the web service named **summer** in project **rt**, the URL would be: http://server:port/edq/management/mbeans/Runtime/Data/Buckets/realtime/Projects/rt/readers/summer?attributes=active,open

And the result might be:

```
{ "active": 0,
  "open": 0
}
```

Set MBean attribute

```
POST http://server:port/edq/mbeanpath/setattribute
```

The payload is a JSON object containing these attributes:

Attribute	Type	Description
attribute	String	The MBean attribute name.
value	Must match type of attribute	The new value for attribute.

The result is a JSON object containing the attribute **previousvalue** set to the previous value of the MBean attribute.

For example, to set the runtime thread count to 4, the URL would be:

```
http://server:port/edq/management/mbeans/Runtime/Engine
```

And the payload would be:

```
{ "attribute": "processThreads",
  "value": 4
}
```

Invoke MBean operation

```
POST http://server:port/edq/mbeanpath/invoke
```

The payload is a JSON object containing these attributes:

Attribute	Type	Description
operation	String	The MBean operation name.
arguments	Array	The invocation arguments.

The result is a JSON object containing the attribute **result** set to the result of the call.

For example, to set the log level for case management searches to FINEST, the URL would be:

```
http://server:port/edq/management/mbeans/Logging/
Case%20Management%20Filter%20Execution
```

And the payload would be:

```
{ "operation": "setLevel",
  "arguments": ["FINEST"]
}
```

This corresponds to adding the following to `logging.properties`.

```
com.datanomic.director.casemanagement.search.level = finest
```

Logging MBeans use the "display name" of the logger in the path, while `logging.properties` uses the internal package names. You can see the MBean package in the MBean **module** attribute by accessing the MBean JConsole on a running EDQ server edq | Logging | Case Management Filter Execution.

7

Using Triggers

This chapter describes how to use the trigger functionality in . This document describes where triggers are installed, how to call them, and how you can use them.
This chapter contains the following topics:

- Overview of the Triggers Functionality
- Required Skills to Use Triggers
- Storing Triggers
- Configuring Triggers Using the Script Trigger API
- Extending the Configuration of Triggers Using Properties Files
- Understanding EDQ Trigger Points
- Understanding TriggerInfo Methods
- Setting Trigger Levels
- Using JMS in Triggers
- Exposing Triggers in a Job Configuration
- Trigger Examples

7.1 Overview of the Triggers Functionality

Triggers in are scripts (JavaScript or Groovy) that can be called at various *trigger points* in the EDQ system. There are two types of triggers: predefined triggers and custom triggers.

7.1.1 About Predefined Triggers

Predefined triggers are included with the EDQ installation. They are visible in the Director user interface and can be used in a job configuration to start the job, shut down web services, send email notifications, and run another job from within a job. Director users can set these triggers to run at the following trigger points: the start of a job, the end of a job, or both. You can learn more about predefined triggers in the Director online help system.

7.1.2 About Custom Triggers

Custom triggers can be written by someone skilled in Javascript or Groovy to extend the functionality of EDQ to achieve specific workflow objectives. You can use custom triggers to perform tasks such as:

- sending an email message
- sending a JMS message
- calling a web service
- writing a file
- sending a text message

You can run custom triggers at any of the following predefined trigger points:

- Before running a job phase
- After running a job phase
- On making a match decision
- On making a transition in Case Management
- When a job completes

Each of these trigger point has a unique path and a set of defined arguments that are passed to the trigger through a special API. For more information, see Understanding EDQ Trigger Points.

Custom triggers are described in the rest of this document.

7.2 Required Skills to Use Triggers

Knowledge of Javascript or Groovy is required to create and deploy custom triggers in EDQ.

7.3 Storing Triggers

Custom triggers must be stored in the `triggers` subdirectory of the EDQ `config` (configuration) directory. New or updated triggers are loaded automatically without requiring a system restart.

7.4 Configuring Triggers Using the Script Trigger API

You can use the functions of the script API to create your triggers. These functions are defined in the trigger code. Although the examples in this document are JavaScript, the same API is available in Groovy.

The following are descriptions of each function in this API.

getPath()
Returns a string that defines the path that the trigger will handle. Each trigger point has a unique path. Any trigger that matches a given path is executed when the trigger point is reached. For more information about trigger points, see Understanding EDQ Trigger Points. This function is a regular expression. For example, the path `/log/com\.datanomic\..*` would match any logging path where the logger name contains the string `datanomic` (in other words, any logger defined in EDQ, the word "datanomic" being another name for EDQ).

run(*path*, *id*, *env*, *arg1*, *arg2* ...)
Executes the trigger. For more information about what is returned by the trigger API for each of these variables, see Understanding EDQ Trigger Points.

> ***path***
> The path of the trigger, for example `/runtime/engine/interval/end`.
>
> ***id***
> The trigger ID. The ID is set when the trigger is configured in the Director user interface. The ID is null for simple triggers.

env
The trigger environment in the form of one or more key/value pairs, for example `env.project = project name`. The *`env`* input is specific to the trigger point. These values are exposed as properties of the `env` object in the script. Most trigger points will pass in the associated EDQ project ID and project name.

arg
Extra arguments that are specific to the trigger point. For example, the `Interval end` trigger point returns the following: Task context object, process options, interval number (>= 1), execution statistics.

filter(*path*, *env*)
(Optional function) Filters out the trigger before it can be executed. Use this filter to avoid the overhead of executing a trigger that will not be needed. Return `true` to enable the trigger or `false` to disable it.

path
The path of the trigger.

env
The trigger environment in the form of one or more key/value pairs. The *`env`* input is specific to the trigger point. These values are exposed as properties of the `env` object in the script. Most trigger points will pass in the associated EDQ project ID and project name. In the following example, the trigger is enabled only if the associated project is named "My project."

```
function filter(path, env) {
  return env.project == 'My project';
}
```

getLevel()
(Optional function) Returns the maximum level the trigger will accept. For example, the following statement allows the trigger to accept all levels, regardless of other settings in the trigger system. For more information about setting levels, see Setting Trigger Levels.

```
function getLevel() {
  return Level.SEVERE;
}
```

getTriggerNames(*path*, *env*)
(Optional function) Returns an array of `TriggerName` objects for display in the Director user interface. For more information, see Exposing Triggers in a Job Configuration. Getting trigger names and exposing them in the Director interface is only possible with the job configuration screen.

7.5 Extending the Configuration of Triggers Using Properties Files

You can specify additional configuration for script triggers in properties files. Access to these properties is by means of a predefined object named `config`, which is available in all triggers. The base directory in EDQ for these properties files is the subdirectory `config` within the `triggers` directory. The following are useful methods for the `config` object.

config.getTriggerConfigFiles(*base*, *pattern*)
Returns an array of file objects whose names match a search pattern within a specified directory in the `triggers/config` directory.

base
The name of a directory within the `triggers/config` directory.

pattern
A regular expression (regex) that defines the search pattern to match.

config.loadProps(*file*)
Loads a specified Java properties file and return it as a JavaScript object.

file
The name of the Java properties file.

7.6 Understanding EDQ Trigger Points

This section describes the trigger points within EDQ at which you can call custom triggers.

Log Message
Called whenever a log message is generated in the system.

Component	Description
`Path`	`/log/`*`loggername`*
`Env`	`null`
`Arguments`	`java.util.logging.LogRecord`

Syslog Message
Called whenever a high-level `syslog` log message is generated. The *`source`* argument is a Java object that contains details of the event source. It can be converted to string for display.

Component	Description
`Path`	`/syslog`

`Env`	`env.event` = *`event_name`*
	`env.source` = *`event_source_as_string`*

`Arguments`	*`event_name`*, *`source`*, *`message`*

Process start
Called when a process starts. The arguments are Java objects that contain information on the process configuration.

Component	Description
`Path`	/`runtime/engine/task/start`

`Env`	`env.project` = *`project_name`*
	`env.projectID` = *`project_ID`*
	`env.missionname` = *`job_name`*
	`env.processname` = *`process_name`*

`Arguments`	*`Task_context_object`*, *`process_options`*

> **Note:**
>
> When specifying the path for starting a task, the trigger script must include `addLibrary('runtime')` to avoid the trigger script from throwing an error.

Process end
Called when a process stops. The arguments are Java objects that contain information on the process configuration.

Component	Description
`Path`	`/runtime/engine/task/end`

`Env`	`env.project` = *`project_name`*
	`env.projectID` = *`project_ID`*
	`env.missionname` = *`job_name`*
	`env.processname` = *`process_name`*

`Arguments`	*`Task_context_object`*, *`process_options`*

> **Note:**
>
> When specifying the path for ending a task, the trigger script must include `addLibrary('runtime')` to avoid the trigger script from throwing an error.

Interval end
Called at the end of a normal process or at the end of each interval of a process that is run in interval mode. Returns statistics on the number of records executed, etc.

Component	Description
`Path`	`/runtime/engine/interval/end`

`Env`	`env.project` = *`project_name`*
	`env.projectID` = *`project_ID`*
	`env.missionname` = *`job_name`*
	`env.processname` = *`process_name`*

`Arguments`	*`Task_context_object`*, *`process_options`*, *`interval_number (>= 1)`*, *`execution_statistics`*

Before job phase
Called in a job configuration for 'pre phase' execution.

Component	Description
`Path`	`/missions/phase/pre`

`Env`	`env.project` = *`project_name`*
	`env.projectID` = *`project_ID`*
	`env.missionname` = *`job_name`*
	`env.processname` = *`process_name`*

`Arguments`	None

After job phase

Called in a job configuration for 'post phase' execution.

Component	Description
`Path`	`/missions/phase/post`

`Env`	`env.project` = *`project_name`*
	`env.projectID` = *`project_ID`*
	`env.missionname` = *`job_name`*
	`env.processname` = *`process_name`*

`Arguments`	None

On match decision

Called when EDQ must make a decision about a potential match. This is known as a *relationship decision* trigger. Relationship triggers can include methods that return the relationship and decision data needed to perform matching. This trigger point is specific to Match Review.

Component	Description
`Path`	`/matchreview/relationship/decision/`
`Env`	`env.project` = *`project_name`*
`Arguments`	A list of `TriggerInfo` methods. Each contains data for one relationship. See Understanding TriggerInfo Methods for descriptions of these methods.

When a case is created

Called when a case is created where the case belongs to the respective case source.

Component	Description
`Path`	`/casemanagement/create/<case source name>`

`Env`	`env.sourceName` = `case source name`
	`env.caseType = the type of the case or alert`
	`env.currentState = the current state of the created case`

`Arguments`	`com.datanomic.director.casemanagement.beans.CaseBean`

When a case or alert has transitioned

Called after a case or alert has transitioned into the next logical state corresponding to the respective workflow.

Component	Description
`Path`	`/casemanagement/transition/<workflow name>/<transition>`

Component	Description

`Env`	`env.sourceName = case source name`
	`env.caseType = the type of the case, 'case' or 'alert'`
	`env.currentState = the current state of the created case`

`Arguments`	`com.datanomic.director.casemanagement.beans.CaseBean,java.util.List<com.datanomic.director.casemanagement.beans.CaseHistoryBean>,comment,restrictingPermission` Where `comment` is entered by the user and `restrictingPermission` is the permission required to access the comment.

When a case or alert has been updated

Called when a case or alert is updated by a user. This includes assignment, state change, priority change, or performing any other edit.

Component	Description
`Path`	`/casemanagement/update/<case source name>`

`Env`	`env.sourceName = case source name`
	`env.caseType = the type of the case, 'case' or 'alert'`
	`env.currentState = the current state of the created case`

`Arguments`	`com.datanomic.director.casemanagement.beans.CaseBean,java.util.List<com.datanomic.director.casemanagement.beans.CaseHistoryBean>,comment,restrictingPermission` Where `comment` is entered by the user and `restrictingPermission` is the permission required to access the comment

When a comment is added for a case or alert

Called when a comment is added for a case or alert.

Component	Description
`Path`	`/casemanagement/commented/<case source name>`

`env.sourceName = case source name`
`Env`
	`env.caseType = the type of the case, 'case' or 'alert'`
	`env.currentState = the current state of the created case`

Component	Description
`Arguments`	`com.datanomic.director.casemanagement.beans.CaseBean,java.util.List<com.datanomic.director.casemanagement.beans.CaseHistoryBean>,comment,restrictingPermission` Where `comment` is the user entered comment and `restrictingPermission` is the permission required to access the comment.

After a system update occurs
Called after a case or alert is updated as part of an escalation or bulk update.

Component	Description
`Path`	`/casemanagement/systemupdate/<case source name>`

`Env`	`env.sourceName = case source name`
	`env.caseType = the type of the case, 'case' or 'alert'`
	`env.currentState = the current state of the created case`

`Arguments`	`com.datanomic.director.casemanagement.beans.CaseBean,java.util.List<com.datanomic.director.casemanagement.beans.CaseHistoryBean>,comment,restrictingPermission` Where `comment` is the user entered comment and `restrictingPermission` is the permission required to access the comment.

7.7 Understanding TriggerInfo Methods

This section explains each of the methods that are associated with the `TriggerInfo` trigger point. These methods are specific to the `TriggerInfo` trigger point for use in Match Review.

Table 7-1 Methods Associated with the TriggerInfo Trigger Point

Method	Data Returned	Description
`getPreviousMatchStatus()`	String	Returns the match status prior to the decision.
`getPreviousRealtionshipReviewStatus()`	String	Returns the relationship review status prior to the decision.
`getRelationshipId()`	Integer	Returns the relationship ID.
`getRecordId()`	Integer	Returns the ID of the first record.
`getInputId()`	Integer	Returns the ID of the first input.
`getRelatedRecordId()`	Integer	Returns the ID of the second record.

Table 7-1 (Cont.) Methods Associated with the TriggerInfo Trigger Point

Method	Data Returned	Description
`getRelatedInputId()`	Integer	Returns the ID of the second input.
`getReviewStatus()`	String	Returns the review status of the new relationship.
`getMatchStatus()`	String	Returns the new match status.
`getRuleName()`	String	Returns the name of the rule that generated the relationship.
`getCommentUser()`	String	Returns the user name of the person that made the comment.
`getReviewComment()`	String	Returns any comment that was made.
`getCommentDate()`	Date	Returns the date and time that the comment was made (if comment is present).
`getReviewedUser()`	String	Returns the name of the user who performed the review.
`getReviewDate()`	Date	Returns the date and time that the review was performed.
SourceAttribute `getRecordSourceAttributes()`	List	Returns all the source attributes (columns) that make up the first record.
SourceAttribute `getRelatedRecordSourceAttributes()`	List	Returns all the source attributes (columns) that make up the second record.
`getRecordAttributeValue(`*SourceAttribute* `sa)`	Value	Returns the value of the given source attribute (column) of the first record.
`getRelatedRecordAttributeValue(`*SourceAttribute* `sa)`	Value	Returns the value of the given source attribute (column) of the second record.

7.8 Setting Trigger Levels

Every trigger point has an associated level, which is a `java.util.logging.Level` value. By default trigger calls with a level lower than `INFO` are ignored.

One way to modify the level is to create a file named `levels.properties` in the `triggers` subdirectory of the `config` directory. This file can contain both a default level and one or more override levels for individual paths. Example 7-1 sets the default level to `FINE` and sets the level for the path `/runtime/engine/.*` to `FINER`. You can define your own prefix for the pattern and level properties.

Another way to modify the level is to define a `getLevel` function in the trigger. See Configuring Triggers Using the Script Trigger API for a description.

Example 7-1 Setting Trigger Levels

```
default = fine
runtime.pattern = /runtime/engine/.*
runtime.level   = finer
```

7.9 Using JMS in Triggers

To enable Java Message Service (JMS) within a trigger file, follow these steps.

1. Load the internal JavaScript JMS library.

   ```
   addLibrary("jms");
   ```

2. Load properties that define the JMS configuration. These properties are augmented with the JMS settings from the standard `realtime.properties` file that is shipped in the EDQ configuration directory. The default version of this file defines properties for the open-source ActiveMQ message broker that is bundled with EDQ. At minimum, the trigger should supply a value for the `destination` property, which names the JMS topic or queue to use.
3. Create a JMS object.

   ```
   var jms = JMS.open(props);
   ```

4. Send a text message.

   ```
   jms.send(str)
   ```

5. Send a JMS map message built from a script object.

   ```
   jms.sendMap(jsobj)
   ```

6. Create a text message. Properties and header values can be set on the message before transmission.

   ```
   var msg = jms.createTextMessage(str)
   ```

7. Create a map message. Properties and header values can be set on the message before transmission.

   ```
   var msg = jms.createMapMessage(jsobj)
   ```

8. Send a message that was created by one of the two preceding methods.

   ```
   jms.sendMessage(msg)
   ```

7.10 Exposing Triggers in a Job Configuration

Triggers are selected for use in a job when configuring a job phase in Director. They can be set to run before or after a job phase. To make triggers available for selection on the configuration screen, each trigger must be able to return a list of names. This allows one trigger to perform multiple tasks as needed.

A trigger name has the following components:

- an internal ID that is passed to the trigger **run** function. See Configuring Triggers Using the Script Trigger API for a description of this function.
- a visible label
- a group name

Trigger names with the same group are shown as a single node in the job configuration screen.

To create a new trigger name:

```
var n1 = new TriggerName(id, label)
n1.group = "My group";
```

To return trigger names from a trigger:

To return trigger names, use the `getTriggerNames` function as shown in this example.

```
function getTriggerNames(path, env) {
  var n1 = new TriggerName(id1, label1);
  var n2 = new TriggerName(id2, label2);
  ...
  n1.group = "My group";
  n2.group = "My group";
  ...
  return [n1, n2 ...]
}
```

See Configuring Triggers Using the Script Trigger API for more information about `getTriggerNames`.

7.11 Trigger Examples

The following are examples of how you can use custom triggers.

> **Note:**
>
> The examples in this document are JavaScript, but the same API is available in Groovy.

Example 1 Use a Trigger to Send Log Messages Via JMS
In this example, the logging library imports a logging object that can be used to format and output the message. The JMS properties file is loaded from `triggers/config/jms/jms.properties` in the EDQ configuration directory.

```
// Test trigger for task running with JMS
addLibrary("logging");
addLibrary("jms");

function getPath() {
  return "/log/com\.datanomic\..*";
}
function run(path, id, env, logrecord) {

  var pfiles = config.getTriggerConfigFiles("jms",
                             "jms\\.properties");

  if (pfiles.length > 0) {
    var props = config.loadProps(pfiles[0]);

    var jms = JMS.open(props);
    var msg = logging.format(logrecord);
    var len = msg.length;

// Remove trailing newlines

    while (len > 0) {
      var c = msg.charAt(len - 1);
      if (c != '\n' && c != '\r') {
        break;
      }

      len--;
    }

    jms.send(msg.substring(0, len));
    jms.close();
  }
}
```

Example 2 Use a Trigger to Send Syslog Messages Via JMS

In this example, the special `id` directive on the first line (`#! id : syslog`) defines the internal ID of the trigger. If there is more than one trigger definition with the same ID, the later one replaces the former one. In a standard EDQ install, there is a predefined `syslog` trigger that logs messages through the standard logging API. Adding the `id` directive in this example causes the JMS `syslog` trigger to replace the predefined trigger.

```
#! id : syslog

// Test trigger for task running with JMS

addLibrary("logging");
addLibrary("jms");

function getPath() {
  return "/syslog";
}

function getLevel() {
  return Level.SEVERE;
}

function run(path, id, env, level, event, source, message) {

  var pfiles = config.getTriggerConfigFiles("jms",
                "jms\\.properties");
  var props  = null;

  if (pfiles.length == 0) {
    logger.log(Level.WARNING, "syslogger called but no properties");
  } else {

    props = config.loadProps(pfiles[0]);

    var jms    = JMS.open(props);
    var xml    = <syslog level={level}><source>{source}</source><message>{message}</
message></syslog>

    logger.log(Level.INFO, "xml = {0}", xml.toXMLString());
    jms.send(xml.toXMLString());
    jms.close();
  }
}
```

Example 3 Use a Trigger for Mission Phase Notification

In this example, a couple of trigger names are defined and are exposed to the job configuration screen. The trigger writes a log message in this example, but it could also be configured to send JMS notifications.

```
// Test trigger for misssion phase notification

addLibrary("logging");

function getPath() {
  return "/missions/phase/.*";
}

function run(path, id, env) {
 logger.log(Level.INFO, "phase called with path {0} and id {1}", path, id);
}

function getTriggerNames(path, env) {
  var n1 = new TriggerName("logme", "logme2");
  n1.group = "logmegroup";

  var n2 = new TriggerName("n2", "n2");
  n2.group = "logmegroup";
  return [n1, n2];
}
```

8
Using Case Management Scripting

EDQ 12.2.1.4.4 introduces a case management script library that you can use to interact with cases and alerts. This chapter describes how to use the case management script library.
This chapter contains the following topics:

- Overview of the Case Management Script Library
- casemanager object properties
- case bean properties
- case history bean properties
- Source Data Object properties
- Case Management Trigger Environment

8.1 Overview of the Case Management Script Library

You can use the case management script library in EDQ triggers and script processors, to access Case Management data and a number of functions in a supported manner.

To use the library, add this line to the top of the script:

```
addLibrary("casemanagement")
```

The library publishes an object named `casemanager` in the script. See casemanager object properties for more information.

> **Note:**
>
> The case management script library has an implicit dependency on the user cache script library so that the `usercache` object and functions are available. See Using Scripting for User Cache Queries for more information.

8.1.1 casemanager object properties

The `casemanager` object exposes the following constant values:

Name	Description	Value
CaseManagement.CASE_TYPE	The internal type for cases.	case
CaseManagement.ALERT_TYPE	The internal type for alerts.	issue
CaseManagement.PRIO_NONE	Integer value corresponding to the **None** priority setting.	0

Name	Description	Value
CaseManagement.PRIO_LOW	Integer value corresponding to the **Low** priority setting.	1
CaseManagement.PRIO_MEDIUM	Integer value corresponding to the **Medium** priority setting.	2
CaseManagement.PRIO_HIGH	Integer value corresponding to the **High** priority setting.	3
CaseManagement,UNASSIGNED	Use this object to "unassign" a case or alert.	*Unassigned user object*

The `casemanager` object exposes the following functions:

Name	Description
bean = casemanager.loadCaseById(id)	Load case or alert by internal ID.
bean = casemanager.loadCaseByExternalId(extid)	Load case or alert by external ID. The external ID is shown in the UI.
bean = casemanager.getChildCases(id)	Return array of alert beans corresponding to the case with the given internal ID.
bean = casemanager.getRelatedCases(bean)	If the bean is a case, return array containing it along with the child alerts. Else return array containing the parent case bean and its child alerts.
bean = casemanager.updateAssignedUser(casebean, user, [dotrigger])	Update assigned user for case/alert. The user object is obtained from the user cache library methods. Use `null` or `CaseManager.UNASSIGNED` to "unassign" the case or alert. Execute update triggers if `dotrigger` is omitted or is `true`. Return the updated case bean.
bean = casemanager.updateCase(bean, [dotrigger])	Save changes to the case or alert. Execute update triggers if `dotrigger` is omitted or is `true`. Return the updated bean.
bean = casemanager.updateState(bean, transition, comment, permission, [dotrigger])	Update the current state of the case or alert. `transition` is the transition name, `comment` is the comment to associate with the change and `permission` is the restricting permission for the comment. Execute update triggers if `dotrigger` is omitted or is `true`. Return the updated bean.
bean = casemanager.addComment(bean, comment, restrictingpermission, [dotrigger])	Add a comment to the case or alert. Execute update triggers if `dotrigger` is omitted or is `true`. Return the updated bean.
extstate = casemanager.getExternalState(source, casetype, state)	Get the "external" state (such as AWAITING_REVIEW) corresponding to the given source, case type, and case state.
sd = getSD(id)	Load source data for the alert with the given ID. The structure of the result is described in Source Data Object properties.
bool = CaseManagement.isLockError(e)	Check whether an error indicates that a case or alert had been modified elsewhere between loading and saving. You can use this to support retries in a script.

The argument to the `isLockError` function is an error from a script catch clause.

Example:

```
var cb = casemanager.loadCaseById(caseid)

cb.priority = CaseManagement.PRIO_HIGH

try {
  cb.save();
} catch (e) {
  if (CaseManagement.isLockError(e)) {
    ... retry logic  ...
  }
}
```

8.1.2 case bean properties

The case bean passed to trigger calls and used in `casemanager` function has the following properties:

Property	Type	Is Writable	Description
id	Number	No	Internal ID of the case/alert.
parentId	Number	No	For alerts, internal ID of the parent case.
version	Number	No	Internal version number of the case/alert.
caseGroup	String	No	Case "group" (such as "match").
caseType	String	No	Type: "case" for cases and "issue" for alerts.
externalId	String	No	The external ID of the case/alert, as seen in the user interface.
sourceName	String	No	Case source name.
sourceId	String	No	Internal source "id".
caseKey	String	No	Case/alert key.
keyLabel	String	No	Display version of case key.
flagKey	String	No	Flag key.
description	String	Yes	Description string.
currentState	String	No	Current state of case/alert.
externalState	String	No	The 'external' state of the case/alert, such as AWAITING_REVIEW.
derivedState	String	No	Derived state.
createdBy	Number	No	ID of user who created case/alert.
createdDateTime	Date	No	Creation timestamp.

Property	Type	Is Writable	Description
modifiedBy	Number	No	ID of user who last modified case/alert.
modifiedDateTime	Date	No	Timestamp of last modification.
assignedUser	Number	No	ID of currently assigned user.
assignedBy	Number	No	ID of user who performed last assignment.
assignedDateTime	Date	No	Timestamp of last assignment.
priority	Number	Yes	Priority number: 0 = None, 1= Low, 2 = Medium, 3 = High
permission	String	Yes	Case/alert permission string.
stateExpiry	Date	Yes	Timestamp of state expiry. Updates are ignored if current state is not enabled for expiry.
stateChangeBy	Number	No	ID of user who last changed state.
stateChangeDateTime	Date	No	Timestamp of last state change.
reviewFlag	Boolean	Yes	Review flag.
updatedBy	Number	No	ID of user who last updated review flag.
updatedDateTime	Date	No	Timestamp of last review flag update.
groupId	String	No	Internal group ID.
groupLevel	Number	No	Internal group level.
extendedAttributeN	String/number/boolean	Yes	The value of extended attribute (flag) N.
extendedAttributeNModifiedBy	Number	No	ID of user who last changed extended attribute (flag) N.
extendedAttributeNModifiedDateTime	Date	No	Timestamp of last modification of extended attribute (flag) N.

The case bean passed to trigger calls and used in `casemanager` function has the following functions:

Name	Description
value = casebean.getExtendedAttribute(n, [deflt])	Get the value of extended attribute (flag) N. If `deflt` is omitted or `true`, return the default value if the attribute has not been set. This function is an alternative to the `extendedAttributeN` and allows extended attribute value to be returned using a computed index.
casebean.setExtendedAttribute(n, v)	Set the value of extended attribute (flag) N.
userid = casebean.getExtendedAttributeModifiedBy(n)	Get the ID of user who last changed extended attribute (flag) N.
date = casebean.getExtendedAttributeModifiedDateTime(n)	Get the Timestamp of last modification of extended attribute (flag) N.
sd = casebean.getSD()	Load source data for the alert. The structure of the result is described in Source Data Object properties.
casebean.save([dotrigger])	Save the case or alert after modifications. Fire update triggers if `dotrigger` is omitted or is `true`.

8.1.3 case history bean properties

The case history bean passed to trigger calls has the following read only properties:

Property	Description
id	Internal ID of case history record.
caseId	Internal ID of associated case or alert.
sourceName	Name of source.
modifiedBy	ID of user making change.
modifiedDateTime	Timestamp of change.
attribute	Attribute name.
oldValue	Internal form of previous value.
newValue	Internal form of new value.
oldValue2	Display form of previous value.
newValue2	Display form of previous value

For string values, the internal and display values are the same. For the priority attribute the internal value is numeric and the display form is the corresponding priority name. For attributes associated with users, the internal form is the user ID and the display form is the user display name.

8.1.4 Source Data Object properties

The source data object returned by the `getSD` functions has the following read only properties:

Property	Description
datasets	Array of data sets (sources).
storageTime	Timestamp of data storage.

Source datasets are identified by an internal ID (a numeric string or "reln" for relationship data) and by a label as seen in the user interface. The source data object has the following functions to extract datasets and values:

Function	Description
dataset = dataset(id)	Get dataset by internal ID.
dataset = datasetFromLabel(label)	Get dataset by UI label.
values = values(dsid, itemid)	Get values for dataset with internal ID `dsid` and item with internal ID `itemid`. The result is an array with one value for each source record associated with the alert.

The functions return null if an unknown ID or label is used.

Each dataset object has the following read only properties:

Property	Description
id	The internal ID of the data set.
label	The UI label for the data set.
items	Array of data items.

Data items within a dataset are also identified by an internal ID (column name or relationship attribute) and a label as seen in the UI.

The dataset object has the following functions to extract data items and values:

Function	Description
dataitem = item(id)	Get data item by internal ID.
dataitem = itemFromLabel(label)	Get data item by UI label.
values = values(itemid)	Get values for the item with internal ID `itemid`. The result is an array with one value for each source record associated with the alert.

The functions return null if an unknown ID or label is used.

Each data item has the following read only properties:

Property	Description
id	The internal ID of the data item.
label	The UI label of the data item.
values	Array of item values, one for each source record associated with the alert.

The item values are returned as strings, numbers, or dates depending on the source data column type.

8.2 Case Management Trigger Environment

The third argument to all trigger function calls is the trigger "environment". For example, case management update triggers are defined with:

```
function getPath() {
  return "/casemanagement/update/.*";
}

function run(path, id, env, casebean, history, comment) {
  ...
}
```

The environment object for case management triggers contains the following read only properties:

Property	Description
sourceName	The source name.
caseType	The case type ("case" or "issue").
currentState	The current state of the case or alert.
origin	The origin of the update.
userid	The ID of the user making the update.

The origin property identifies the root cause of the update. It can have one of the following values:

Property	Description
user	Change made by user in case management UI.
api	REST API call.
bulk	Bulk update.
expiry	Change caused by automatic state expiry.
creator	Case creation.
import	Case import.
script	Change originated in call made by this script library.

9 Using Scripting for User Cache Queries

Job and case management triggers often need to convert user IDs to user names and extract user attributes such as emails. EDQ 12.2.1.4.4 provides a script library with functions for user cache and user query. This chapter describes how to use the script library.

To use the library, add this line to the top of the script:

```
addLibrary("usercache")
```

The library publishes an object named **usercache** in the script, which includes the following functions:

Name	Description
user = usercache.getUser(id)	Map a numeric ID to a user object. Return `null` if no user is found. The user may be retrieved from the record of deleted users.
user = usercache.getLiveUser(id)	Map a numeric ID to an existing user. Users recorded as deleted are not returned.
user = usercache.lookup(username, [withid])	Lookup a user by username by searching all configured realms. If `withid` is omitted or is `true`, an internal ID is allocated for the user if this has not been done before. Internal IDs are required for certain APIs such as case assignment.
user = usercache.lookupInRealm(username, realm, [withid])	Lookup a user by username in a single realm. Use `null` for the internal realm.

The user object returned by these method has the following read-only properties:

Property	Description
id	Internal ID of the user. The user with ID 0 is the special "system user".
userName	The user name.
displayName	User display name. Defaults to user identity. `userdisplayname` setting in *login.properties* overrides this value.
realm	The user realm. `null` for the "internal" realm.
identity	The internal user identity - *username@realm*.
fullName	The user's full name.
emailAddress	The user's email address.
organization	The user's organization string.
groups	Array of strings representing the user's group membership.

10
Accessing EDQ Files Remotely

This chapter describes how to access certain directories in the EDQ directory.

10.1 Using HTTP and HTPS Server to Access EDQ Files

EDQ 14.1.2.0.0 and later allows access to EDQ configuration and log files using HTTP and HTTPS. Web access is not enabled by default. Use settings in *director.properties* and an MBean to control web access to these files.

You can configure web access for these spaces:

Space	Description	Required Permissions	Writable
baseconfig	The home configuration directory.	System administration	No
localconfig	The local configuration directory.	System administration	Yes
landingarea	The data file landing area.	Upload/ download files	Yes
projectlandingarea	The project-specific data file area.	Upload/ download files	Yes
logs	The EDQ log files.	System administration	No
wlslogs	The WebLogic logs.	System administration	No

Directories under *projectlandingarea* are EDQ project names. The calling user must have permission to access any project referenced in a path.

The web path is:

```
http://server:port/edq/files/spacename/...
```

To list the contents of a space or subdirectory use:

```
http://server:port/edq/files/space/
```

```
http://server:post/edq/files/space/directory/
```

Listing contents requires that the space has list mode enabled. See Uploading and Deleting files.

By default, lists are generated for display in the browser. To generate a list as a JSON array for application use, add *?format=json* to the URL.

The result is a JSON array in which each element contains these attributes:

Attribute	Type	Description
name	String	The entry name.
Size	Number	The file size. This attribute is not present if the entry is a directory.
directory	Boolean	**true** if entry is a directory.
timestamp	String	The modification timestamp.

Uploading and Deleting files

To upload a file to a writable space, use a `PUT` or `POST` request. To delete a file, use a `DELETE` request. You can delete a directory only if it does not contain any entries. If you try to delete a non-empty directory, the request fails with a 400 error.

Enabling Web Access

To enable web access to file spaces, add the property **fileserver.enabled.spaces** to *director.properties*. The value is a space or comma separated list of space names, with an optional mode specifier. Use **all** to enable all spaces, or **none** (or an empty list) to disable all spaces. The mode for a space is specified in brackets after the space name. The mode values can be:

Attribute	Description
r	Read only mode. Space cannot be written or listed.
rl	Read/list. Space can be read and listed, but not written.
rw	Read/write. Space can be read and written, but not listed.
rwl	Read/write/list. Space can be read, written, and listed.
w	Write only. Space cannot be read or listed.
wl	Write/list. Space cannot be read.

Non-writable spaces such as *logs* cannot be made writable.

Examples:

```
fileserver.enabled.spaces = all
fileserver.enabled.spaces = logs(rl) localconfig(rl)
fileserver.enabled.spaces = localconfig landingarea
```

By default the top level at `http://server:port/files/` cannot be listed. To allow a listing of available spaces, add `fileserver.listable = true` to *director.properties*.

Runtime Configuration

To enable configuration of the file server at runtime, add:

```
fileserver.dynamic = true
```

to *director.properties*. You can then control the set of enabled spaces and the global listable flag using the MBean with name:

```
edq:component=System,name=FileServer
```

and web path:

http://server:port/edq/management/mbeans/System/FileServer

The MBean has these attributes:

Attribute	Type	Description
enabledSpaces	Array of strings	The list of enabled spaces with modes.
listable	Boolean	The top-level listable flag.

and these operations:

Attribute	Description
disableAllSpaces()	Disable all file spaces.
enableAllSpaces()	Enable all spaces with default modes.
setEnabledSpaces(list)	Set list of enabled spaces - equivalent to setting **fileserver.enabled.spaces**.
disbleSpace(space)	Disable a named space.
enableSpace(space)	Enable a named space, with optional mode.

The MBean is web enabled and can be read and written using the REST APIs. See REST API for MBeans for more information. For example, to set the list of spaces, use a payload like:

```
{ "operation": "setEnabledSpaces",
  "arguments": ["wlslogs(rl) baseconfig(l)"]
}
```

10.2 Using SFTP Server to Access EDQ Files

EDQ is supplied with internal Secure File Transfer Protocol (SFTP) server. This server enables remote access to the configuration file area and landing area files.

The SFTP server is controlled by the `sshd.port` property in `director.properties`. The default value is 2222.

The following directories are available via the SFTP server:

Directory	Description
`config`	This corresponds to the EDQ base configuration directory (`edqhome`).
`config1`	This corresponds to the EDQ local configuration directory (`edq.local.home`).
`landingarea`	This corresponds to the `landingarea` directory in the EDQ installation.
`projectlandingarea`	This corresponds to the project specific landing areas in the EDQ installation.
`commands`	This corresponds to the `commandarea` directory in the EDQ local configuration directory (`oedq_local_home`).

11 Defining Housekeeping Rules

Housekeeping tasks help you to remove old and unwanted event log data, produced by the running of EDQ jobs in a certain time interval. You can schedule these tasks to run regularly to perform the required functions.

This chapter describes how to define housekeeping rules:

- For the Event Log Table (dn_eventlog2)
- For the Task Status Table (dn_taskstatus)

11.1 For the Event Log Table

When event log data uses a large amount of tablespace, for example when jobs are run very regularly on a schedule, you can define housekeeping rules for handling the situation.

You can activate this functionality on the server by adding an `xml` file named `eventlog.xml` to the housekeeping subfolder of the EDQ local home directory .

> **Note:**
>
> The housekeeping subfolder is not available in the EDQ local home directory by default. So you need to create one, to configure this functionality.

The interval units and parameter options together define the time older than which events are deleted when the purger runs. The interval units (denoted in days, hours, minutes or seconds) represent the time interval to check, and the parameter value specifies the measurement older than which events can be deleted.

For example - The below rule runs an event log purging housekeeping task every morning at 2 am (server time) and deletes all events log older than 10 days.

```
.
<housekeeping>
.
  <task name="eventpurger">
    <start>02:00:00</start>
    <interval units="hours">24</interval>
    <parameter>10</parameter>
  </task>
.
```

11.2 For the Task Status Table

When task status data uses a large amount of tablespace, for example when jobs are run regularly on a schedule, you can define housekeeping rules for handling the situation.

You can activate this functionality on the server by adding an xml file named `taskstatus.xml` to the housekeeping subfolder of the EDQ local home directory. The task name for the task status housekeeping should be *taskstatusdao*.

> **Note:**
>
> The housekeeping subfolder is not available in the EDQ local home directory by default. So you need to create one, to configure this functionality.

The interval units and parameter options together define the time older than which task statuses are deleted when the purger runs. The interval units (denoted in days, hours, minutes or seconds) represent the time interval to check, and the parameter value specifies the measurement older than which task statuses can be deleted.

For example - The below rule runs an task status purging housekeeping task every morning at 2.30 am (server time) and deletes all task statuses older than 100 days.

```
<housekeeping>

  <task name="taskstatusdao">
    <start>02:30:00</start>
    <interval units="hours">24</interval>
    <parameter>100</parameter>
  </task>

</housekeeping>
```

> **Note:**
>
> You need not restart the server after changing the housekeeping rules, for the changes to take effect.

12 Third-Party License Attributions

For information on the legal attributions for third party software and components included in the EDQ product, refer to Oracle Enterprise Data Quality section of Data Integration Products chapter in the *Oracle® Fusion Middleware Licensing Information User Manual*.

13 Limits in EDQ

This chapter describes the guidelines on various limits that EDQ applies when reading in and writing data:

- **Oracle VARCHAR Columns in EDQ Results Schema**

  Oracle databases 12c and later can be configured to support 32767 bytes or 4000 bytes as the maximum size of VARCHAR columns. When EDQ starts up the maximum size is detected automatically and all VARCHAR columns in results tables are created with this size.

  EDQ version 12.2.1.4.3 onwards you can configure individual columns to be marked as "long". Columns marked as long are created as VARCHAR(32767) whereas other columns are created as VARCHAR(4000).

  You can control the default column sizes by setting the following properties in the file `director.properties`:

  - `oracle.default.string.size`: Sets the size used for any column not marked as "long". The default value is 4000.
  - `oracle.max.string.size`: Sets the size used for any column marked as "long". The default value is 32767 for databases with extended strings enabled, and 4000 otherwise.

  Note that the long column selection flags are not shown if the values for **oracle.default.string.size** and **oracle.max.string.size** are equal. This could happen, for example, with databases that do not have extended strings enabled.

  This column size cannot be larger than the limit imposed by the database (4000 or 32767).

  When executing a snapshot, values longer than the maximum string size are truncated automatically, and marked as truncated in the results display for the snapshot. All other data (processor results and match data, for example) are not truncated and the process fails with an `ORA-01461: can bind a LONG value only for insert into a LONG column` or `ORA-12899: value too large for column` error, if a written value is longer than the maximum value.

  If a results table column contains long values, indexing for the column fails with an `ORA-01450: maximum key length (string) exceeded` error. This is recorded in the EDQ logs as a warning but will not cause a process to fail.

  If you wish to truncate snapshot columns to a length less than the configured maximum, set the `snapshot.max.string.size` in the file `director.properties`. For example:

  ```
  snapshot.max.string.size = 3000
  ```

  This is useful in avoiding the `ORA-01450` errors on snapshot column indexes or if data calculated from snapshot column may exceed the limit for results tables.

> **Note:**
>
> All VARCHAR columns in tables created for exports are limited to 4000 bytes. If you want to export to tables with longer columns, create the table first before configuring the export.

- **Row limit when using the xls format for Microsoft Excel** - When configuring a Microsoft Excel Data Store that uses a .xls extension, exports are limited to 65536 rows. This is because such file types support only up to this specified limit. Larger volumes of data should always use the .csv export format, though it is also possible to use the .xlsx format, provided there is sufficient memory.
  Go through the best practice guidelines listed below for a better understanding.
- **Practical row limits when using Microsoft Excel** - EDQ supports direct reading and writing of Excel files using both client-side and server-side data stores.

  Follow the below best practice guidelines:

  - The buttons on the Results Browser that enable easy sharing of results to Excel are designed for sharing any results summaries (results views showing statistics rather than data) and small samples (up to 1000 records) of data. They should not be used to attempt to export large volumes of data as this is very likely to breach client-side memory limits
  - Always use CSV file formats for writing out large volumes of data. CSV files can imported easily into Excel for data viewing.
  - If Excel is used, specify an XLSX (not XLS) file extension, and enable the option to always overwrite the file (stream data) on export. Files are always streamed during snapshots (during data import). Streaming mode uses significantly less memory when writing large XLSX files, but does not preserve worksheets and does not support append mode.

14 Backing Up and Restoring EDQ Server

This chapter provides an introduction to backing up and recovering Oracle Enterprise Data Quality server, including backup and recovery recommendations for performing disaster recovery.

Each EDQ server (Active/Passive/Production/DR) needs to be installed separately, i.e. has a separate installation of the Fusion Middleware Infrastructure and (especially) the FMW repository schemas.

> **Note:**
>
> For backup or restore of an EDQ server running on Tomcat, there is no FMW Infrastructure, but each server should be installed separately.

The timezone of each EDQ server must be the same in order to ensure that the configuration logic is identical, as the server timezone can play a role in Date/Time conversions in EDQ processes.

Perform the following steps for backing up and restoring an instance of EDQ on Oracle WebLogic server:

To back up an EDQ server:

1. Stop the server (to ensure the database is static).
2. Backup the `EDQCONFIG` schema.
   It is not normally necessary or advisable to backup the `EDQSTAGING` or `EDQRESULTS` schemas as they generally contain only temporary data that can be restored by re-running jobs. An exception is that `EDQRESULTS` should be backed up, if you need to see results (For example- in Director) of previously run jobs, for example any Results Books that have not been exported. In this case, it is advisable to minimize the size of the `EDQRESULTS` schema before backing up by purging any old projects for which results are not required, and deleting any projects that are no longer needed (backing them up by packaging them to DXI files first, if required).
3. Backup the files in the EDQ Local Home area, with the exception of the logs directory.

To restore an EDQ server:
For WebLogic server:

1. Stop the server.
2. Restore the `EDQCONFIG` schema from backup, dropping the "fresh" schema created on the passive instance by RCU and restoring to the same database name. All other schemas, including the `EDQSTAGING` and `EDQRESULTS` schemas, should be freshly initialized as created by RCU, but with sufficient tablespace configured to be operational. If `EDQRESULTS` was backed up, restore this as well.
3. Restore the files in the EDQ local home, with the exception of the backed up `director.properties` file. Settings from this should be merged carefully on to the restored instance, to ensure that the pointers to the EDQ databases are correct. This means, on the

Oracle WebLogic server, the configured data sources in the domain are correct, and the `director.properties` file can be restored as is, with no impact.

4. Restart the server and test by running jobs.

For Tomcat server:

1. Stop the server.
2. Restore the `EDQCONFIG` schema. Restore `EDQRESULTS` also if this was backed up. Otherwise, create empty `EDQRESULTS` schema for a new installation.
3. Restore the files as above, with the exception of `director.properties`.
4. Ensure the two pointers to `EDQCONFIG` and `EDQRESULTS` in `director.properties` are correct.
5. Restart EDQ.

15

Configuring Schema Password Expiry Warnings and Wallet Refresh

This chapter describes how to manage schema password expiry and wallet refreshes to maintain installations that use an Oracle database for the configuration and results schemas.

This chapter contains the following topics:

- Configuring Schema Password Expiry Warnings
- Configuring Schema Password Reset
- Configuring Automatic Wallet Refresh

15.1 Configuring Schema Password Expiry Warnings

When EDQ uses an Oracle database for the configuration and results schemas, a task is run periodically to check the password expiry time of the schema passwords. If an expiry time is found within a defined threshold, EDQ can generate warnings.

This topic covers:

- Configuration
- Triggers

Configuration

To configure the password expiry checks and notifications use the following properties in *director.properties* within the EDQ local home directory:

Property	Description	Default Value
`schema.password.expiry.check.interval`	Interval between expiry checks. Value must not be less than 60s. Set to 0 to disable expiry checking.	1d Valid suffixes are `d` (days), `h` (hours), `m` (minutes), `s` (seconds) If no suffix is specified, the default suffix will be `d` (days).

Property	Description	Default Value
`schema.password.expiry.warning.threshold`	Generates warnings when the expiry time is within this interval.	7d Valid suffixes are `d` (days), `h` (hours), `m` (minutes), `s` (seconds) If no suffix is specified, the default suffix will be `d` (days).

`schema.password.expiry.warning.frequency`	Specifies the number of expiry checks after which to generate warnings.
	Use this option to limit the number of warnings generated. For example, if you want more frequent checks for expiry, but do not want warnings on every check, set the following:
	`schema.password.expiry.check.interval = 12h`
	`schema.password.expiry.warning.frequency = 4`
	EDQ will check for password expiry every 12 hours, but generate warnings every other day.
		1

`schema.password.expiry.warning.emails`	Space or comma separated list of email addresses used by built-in trigger.	

Triggers

Expiry warnings are generated by running triggers with these paths:

```
/schema/config/expiring
/schema/results/expiring
```

There is a built-in trigger that sends mails to the addresses configured with the `schema.password.expiry.warning.emails` property. You can specify multiple addresses by using commas or spaces. Specify SMTP details in *mail.properties* with `enabled = true` for emails to be sent.

You can also define custom triggers for additional flexibility. The arguments to the trigger are:

- `label` - Schema label - "config" or "results"
- `user` - Database username for schema
- `date` - Expiry time

Here's an example that logs a message, generates a push notification, and sends an email:

```
addLibrary("logging")
addLibrary("webpush")
addLibrary("mail")

function getPath() {
 return "/schema/(config|results)/expiring"
}

function run(path, id, env, label, user, date) {
  logger.log(Level.INFO, "{0} [{1}] expiring {2}", label, user, date);

  var p = WebPush.create(`${label} schema password will expire on ${date}`)

  p.title = "Database password expiry warning"
  p.push()

  var mh  = Mail.open({enabled : true});
  var msg = mh.newMessage("Database password expiry warning")

  msg.text = `${label} schema password will expire on ${date}`
  msg.addTo("admin@example.com")
  msg.type = "text/plain";
  msg.send()
}
```

15.2 Configuring Schema Password Reset

> **Note:**
>
> This information is applicable to EDQ installations running on Apache Tomcat environments only.

For EDQ running on Tomcat where the database URLs and credentials are configured in *director.properties*, you can change the schema passwords in the database without the need to edit *director.properties* and restart the server. You can trigger a schema password reset in any of the following ways:

- Configuring Automatic Reset of Schema Password
- Resetting Schema Password Using REST API
- Resetting Schema Password Using a Script Library in a Trigger

This topic also covers:

- Triggers
- Clustering Considerations
- Password Strength

Configuring Automatic Reset of Schema Password

To configure automatic password reset, set the property `schema.password.auto.reset.after`. This value specifies the number of times password expiry is detected before the password is reset in the database. To disable automatic reset, set the value to `-1`.

Examples
To reset the password immediately when expiry is detected by automatic checks, set the following:

```
schema.password.auto.reset.after = 0
```

To reset the password after 5 warnings, set the following.

```
schema.password.auto.reset.after = 5
```

If `schema.password.expiry.check.interval` is set at one day (the default), this setting gives the administrator five days to update the password manually before the automatic reset occurs.

Resetting Schema Password Using REST API

To use a system administration REST API to reset the schema password, use the following interface. Note that the user must have the system administration permission to run this request.

```
POST https://server/edq/admin/schemas/setpassword
```

The payload to the request contains the attributes listed in the following table:

Attribute	Description
label	Required. Schema label. The value must be "config" or "results".
password	Required. The new password. Use "" or "%" to specify a random password.

Resetting Schema Password Using a Script Library in a Trigger

Trigger scripts can update schema passwords using a script library. To use the library, add this line to the top of the script:

```
addLibrary("schemas")
```

The library publishes the following constant fields:

Name	Description	Value
Schemas.CONFIG	Internal label for config schema.	config
Schemas.RESULTS	Internal label for results schema.	results

and an object `schemas` with the following method:

```
schemas.updatePassword(label [, password])
```

This updates the password for one of the schemas. `label` identifies the schemas and must be set to "config" or "results". If `password` is omitted, a random password is used.

The following is an example that uses the expiry warning trigger:

```
addLibrary("schemas")

function getPath() {
 return "/schema/(config|results)/expiring"
}

function run(path, id, env, label, user, date) {
  schemas.updatePassword(label);
}
```

Triggers

Password reset for a schema runs triggers with these paths:

```
/schema/config/passwordreset
/schema/results/passwordreset
```

Similar to schema password expiry warnings, there is a built-in trigger that sends mails to the addresses configured with the `schema.password.expiry.warning.emails` property. You can specify multiple addresses by using commas or spaces. Specify SMTP details in *mail.properties* with `enabled = true` for emails to be sent.

You can also define custom triggers for additional flexibility. The arguments to the trigger are:

- `label` - Schema label - "config" or "results"
- `user` - Database username for schema
- `password` - The new password

Note that emails that are generated by the default trigger do not include the new password.

Clustering Considerations

If EDQ is running in a cluster of Tomcat servers, the default update process assumes that the same *director.properties* file is shared amongst all the servers. The internal data source passwords are updated on all servers, but the properties are written on a single server only. If each server has a distinct *director.properties*, you need to set the following:

```
schema.password.shared.properties = false
```

Password Strength

Random passwords for Oracle are constructed with a fixed length and minimum counts of upper and lower case letters, and fixed counts of digits and special characters (-_#). The counts can be overridden using these properties:

Property	Description	Default Value
oracle.pw.length	Password length.	12
oracle.pw.lower	Minimum number of lower case letters.	2
oracle.pw.upper	Minimum number of lower case letters.	2
oracle.pw.numeric	Number of digits.	2
oracle.pw.special	Number of special characters.	2

15.3 Configuring Automatic Wallet Refresh

> **Note:**
>
> This information is applicable to EDQ installations running on Apache Tomcat environments only.

If EDQ is using an Autonomous Database instance as its repository database with mTLS enabled, the wallet files need periodic refresh since the embedded certificates have a limited lifetime. EDQ can be configured to refresh wallet files that are older than a defined time.

This topic covers the following:

- Prerequisites to Configure Automatic Wallet Refresh
- Configuring Automatic Wallet Refresh

Prerequisites to Configure Automatic Wallet Refresh

To support automatic wallet refresh, you must configure the OCID of the Autonomous Database instance using the following properties:

Property	Description
`dataSource.adb.ocid`	OCID of configuration schema database.
`resultsDataSource.adb.ocid`	OCID of results schema database.

Additionally, the JDBC URLs must be as follows:

```
jdbc:oracle:thin:@service?TNS_ADMIN=/pathtowalletdirectory
```

If both schemas use the same database instance and the same wallet directory, set only `dataSource.adb.ocid`. Do not set `resultsDataSource.adb.ocid`.

Configuring Automatic Wallet Refresh

To enable automatic wallet refresh, set the property `schema.wallet.refresh.interval`. This is the minimum wallet age after which a refresh is performed. The age of a wallet is determined from the modification time of the *cwallet.sso* file. The property value is a duration with d/h/m/s suffixes. If no suffix is present the value is treated as a number of days.

Examples

```
schema.wallet.refresh.interval = 30d
schema.wallet.refresh.interval = 60
```

16 Using the User Garbage Collection Tool to Manage Deleted Users

This chapter describes how to use the `usergc.jar` tool to identify and cleanup deleted users.

A user that has been deleted from the internal store or from any external store, such as LDAP, may generate a record in the **dn_usergraveyard** table. Such users then appear in case management assigned to or modified by searches and can never be cleaned up.

EDQ 14.1.2.0.0 introduces a the usergc.jar tool, which can check whether deleted users are referenced in any of the configuration schema tables.

Execute the following command to run the tool:

```
$ java -jar usergc.jar [-tables list] [-summary] [-clean file] dbspecifier
```

The final argument specifies the connection to the configuration schema.

For Oracle the specifier has the format:

`oracle:SID@HOST:PORT/USER/PASSWORD`

or

`oracle:#SERVICE@HOST:PORT/USER/PASSWORD`

Here,

- **SID** is the database SID.
- **SERVICE** is the database service name.
- **HOST** is the database host name.
- **PORT** is the listener port. You can omit the :PORT part if the port is the default 1521.
- **USER** and **PASSWORD** are the credentials for the configuration schema.

For Derby the specifier has the format:

`derby:(PATHTODB)`

Here **PATHTODB** is the location of the Derby database directory.

The usergc.jar tool scans tables in the database and records usage by users that are marked as deleted in the **dn_usergraveyard** table. The tool dose not consider usage in the audit log and session history tables because these do not impact functionality.

The other command parameters that you can use are:

-tables list - To check only the tables in the comma-separated list. This parameter can be used to speed up the check when usage only for specific tables is required.

-summary - To list only the number of occurrences of each user in each table. If **-summary** is not used, then details of each usage are output. For example, for usage in the **dn_case** table, the external IDs of each record are output. The detailed listing can be used in the case management to guide updates to remove references to users, but this may be too verbose in for the initial scan.

-clean file - To generate SQL in the file to remove users that are not referenced in any table. Do not use this in conjunction with **-tables list** because some reference to the user could be missed.

17

Updating Database Passwords using setpws.jar

This chapter describes how to use the `setpws.jar` tool to update database passwords in *director.properties*. This information is applicable to Apache Tomcat environments only.

Schema passwords are often stored in an encrypted format in the *director.properties* file, which can be difficult to update when schema passwords are refreshed. EDQ 12.2.1.4.3 introduces a tool called `setpws.jar` that you can use to update EDQ configuration with new passwords for configuration and results schemas.

Execute the following command to see the usage summary to run the tool:

```
$ java -jar setpws.jar -help
Updates EDQ configuration with new passwords for configuration and results 
schemas.

Available options:

-confdir          EDQ local configuration directory.
-setdb            Update schema passwords in database.  Current passwords 
must not be expired.
-setdbadmin       Update schema passwords in database using administrator 
account.
-adminpw          Database admin/system password.  Used if -setdbadmin is 
present.
-configpw         New password for configuration schema
                    Use - to leave password unchanged.
                    Use % to generate a random password.
-resultspw        New password for results schema.
                    Use - to leave password unchanged.
                    Use % to generate a random password.

Passwords not specified on the command line can be entered interactively.
The configuration directory defaults to /opt/edq/oedq.local.home on OCI 
systems.
```

Here,

- **-confdir** is the location of the EDQ local configuration directory that contains *director.properties* and *kfile*. On OCI systems this defaults to */opt/edq/oedq.local.home*, which is the location used in the Marketplace images.
- **-setdb** updates the schema passwords in the database along with other changes to *director.properties*. This option can be used to refresh passwords in one step.

> **Note:**
>
> This option will not update schema passwords that have expired. To change expired passwords use **-setdbadmin**.

- **-setdbadmin** updates the schema passwords in the database using an administrator account (system or admin for Autonomous Database instances). Use this instead of **-setdb** if the existing passwords have expired.
- **-adminpw** lets you specify the database administrator password. Use this along with **-setdbadmin**. If this option is omitted, the tool prompts for the new password.
- **-configpw** lets you specify the new configuration schema password. If the value is %, a random password is generated. Using random passwords with **-setdb** or **-setdbadmin** is the simplest way to update passwords. Use - to leave the password unchanged. If this option is omitted, the tool prompts for the new password.
- **-resultspw** lets you specify the new results schema password. If this option is omitted, the tool prompts for the new password.

For EDQ instances that are created from the Oracle Cloud Infrastructure, this tool also refreshes wallets for Autonomous Database schemas. On such instances, run the tool using the following command:

```
$ sudo -u tomcat /opt/java/bin/java -jar /opt/edq/edq/oracle.edq/
setpws.jar ...
```

18 Using the Local Web Content Directory

EDQ 12.2.1.4.4 adds a new "local" web folder to store additional static web content to an EDQ application installation, for example, to support the display of a custom image in a web notification, or in an extended user application. This chapter describes how to use the local web content directory.

The URL for files in the local area is:

```
http://server:port/edq/local/
```

This chapter contains the following topics:

- Location of the Local Web Content Directory
- Populating the Local Web Content Directory
- Examples

18.1 Location of the Local Web Content Directory

The default location for the directory containing local content is `localcontent` in the EDQ "local home" configuration directory. You can set an alternative location by setting `localcontent.directory` in *director.properties*. If the value is not an absolute location it is interpreted as a relative path to the local configuration directory.

Examples:

```
localcontent.directory = /opt/EDQ/content
```

```
localcontent.directory = local/branding
```

If the value of `localcontent.directory` is empty, the local content web location is not enabled and references to URLs within `/edq/local` will return a 404 response.

If the URL path is empty or refers to a directory, the server uses the *index.html* file. You can specify alternative "welcome" files with the `localcontent.welcome` setting in *director.properties*. The value is a whitespace or comma separated list of file names.

Example:

```
localcontent.welcome = index.htm
```

18.2 Populating the Local Web Content Directory

You can add files to the local content directory using any of these mechanisms:

- Normal OS copy operations
  Note that the files and directories must be readable by the user running the application server.
- The built-in EDQ sshd server, which exposes the directory with the name "localcontent".
- POST or PUT requests to `http://server:port/edq/local/path`

The user making the request must have the "upload files" permission. Directories used in the path are created automatically.

18.3 Examples

A branding image in Watchlist screening:

```
leftbranding.imagetop    = */local/companylogo.png
```

Adding an image to a push notification:

```
var push = WebPush.create("Job complete on " + server.serverInfo.name)

push.title = "Job notification"
push.image = "local/agamemnon.jpg"
...
```

19 Configuring Secure Script Execution

Processors that support direct entry of scripts in JavaScript or Groovy raised security concerns because these processors could access Java classes directly and perform sensitive operations when executed on the EDQ server. To address this issue, EDQ versions before 14.1.2.0.0 could be configured to use the Java Security Manager framework for secure script execution. However, this feature is deprecated in JDK version 17 and later.

EDQ 14.1.2 introduces a configuration property called `scriptprocessor.security` to control the behavior of direct entry script processors and gadgets. When `scriptprocessor.security` is set to `true`, EDQ exhibits the following behavior:

- Groovy scripts are not allowed. You can no longer sandbox Groovy to prevent direct use of Java classes.
- JavaScript scripts cannot reference any Java classes directly.
- Scripts cannot use the **importClass** and **importClassSet** functions.
- Scripts by default cannot use any script libraries (loaded by **addLibrary**).

Note that this enhancement applies to the use of scripts in direct-entry script processors and gadgets only. This does not affect all the other uses of scripts (for example, triggers and so on).

Using libraries in scripts

When script processor security is enabled, scripts cannot load any of the script libraries provided with EDQ. To enable the use of selected libraries, create a file named `libraries.json` in `localconfig/security/scripting`. The JSON file should contain an object keyed by library name. The value associated with each key is an object containing an available attribute. If this is `true`, the library can be used in scripts.

For example:

```
{ "logging": {"available": true},
  "oci":     {"available": true}
}
```

Preventing infinite loops

Other than the direct insecure use of Java classes, there is another issue with direct entry scripts where a user can create an infinite loop. When the processor is executed, the EDQ server may remain unresponsive indefinitely.

To guard against such loops, you can use the following additional property when you enable script processor security:

```
script.processor.statement.limit = LIMIT
```

This sets a limit on the number of script statements that can be executed in a single invocation of the script. The script will fail if the limit is exceeded. Note that you may need to tune and optimize your JavaScript statements to reach a value that does not generate errors for valid scripts.