Oracle® Fusion Middleware

Administering Oracle Enterprise Data Quality

12*c* (12.1.3)

E51652-01

May 2014

Describes how to administer Oracle Fusion Middleware servers.

ORACLE

Oracle Fusion Middleware Administering Oracle Enterprise Data Quality, 12*c* (12.1.3)

E51652-01

Copyright © 2014 Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate failsafe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

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 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 following documents in the Oracle Enterprise Data Quality documentation set.

EDQ Documentation Library

The following publications are provided to help you install and use EDQ:

- *Oracle Fusion Middleware Release Notes for Enterprise Data Quality*
- *Oracle Fusion Middleware Installing and Configuring Enterprise Data Quality*
- *Oracle Fusion Middleware Administering Enterprise Data Quality*
- *Oracle Fusion Middleware Understanding Enterprise Data Quality*
- *Oracle Fusion Middleware Integrating Enterprise Data Quality With External Systems*

- *Oracle Fusion Middleware Securing Oracle Enterprise Data Quality*
- *Oracle Enterprise Data Quality Address Verification Server Installation and Upgrade Guide*
- *Oracle Enterprise Data Quality Address Verification Server Release Notes*

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

`http://http://docs.oracle.com`

Online Help

Online help is provided for all Oracle Enterprise Data Quality 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:

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 Section 1.4, "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.
`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 Example 3 for how to use the `load` chore with the `Rules` schema to load purge rules.
`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 Section 1.2, "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 Section 1.4, "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>
```

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 Section 1.2.

```
<?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 EDQ to produce email notifications in a number of situations.

- Section 2.1, "Using SMTP to Send Email Notifications"
- Section 2.2, "Using JNDI to Send Email Notifications"
- Section 2.3, "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. This 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
mail.user = depends on client site
mail.password = depends on client site
from.address = edqserver@example.com
```

2.2 Using JNDI to Send Email Notifications

As an alternative to using SMTP, you can use a Java Naming and Directory Interface (JNDI) session by configuring the following properties:

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

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 Case Management

This chapter describes how to configure EDQ to use Case Management.

This chapter includes the following sections:

- Section 3.1, "Understanding and Adding Extended Attributes"
- Section 0.3, "Configuring Data Entry Validation"
- Section 0.4, "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 EDQ 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.

3.1 Understanding and Adding Extended Attributes

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

3.1.1 Default Extended Attributes

In an initial EDQ 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.

3.1.2 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

Tuning EDQ Performance

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

This chapter includes the following topics:

EDQ 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 EDQ 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.

4.1 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:

`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.

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.

4.2 Tuning for Batch Processing

The default tuning settings provided with EDQ 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.

4.3 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.

4.3.1 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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.4 Tuning JVM Parameters

JVM parameters should be configured during the installation of EDQ. For more information, see *Oracle Fusion Middleware Installing and Configuring Enterprise Data Quality*. 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.

4.4.1 Setting the PermGen Space

If the following error message is reported in the log file, it may be necessary to increase the maximum PermGen space available:

```
java.lang.OutOfMemoryError: PermGen space
```

To do this, change the value against the `-XX:MaxPermSize` parameter on the JVM on the EDQ server. It will also be necessary to change the `-XX:ReservedCodeCacheSize` parameter proportionally. For example, if the `MaxPermSize` is doubled from 1024m to 2048m, the `ReservedCodeCacheSize` should be doubled.

4.4.2 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 EDQ 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.

4.5 Tuning Database Parameters

The most significant database tuning parameter with respect to performance tuning within EDQ 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 EDQ 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.

4.6 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.

EDQ 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 `config/properties` directory of the EDQ server to 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

Using JMX Extensions to Monitor EDQ

This chapter describes theEDQ 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:

5.1 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 EDQ 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 EDQ 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 EDQ 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
```

5.2 Understanding JMX Bean Naming

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

The names used for the EDQ 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

5.2.1 Reviewing the Example

This example demonstrates how to modify the default EDQ 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.

5.3 Monitoring Real-Time Processes

EDQ 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.

5.3.1 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.

5.3.2 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:

`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:

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

6

Using Triggers

This chapter describes how to use the trigger functionality in Oracle Enterprise Data Quality. This document describes where triggers are installed, how to call them, and how you can use them.

This chapter contains the following topics:

6.1 Overview of the Triggers Functionality

Triggers in Oracle Enterprise Data Quality 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.

6.1.1 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.

6.1.2 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 Section 6.6, "Understanding EDQ Trigger Points."

Custom triggers are described in the rest of this document.

6.2 Required Skills to Use Triggers

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

6.3 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.

6.4 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 Section 6.6, "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 Section 6.6, "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 Section 6.8, "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 Section 6.10, "Exposing Triggers in a Job

Configuration." Getting trigger names and exposing them in the Director interface is only possible with the job configuration screen.

[*Is this the actual name of the screen? Or should it be Phase Configuration?*]

6.5 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.get TriggerConfigFiles(*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.

6.6 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`*

Component	Description
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*

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*

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 Section 6.7 for decriptions of these methods.

6.7 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 6–1 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.

Table 6–1 (Cont.) Methods Associated with the TriggerInfo Trigger Point

Method	Data Returned	Description
`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.
`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.

6.8 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 6–1 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.

Example 6–1 Setting Trigger Levels

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

Another way to modify the level is to define a `getLevel` function in the trigger. See Section 6.4 for a description.

6.9 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)
   ```

6.10 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 Section 6.4 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 Section 6.4 for more information about `getTriggerNames`.

6.11 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];
}
```

7

Accessing EDQ Files Remotely

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

EDQ is supplied with internal File Transfer Protocol (FTP) and Secure File Transfer Protocol (SFTP) servers. These servers enable remote access to the configuration file area and landing area files.

The FTP server can be accessed with a third-party FTP client using any valid EDQ username and password, connecting to the port specified by the `ftpserver.port` in the `director.properties` file.

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 FTP and SFTP servers:

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 base configuration directory (`edqhome`).
`commands1`	This corresponds to the `commandarea` directory in the EDQ local configuration directory (`oedq_local_home`).