# *Modulant Curtana Developer's Guide*

Version 1.0

September 2001

## *Modulant Curtana Developer's Guide*, version 1.0

Copyright © 2001 Modulant Solutions, Inc. All rights reserved.

September 2001, Version 1.0

**Ownership of Materials.** This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The contents of this manual are furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Modulant. Modulant assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

This manual is protected by copyright and distributed under licenses restricting its use, copying, translation, distribution, and decompilation. Except as permitted by such licenses, no part of this manual may be reproduced in any form by any means without prior written authorization of Modulant. Except as expressly provided herein, Modulant grants no express or implied rights to anyone under any patents, copyrights, trademarks, trade names, or trade secret information with respect to the contents of the manual.

**Ownership of Trademarks.** The trademarks, service marks, product names, company names or logos and other marks displayed in the manual are the property of Modulant Solutions, Inc. or other third parties. Any use of trademarks, service marks, product names, company names or logos, and other marks, including the reproduction, modification, distribution, or republication of same without the prior written permission of the owner is strictly prohibited.

Modulant, the Modulant logo, Curtana, and Balisarda are trademarks of Modulant Solutions, Inc. Other trademarks, service marks, trade names and company logos referenced are the property of their respective owners.

**Disclaimers.** THIS MANUAL IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. FURTHER MODULANT DOES NOT WARRANTY, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

**Notice to U.S. Government Users.** All Modulant products and publications are commercial in nature. The software and documentation are "commercial items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are licensed to U.S. Government end users (A) only as Commercial Items and (B) with only those rights as are granted to all other end users pursuant to the terms and conditions set forth in the Modulant standard commercial agreement for this software. Unpublished rights reserved under the copyright laws of the United States.

# Table of Contents


**List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v**

**List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii**

**List of Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix**

**Preface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xi**

Who Should Read this Guide? . . . . . . . . . . . . . . . . . . . . . . . . . . . . xi
Conventions Used in this Guide . . . . . . . . . . . . . . . . . . . . . . . . . . . xii
What's in this Guide? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii
Related Documents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii

**1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1**

The Data Transformation Process . . . . . . . . . . . . . . . . . . . . . . . . . . 2
Specifying a Transformation Job . . . . . . . . . . . . . . . . . . . . . . . . . 3
The Life Cycle of a Queued Job . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Using the Lifecycle Manager API . . . . . . . . . . . . . . . . . . . . . . . . 7**

Configuring the Modulant Curtana Transformation Engine . . . . . . . . . . . 8
Setting Up the Runtime Environment . . . . . . . . . . . . . . . . . . . . . . . 9
Invoking the Lifecycle Manager . . . . . . . . . . . . . . . . . . . . . . . . . . 10
Adding a Job to the Queue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
Getting the Status of a Job . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
Removing a Job from the Queue . . . . . . . . . . . . . . . . . . . . . . . . . . 15
Using the Lifecycle Manager from the Command Line . . . . . . . . . . . . . 17
Running the Lifecycle Queue . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
Adding a Transformation Job to the Queue . . . . . . . . . . . . . . . . . . . 18
Getting the Status of a Queued Job . . . . . . . . . . . . . . . . . . . . . . . 19
Removing a Transformation Job from the Queue . . . . . . . . . . . . . . . . 20

**3 Lifecycle Manager API Reference . . . . . . . . . . 21**
LifecycleManager Class . . . . . . . . . . 22
addJob() . . . . . . . . . . 23
getInstance() . . . . . . . . . . 24
getJobUpdate() . . . . . . . . . . 24
removeJob() . . . . . . . . . . 25
QueuedJobInfo Interface . . . . . . . . . . 26
getConfigFilePath() . . . . . . . . . . 27
getErrorFilePath() . . . . . . . . . . 27
getID() . . . . . . . . . . 27
getRunEndTime() . . . . . . . . . . 28
getRunStartTime() . . . . . . . . . . 28
getStatus() . . . . . . . . . . 29
getStatusCode() . . . . . . . . . . 30
getStatusMessage() . . . . . . . . . . 30
getSubmitTime() . . . . . . . . . . 31
LifecycleQueue Class . . . . . . . . . . 31

**A *run_config*.xml Reference . . . . . . . . . . 35**
section name="engine" . . . . . . . . . . 37
section name="atsSource" . . . . . . . . . . 38
section name="atsTarget" . . . . . . . . . . 39

**B Configuring Database Connections . . . . . . . . . . 41**
Editing config.xml . . . . . . . . . . 43
Editing datasources.xml . . . . . . . . . . 43
Specifying the Work Data Source . . . . . . . . . . 44
Specifying the Control Data Source . . . . . . . . . . 45

**Index . . . . . . . . . . 47**

# List of Figures

Figure 1: The Role of the Lifecycle Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
Figure 2: The Lifecycle Manager Queue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
Figure 3: Life Cycle of a Queued Job . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

# List of Tables

Table 1: Specifying Transformation Run Stages . . . 3
Table 2: Queued Job Status Values . . . 29
Table 3: The Engine Section of the Configuration File . . . 37
Table 4: The ATS Source Section of the Configuration File . . . 39
Table 5: The ATS Target Section of the Configuration File . . . 40
Table 6: The Lifecycle Manager Section of config.xml . . . 43
Table 7: The Work Data Source Section of datasources.xml . . . 44

# List of Examples


Example 1: Starting the Lifecycle Manager and Adding a Job . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
Example 2: Getting Status Information About a Queued Job . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
Example 3: Removing a Transformation Job from the Queue . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
Example 4: LifecycleQueue Usage Message . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
Example 5: Adding a Job from the Command Line . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Example 6: Getting the Status of a Queued Job. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Example 7: Removing a Job from the Queue. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
Example 8: Configuration File Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Example 9: Database Connection File Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

# Preface

The Modulant Curtana Interoperability Platform marks the first release of Modulant's revolutionary toolset for creating true data interoperability. These tools pave the way to move from application integration and custom solutions to a more universal solution.

The *Modulant Curtana Developer's Guide* describes how to use the Lifecycle Manager API to define a queue of transformation jobs to run independently of the Modulant Curtana Analyst Tool.

This preface contains the following topics:


- Who Should Read this Guide?
- Conventions Used in this Guide
- What's in this Guide?
- Related Documents


## Who Should Read this Guide?

The audience for the Modulant Curtana Interoperability Platform includes integration architects, domain experts, and software developers.

- Integration architects create the mappings from data sources to the Abstract Conceptual Model, and perform transformation runs.
- Domain experts work with integration architects to help them understand both the structure and the context of their community's data.
- Java programmers can use the Lifecycle Manager API to automate the process of transforming data from one or more data sources to another.

This guide assumes that you are familiar with the following topics:

- The Windows operating system
- Data modeling and XML representation of data
- Your application domain
- Relational database management systems (RDBMSs), including Oracle
- Java programming

## Conventions Used in this Guide

The manuals in the Modulant Curtana Interoperability Platform documentation set use the following typographic conventions:

| | |
|---|---|
| **bold text** | File names, XML elements, user interface controls, and language keywords. |
| ***bold italic text*** | Variable elements; for example, parameters in code syntax. |
| *italic text* | New terminology; also emphasized words and book titles. |
| SMALL CAPS | Names of keys on the keyboard. |
| `monospace text` | Examples, such as XML fragments or Java code; or text you type exactly as it appears. |

Descriptions of procedures also use the following conventions:

| | |
|---|---|
| **File > Import** | A menu path to follow; in this example, from the **File** menu, select **Import**. |
| CTRL+C | Press both keys at the same time. |
| ESC F I | Press and release each key in succession. |

The manuals contain notes, tips, and warnings that provide particular information, as follows:

- *Notes* provide related information that does not fit directly into the flow of the surrounding text.
- *Tips* provide hints containing shortcuts or alternative ways of accomplishing a task.
- *Warnings* contain critical information that could prevent physical damage to equipment, data, or people.

Some of the diagrams in the manuals use EXPRESS-G graphical notation. For an explanation of the symbols in these diagrams, see the *Modulant Balisarda Mapping Tool Guide*.

Discussions of XML files contain diagrams that show the structure of the associated DTDs (document type definitions). Each of these diagrams contains a legend describing the symbols in the diagram.

**About Sidebars**

As you read the documentation, you will encounter information contained in sidebars like this one. These sidebars provide background material related to the surrounding information.

## What’s in this Guide?

This guide contains the following topics:

- Chapter 1, “Introduction,” describes the basic operation of the Lifecycle Manager API and how it can queue transformation run jobs.
- Chapter 2, “Using the Lifecycle Manager API,” lists the things you can do with the Lifecycle Manager API, and shows examples of how to accomplish each task.
- Chapter 3, “Lifecycle Manager API Reference,” describes the major classes and interfaces in the Lifecycle Manager API, with a description of each of the public methods.
- Appendix A, “run_config.xml Reference,” shows the format of the configuration file that you create to specify the parameters of a transformation run job.
- Appendix B, “Configuring Database Connections,” describes how to modify the system configuration files to specify the database connection information required by the Lifecycle Manager.

## Related Documents

In addition to this guide, the Modulant Curtana Interoperability Platform Documentation Library contains the following manuals:

- *System Overview*: Introduces the Modulant Curtana Interoperability Platform and the associated methodology, and describes all of the included tools and components.
- *Modulant Balisarda Mapping Tool Guide*: Describes the mapping process, the elements of the Abstract Conceptual Model, and how to use the Modulant Balisarda Mapping Tool to create a mapping specification.
- *Modulant Curtana Analyst Tool Guide*: Provides detailed explanations of each of the parts of the Modulant Curtana Analyst Tool, with information on troubleshooting the results of data transformations.
- *Installation Guide*: Describes the system requirements for the Modulant Curtana Interoperability Platform, and walks you through the installation and configuration process.

In addition to the printed documents, your Modulant Curtana installation contains a complete online Documentation Library, in the **docs** subdirectory. You can find the online Documentation Library in both HTML and PDF format.

To access the Modulant Curtana Documentation Library:

- From the Windows desktop, select **Start > Programs > Modulant > Documentation Library**.

# 1 Introduction

The Modulant Curtana Interoperability Platform is a collection of tools and components that enable interoperability between different types of machines, platforms, and applications. It fosters interoperability by sharing data—including the data's full semantics and the context of the data usage. Preserving semantics and context enables the Modulant Curtana Transformation Engine to resolve potential conflicts in incompatible data.

To create true interoperability, the Modulant Curtana platform transforms data from one source to another by mapping each data source to an abstract representation—the *Abstract Conceptual Model*. The Modulant Curtana Transformation Engine reads mapping specifications for each data source (known as *Application Transaction Sets*, or ATSs) and populates the Abstract Conceptual Model with data from one or more source ATSs. Then it extracts the data from the Abstract Conceptual Model to a target ATS, preserving native context through the entire process.

To perform these data transformations, or *transformation runs*, you can either use the Modulant Curtana Analyst Tool to define and monitor each run, or you can use the Lifecycle Manager API to automate the process and run transformations as batch jobs.

Depending on your system environment, you can use both the Analyst Tool and the Lifecycle Manager API, or you can use either method independently of the other. The Modulant Curtana platform uses an internal database to manage the process of a transformation run. If you use both the Analyst Tool and the Lifecycle Manager API, you are responsible for coordinating concurrent use of this internal database.

The Lifecycle Manager API creates a queue of transformation jobs, which it processes in order. Using this queue, you can define batch transformation jobs to run without operator intervention. To specify a transformation run as a batch job, you create a run configuration file in XML format with the parameters of the job.

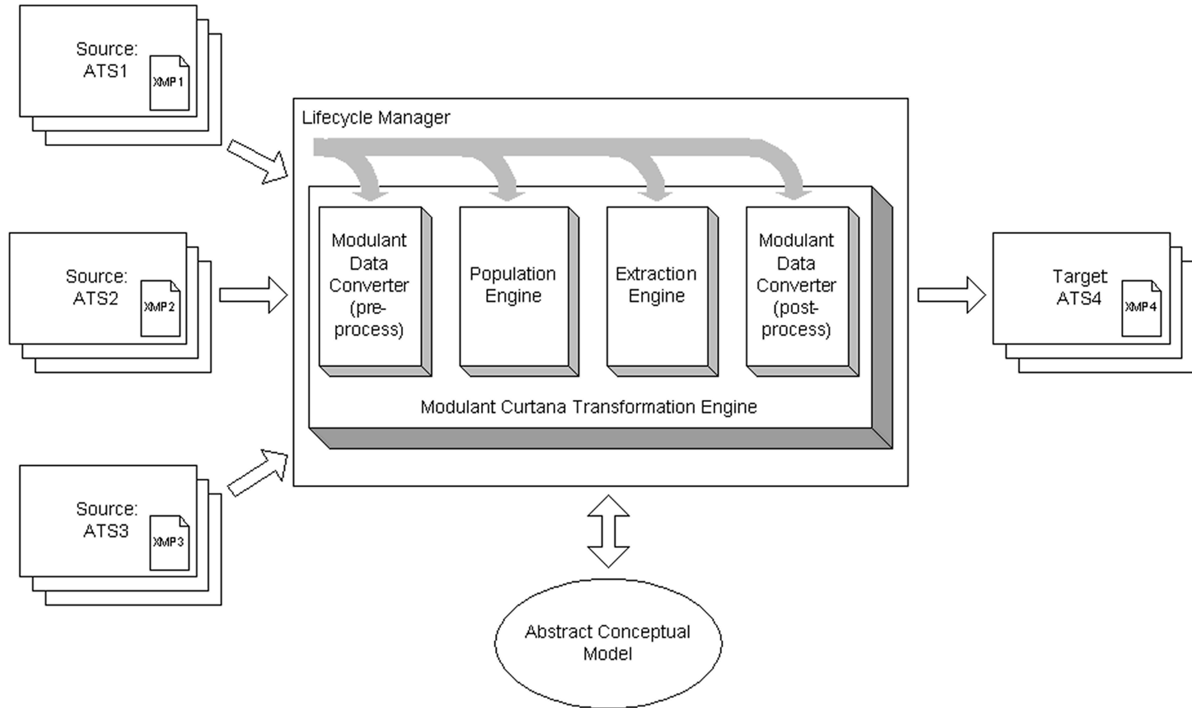
The following topics introduce the APIs you can use to automate Modulant Curtana transformation runs:

- The Data Transformation Process
- The Life Cycle of a Queued Job

## The Data Transformation Process

Figure 1 shows how the Lifecycle Manager controls a transformation run with three source ATSs.

**Figure 1: The Role of the Lifecycle Manager**




To transform data from a source ATS through the Abstract Conceptual Model to a target ATS, the Transformation Engine goes through the following stages:

1. Read the run configuration file.
2. For each source ATS:
   a. Read the XML DTD and the XMP (XML MaP) file for that ATS.
   b. Load the ATS data into the internal database.
   c. If the XMP file contains a **PREPROCESSOR** conversion list, invoke the Modulant Curtana Data Converter to perform the specified conversions.
3. Invoke the Population Engine to populate the Abstract Conceptual Model with the source ATS data.
4. Invoke the Extraction Engine to extract data from the Abstract Conceptual Model into the target ATS tables in the internal database.
5. If the XMP file for the target ATS contains a **POSTPROCESSOR** conversion list, invoke the Data Converter to perform the specified conversions.
6. Extract the target ATS data from the internal database in XML format.

## Specifying a Transformation Job

To define a transformation job, you create a configuration file in XML format with sections that specify the locations of the source ATSs, including the data file, the schema in XML format, and the mapping specification (XMP file). For information, see Appendix A, "run_config.xml Reference."

Table 1 shows where to specify the parameters associated with each part of a transformation run.

**Table 1: Specifying Transformation Run Stages**

| To specify: | Fill in: | For more information, see: |
|---|---|---|
| Transformation Engine configuration settings | The **engine** section of the run configuration file | "section name="engine"" on page 37 |
| One or more source ATSs | The **atsSource** section of the run configuration file | "section name="atsSource"" on page 38 |
| Data conversion during the preprocessor phase | The **PREPROCESSOR configurationList** element of the XMP file associated with a source ATS | "Specifying Data Conversions" on page 181 of the *Modulant Balisarda Mapping Tool Guide* |
| The target ATS | The **atsTarget** section of the run configuration file | "section name="atsTarget"" on page 39 |
| Data conversion during the postprocessor phase | The **POSTROCESSOR configurationList** element of the XMP file associated with the target ATS | "Specifying Data Conversions" on page 181 of the *Modulant Balisarda Mapping Tool Guide* |

## The Life Cycle of a Queued Job

The Lifecycle Manager creates a queue of jobs as a single-threaded process in a single JVM.

Figure 2 shows the Lifecycle Manager queue and what happens when you add a new transformation run job.

**Figure 2: The Lifecycle Manager Queue**

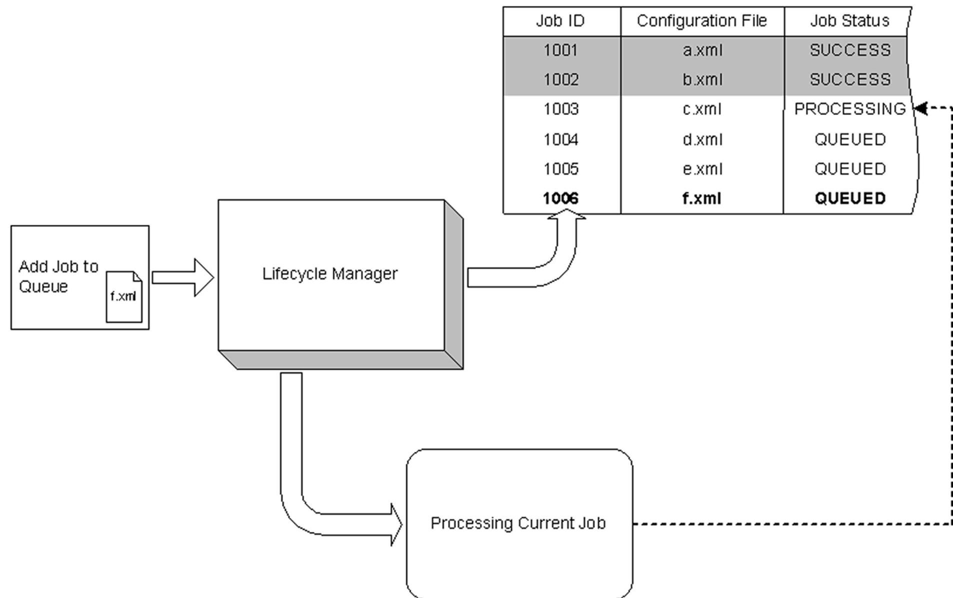
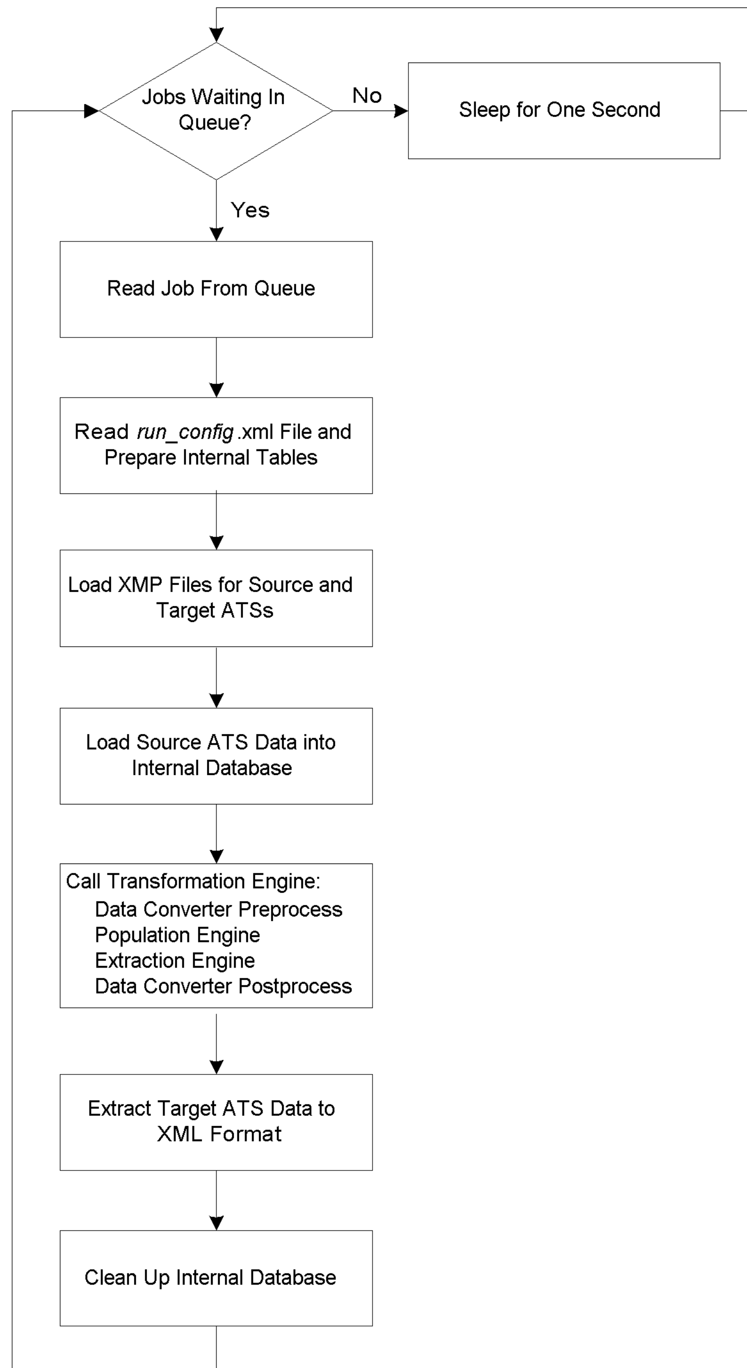



Figure 3 shows how the Lifecycle Manager identifies and processes the current job.

**Figure 3: Life Cycle of a Queued Job**



# 2 Using the Lifecycle Manager API

Using the Lifecycle Manager API, you can create a job queue from which you can perform Modulant Curtana transformation runs independently of the Modulant Curtana Analyst Tool. You specify the parameters of each transformation run in a configuration file, which the Lifecycle Manager reads.

The Lifecycle Manager writes status and error messages to the standard Modulant Curtana log file (**curtana.log**), which resides in the **logs** subdirectory of your installation.

Unlike the Analyst Tool, which can use either Access or Oracle for its internal database, the Lifecycle Manager API requires an Oracle database.

The following topics describe what you can do with the Lifecycle Manager API:

- Configuring the Modulant Curtana Transformation Engine
- Setting Up the Runtime Environment
- Invoking the Lifecycle Manager
- Adding a Job to the Queue
- Getting the Status of a Job
- Removing a Job from the Queue
- Using the Lifecycle Manager from the Command Line

**Note:** Before you can use the Lifecycle Manager API, you must prepare the internal database. To do this, you must run the Database Configuration script supplied with the Modulant Curtana platform; for instructions, see the *Installation Guide*. This script creates core tables that the Lifecycle Manager and the Transformation Engine use.

## Configuring the Modulant Curtana Transformation Engine

The Lifecycle Manager API uses two databases: one to manage the job queue (known as the *control database*), and one to store internal data for transformation runs (known as the *working database*). In order to use the Lifecycle Manager, you must specify the locations of both of these databases. These can point to the same database, or different ones.

**WARNING:** Do not add any tables of your own to the working database; the Lifecycle Manager will remove them after each transformation run.

The Lifecycle Manager uses two files to specify database connection information. These files reside in the **conf** subdirectory of your Modulant Curtana installation. You specify the information about the working databases in these files, as follows:

- **conf\config.xml**

  This file contains a section with database information for the Lifecycle Manager. You specify logical names for the internal databases. For example:

  ```
  <section name="lifecycle">
      <entry key="workDataSource">LMWork</entry>
      <entry key="controlDataSource">LMControl</entry>
  </section>
  ```

  Both of these entries are required, even if they point to the same database.

- **conf\datasources.xml**

  This file describes database connections that both the Analyst Tool and the Lifecycle Manager use. You only need to edit this file yourself if you use the Lifecycle Manager.

  This file must have an entry corresponding to each data source you list in the Lifecycle section of **config.xml**. For example:

  ```
  <section name="LMWork">
      <entry key="type">JDBC</entry>
      <entry key="driver">sun.jdbc.odbc.JdbcOdbcDriver</entry>
      <entry key="URL">jdbc:odbc:devz</entry>
      <entry key="user">user7</entry>
      <entry key="password">user7</entry>
      <entry key="dataSourceType">ORACLE</entry>
      <entry key="dataSourceName">devz</entry>
  </section>
  <section name="LMControl">
      <entry key="type">JDBC</entry>
      <entry key="driver">sun.jdbc.odbc.JdbcOdbcDriver</entry>
      <entry key="URL">jdbc:odbc:devz</entry>
  ```

```
    <entry key="user">user7</entry>
    <entry key="password">user7</entry>
</section>
```

Note that the section for the database used by the Transformation Engine has two entries (in bold) that the entry for the control database does not require. If you use the same data source for both purposes, you must include these entries.

**WARNING:** The Lifecycle Manager requires that you assign only **CONNECT** and **RESOURCE** roles to Oracle database users. Do not set any session-idle timeouts for database connections, and do not use Oracle's Multi-Threaded Server (MTS) for connection pooling.

For more information about the system configuration file settings the Lifecycle Manager expects, see Appendix B, “Configuring Database Connections.”

## Setting Up the Runtime Environment

Before you can use the Lifecycle Manager API, your Java **CLASSPATH** must include the following entries (in addition to entries referencing your custom code):

```
curtana.root\lib\classes12.jar
curtana.root\lib\xalan.jar
curtana.root\lib\xerces.jar
curtana.root\lib\ecs.jar
curtana.root\lib\jdom.jar
curtana.root\lib\jbcl.jar
curtana.root\lib\log4j.jar
curtana.root\curtana.jar
```

**Note:** The file **curtana.jar** is in the root directory of your Modulant Curtana installation, and the remaining files are in the **lib** subdirectory.

In addition, the Lifecycle Manager relies on a set of **.dll** files, which are installed in the ***curtana.root*** directory. Therefore, the ***curtana.root*** directory must also be in your **PATH** environment variable.

To set the **PATH** environment variable, use the following command:

```
set PATH=%PATH%;curtana.root
```

# Invoking the Lifecycle Manager

The Lifecycle Manager is a singleton class. Only one instance of it can be running at a time in the current JVM. The Lifecycle Manager creates and maintains a job queue in a single thread, processing one job at a time. Figure 2 on page 4 shows how the Lifecycle Manager works with the job queue.

When it starts, the Lifecycle Manager returns a reference to itself, which you can use to manage and monitor queued jobs. For more information, see "LifecycleManager Class" on page 22.

Example 1 on page 11 shows how to invoke the Lifecycle Manager and add a job to the queue. For more information about starting the Lifecycle Manager, see "getInstance()" on page 24.

The Lifecycle Manager starts a worker thread to process transformation runs. When you invoke the **getInstance()** method, the Lifecycle Manager attempts to start this worker thread if it has not already started one.

The worker thread is responsible for executing any transformation runs (jobs) waiting in the Lifecycle Manager's queue. If the queue is empty when you add a job, the worker thread begins executing the new job immediately. If there is nothing in the queue, the worker thread sleeps for one second before repolling the queue.

When you start the Lifecycle Manager, it determines whether a worker thread is active in the current Java Virtual Machine (JVM) or in another JVM. It does this by checking the control database defined in **datasources.xml**. If the Lifecycle Manager is already running in another JVM (using the same database connections), the worker thread will not start.

If the worker thread *is* started, it continues to run until you stop the current JVM.

Therefore, the examples in this chapter assume that only one JVM is running, and that the Lifecycle Manager continues polling the queue after running each example. This is because the Lifecycle Manager has started the worker thread in non-daemon mode, which keeps it running until you stop the JVM.

## Adding a Job to the Queue

After the Lifecycle Manager starts, you can add transformation run jobs to its queue.

To specify a job, you must have a run configuration file with the names and locations of the source and target ATS files. For each ATS, the configuration file must specify:

- the name of the ATS

  The Transformation Engine uses this name to identify the ATS in the internal database tables.

- the full path to the file containing the ATS data
- the full path to a DTD that describes the ATS schema
- the full path to the XMP file that contains mapping information for this ATS
- the format of the ATS data; at this time, the only valid format is XML

For details about the run configuration file format, see Appendix A, “run_config.xml Reference.”

Example 1 shows how to add a job to the Lifecycle Manager’s queue. For more information, see “addJob()” on page 23.

**Example 1: Starting the Lifecycle Manager and Adding a Job**

```
package abc.xyz;

import modulant.lifecycle.LifecycleManager;
import modulant.lifecycle.queue.QueuedJobInfo;
import modulant.lifecycle.queue.QueueException;

public class LMExample {

    public static void main(String[] args)
    throws Exception {

    String configFileLocation = args[ 0 ];

    QueuedJobInfo info = null;
    LifecycleManager lMan = LifecycleManager.getInstance();

    try {
        System.out.println( "..Invoking LifecycleManager" );
        info = lMan.addJob( configFileLocation );

        System.out.println( "..Add Job successful - Job Info:: " );
        System.out.println( info.toString() );
    }
```

```
    catch(QueueException qe) {
        String msg = qe.getMessage();
        System.out.println( "..Add Job failed:: " + msg );
        }
    }
}
```

To run this example, enter the following command:

```
$> java "-Dcurtana.root=d:\modulant\curtana" abc.xyz.LMExample
    c:\runs\run_config.xml
```

This example produces the following output:

```
..Invoking LifecycleManager
..Add Job successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>
```

If you specify an invalid run configuration file path, you get the following output:

```
$> java "-Dcurtana.root=d:\modulant\curtana" abc.xyz.LMExample
    c:\runs\run_config_bak.xml
..Invoking LifecycleManager
..Add Job failed:: Queue Exception - file
c:\runs\run_config_bak.xml does not exist
```

When you send a request the Lifecycle Manager to add a job to its queue, you specify the location of the configuration file that describes the transformation run for that job. The Lifecycle Manager adds the job to the queue and returns a **QueuedJobInfo** object with information about that job and its status. For more information, see "QueuedJobInfo Interface" on page 26.

## Getting the Status of a Job

After the Lifecycle Manager has added a job to the queue, you can request information about the job, including the status of the job, the location of the configuration file, and the time the transformation run started and finished.

You can request the status of a job in either of two ways:

- by the job ticket number (which you can get by calling **QueuedJobInfo.getID()**)
- using the **QueuedJobInfo** object you received when you added the job to the queue

Table 2 on page 29 lists the possible status values for a transformation job.

Example 2 shows how to get information about a job in the queue. For more information, see "getJobUpdate()" on page 24.

### Example 2: Getting Status Information About a Queued Job

```
package abc.xyz;

import modulant.lifecycle.LifecycleManager;
import modulant.lifecycle.queue.QueuedJobInfo;
import modulant.lifecycle.queue.QueueException;

public class LMExample {

    public static void main(String[] args)
    throws Exception {

        String configFileLocation = args[ 0 ];

        QueuedJobInfo info = null;
        LifecycleManager lMan = LifecycleManager.getInstance();

        try {
            System.out.println( "..Invoking LifecycleManager" );
            info = lMan.addJob( configFileLocation );

            System.out.println( "..Add Job successful -
                Job Info:: " );
            System.out.println( info.toString() );

            int id = info.getID();

            QueuedJobInfo info2 = lMan.getJobUpdate( id );
            System.out.println( "\n..Getting Job update successful
                - Job Info:: " );
            System.out.println( info2.toString() );

            QueuedJobInfo info3 = lMan.getJobUpdate( info );
            System.out.println( "\n..Getting Job update successful -
                Job Info:: " );
            System.out.println( info3.toString() );
        }
        catch(QueueException qe) {
            String msg = qe.getMessage();
            System.out.println( "..Job Manipulation failed:: "
                + msg );
        }
    }
}
```

To run this example, use the following command:

```
$> java "-Dcurtana.root=d:\modulant\curtana" abc.xyz.LMExample
    c:\runs\run_config.xml
```

This command produces the following output:

```
..Invoking LifecycleManager
..Add Job successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Getting Job update successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Getting Job update successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>
```

# Removing a Job from the Queue

You can ask the Lifecycle Manager to remove a job from the queue at any time until the job has started. Once a transformation run is in progress, it cannot be removed. If you request that a running job be removed, the Lifecycle Manager throws an exception.

You can specify a job to remove in either of two ways:

- by the job ticket number (which you can get by calling **QueuedJobInfo.getID()**)
- using the **QueuedJobInfo** object you received when you added the job to the queue

Example 3 shows how to remove a job from the Lifecycle Manager's queue. For more information, see "addJob()" on page 23.

**Example 3: Removing a Transformation Job from the Queue**

```
package abc.xyz;

import modulant.lifecycle.LifecycleManager;
import modulant.lifecycle.queue.QueuedJobInfo;
import modulant.lifecycle.queue.QueueException;

public class LMExample {

   public static void main(String[] args)
   throws Exception {

      String configFileLocation = args[ 0 ];

      QueuedJobInfo info = null;
      LifecycleManager lMan = LifecycleManager.getInstance();

      try {
      System.out.println( "..Invoking LifecycleManager" );
      info = lMan.addJob( configFileLocation );

      System.out.println( "..Add Job successful - Job Info:: " );
      System.out.println( info.toString() );

      info = lMan.getJobUpdate( info );
      System.out.println( "\n..Getting Job update successful -
         Job Info:: " );
      System.out.println( info.toString() );

      System.out.println( "\n..Current Job status is:: "
         info.getStatus() );

      lMan.removeJob( info );
```

```
            System.out.println( "\n..Job removed successfully" );
        }
        catch(QueueException qe) {
            String msg = qe.getMessage();
            System.out.println( "..Job Manipulation failed:: "
                + msg );
        }
    }
}
```

To run this example, use the following command:

```
$> java "-Dcurtana.root=d:\modulant\curtana" abc.xyz.LMExample
    c:\runs\run_config.xml
```

This command produces the following output:

```
..Invoking LifecycleManager
..Add Job successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Getting Job update successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Current Job status is:: QUEUED

..Job removed successfully
```

Unless the Lifecycle Manager encounters a problem while removing the specified job, you will receive no feedback to let you know that the job has been removed. If the job is already running, the Lifecycle Manager throws a **QueueException**, and the job continues until the transformation run is complete.

If there were no other jobs in the queue, the Lifecycle Manager begins executing this one immediately, which means that you can't remove it. For example:

```
..Invoking LifecycleManager
..Add Job successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>QUEUED</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Getting Job update successful - Job Info::
<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
    <jobID>10985</jobID>
    <configFileLocation>C:\runs\run_config.xml</configFileLocation>
    <status>PROCESSING</status>
    <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
    <jobRunStartTime></jobRunStartTime>
    <jobRunEndTime></jobRunEndTime>
    <errorFileLocation></errorFileLocation>
    <statusMessage></statusMessage>
</JobInfo>

..Current Job status is:: PROCESSING
..Job manipulation failed:: Queue Exception - Job 10985 is
currently processing, and cannot be removed
```

## Using the Lifecycle Manager from the Command Line

Instead of building support for the Lifecycle Manager's queuing mechanism into a larger application, you can interact with the Lifecycle Manager directly, using a command-line tool known as the **LifecycleQueue**. This tool is available in the **modulant** package.

The **LifecycleQueue** allows end-users, such as integration architects, to add transformation jobs to the queue, obtain the current status of a queued job, or remove a job from the queue.

This section addresses the following topics:

- Running the Lifecycle Queue
- Adding a Transformation Job to the Queue
- Getting the Status of a Queued Job
- Removing a Transformation Job from the Queue

## Running the Lifecycle Queue

Use the following command to run the **LifecycleQueue**:

```
$> java "-Dcurtana.root=d:\modulant\curtana"
modulant.LifecycleQueue <command> <options>
```

If you run this command without specifying a command or any options, you get a help message describing how to use the command, shown in Example 4.

**Example 4: LifecycleQueue Usage Message**

```
******************************************************************
Welcome to Curtana!

The correct usage of this utility is one of the following:
(1) ADD Default - java "-Dcurtana.root=<curtana.root>"
modulant.LifecycleQueue ADD
(2) ADD Explicit - java "-Dcurtana.root=<curtana.root>"
modulant.LifecycleQueue ADD <config_file>
(3) STATUS Explicit - java "-Dcurtana.root=<curtana.root>"
modulant.LifecycleQueue STATUS <job_id>
(4) REMOVE Explicit - java "-Dcurtana.root=<curtana.root>"
modulant.LifecycleQueue REMOVE <job_id>

where:
 ADD - Adds a run configuration file to the work queue to be
   processed.
   Omitting <config_file> assumes the run configuration to be:
   <curtana.root>\conf\run_config.xml

STATUS - Retrieves job status from the work queue based on a Job
   ID.
   The Job ID must be specified, or no action will be taken.
   The Job ID must be an integer.

REMOVE - Removes a job from the work queue based on a Job ID.
   The Job ID must be specified, or no action will be taken.
   The Job ID must be an integer.

For additional information on the command line API, please refer to
the user manual.
******************************************************************
```

## Adding a Transformation Job to the Queue

To add a job to the queue, you specify the location of a run configuration file that contains the parameters of the transformation job. If you do not specify a configuration file name, the **LifecycleQueue** looks for a run configuration file in the default location ***curtana.root*\conf\run_config.xml**.

When you add a job to the queue, the **LifecycleQueue** prints a DOM representation of the of the queued job (a **QueuedJobInfo** object) to the console, much like the examples in the previous sections. For more information about the returned job object, see "QueuedJobInfo Interface" on page 26.

Example 5 shows the syntax you use to add a job to the queue, followed by the XML representation of the queued job.

**Example 5: Adding a Job from the Command Line**

```
$> java "-Dcurtana.root=d:\modulant\curtana"
modulant.LifecycleQueue ADD c:\runs\run_config.xml

<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
   <jobID>10985</jobID>
   <configFileLocation>C:\runs\run_config.xml</configFileLocation>
   <status>QUEUED</status>
   <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
   <jobRunStartTime></jobRunStartTime>
   <jobRunEndTime></jobRunEndTime>
   <errorFileLocation></errorFileLocation>
   <statusMessage></statusMessage>
</JobInfo>
```

# Getting the Status of a Queued Job

To request the status of a queued job, you specify the job ID. The job ID is the first element (**jobID**) of the XML representation of the job that the **LifecycleQueue** returned when you added the job.

Example 6 shows the syntax you use to get the status of a queued job. The **LifecycleQueue** returns another copy of the XML DOM object containing the job's status.

**Example 6: Getting the Status of a Queued Job**

```
$> java "-Dcurtana.root=d:\modulant\curtana"
modulant.LifecycleQueue STATUS 10985

<?xml version="1.0" encoding="UTF-8"?>
<JobInfo>
   <jobID>10985</jobID>
   <configFileLocation>C:\runs\run_config.xml</configFileLocation>
   <status>PROCESSING</status>
   <jobSubmitTime>2001.07.31.18:33:05</jobSubmitTime>
   <jobRunStartTime>2001.07.31.18:35:05</jobRunStartTime>
   <jobRunEndTime></jobRunEndTime>
   <errorFileLocation></errorFileLocation>
   <statusMessage></statusMessage>
</JobInfo>
```

## Removing a Transformation Job from the Queue

To remove a job from the queue, you specify the job ID. The job ID is the first element (**jobID**) of the XML representation of the job that the **LifecycleQueue** returned when you added the job.

Example 7 shows the syntax you use to get the status of a queued job. After removing the job, the **LifecycleQueue** prints a confirmation message.

**Note:** If the transformation job has already started, you cannot remove it from the queue.

#### Example 7: Removing a Job from the Queue

```
$> java "-Dcurtana.root=d:\modulant\curtana"
modulant.LifecycleQueue REMOVE 10985

QueuedJobInfo 10985 removed successfully.
```

# 3 Lifecycle Manager API Reference

The Lifecycle Manager API lets you perform transformation runs separately from the Modulant Curtana Analyst Tool. Using this API, you can define jobs that execute transformation runs and queue them to run independently. You can use this API in standalone Java programs, or inside of existing EJBs (Enterprise JavaBeans) or servlets in a single JVM.

This API resides in the package **modulant.lifecycle**, which contains the singleton class **LifecycleManager** and the interface **QueuedJobInfo**:

- The **LifecycleManager** class creates a job queue in a single thread, and provides methods that let you control and monitor the jobs in the queue. Each job is a single transformation run. For details, see “LifecycleManager Class” on page 22.
- The **QueuedJobInfo** interface describes an individual jobs in the job queue. It contains methods that provide status information about the job. For details, see “QueuedJobInfo Interface” on page 26.

In addition, you can use the Lifecycle Manager functions directly from the command line, using the **LifecycleQueue** class. For details, see “LifecycleQueue Class” on page 31.

The Lifecycle Manager API uses a configuration file for each job that contains information about the transformation run, including the names and locations of the source and target ATSs and the associated XMP files. For information, see Appendix A, “run_config.xml Reference.”

Figure 3 on page 5 shows the flow of the job queue and the life cycle of a job as it is being processed.

## LifecycleManager Class

Singleton class that manages a queue for the invocation and running of one or more transformation runs.

### Syntax

```
package modulant.lifecycle;

import modulant.lifecycle.queue.QueueException;
import modulant.lifecycle.queue.QueuedJobInfo;

public class LifecycleManager
extends java.lang.Object
{
   public QueuedJobInfo addJob(java.lang.String configFilePath)
   public static final LifecycleManager getInstance()
   public QueuedJobInfo getJobUpdate(int jobID)
   public QueuedJobInfo getJobUpdate(QueuedJobInfo info)
   public void removeJob(int jobID)
   public void removeJob(QueuedJobInfo info)
}
```

### Description

The **LifecycleManager** class manages a queue of one or more jobs that define transformation runs. Once you add a job to the queue, the **LifecycleManager** handles the entire process, as shown in Figure 3 on page 5.

The **getInstance()** method starts the **LifecycleManager**; it acts as a singleton within the current JVM. The **LifecycleManager** is responsible for queueing jobs (in the form of a run configuration file) in the work queue, reporting the status of a particular job when polled, and removing a job when requested.

The **LifecycleManager** spawns a worker thread, which queries the work queue and executes queued jobs (that is, performs the defined transformation run), one after the other, in the order they appear in the queue.

# addJob()

```
public QueuedJobInfo addJob(java.lang.String configFilePath)
    throws QueueException
```

## Description

Adds a job to the work queue, using the information in the run configuration file.

## Parameter

| | |
|---|---|
| ***configFilePath*** | The path to the location of the **run_config.xml** file, including the file name, that defines the parameters of the transformation run associated with this job. |

## Return Value

Returns a **QueuedJobInfo** object with information about the job's position in the queue and its status.

## Exception

Throws **QueueException** if an exception is encountered while adding the job to the queue; for example, if the specified configuration file does not exist.

## Example

Example 1 on page 11 shows how to use this method to add a job to the Lifecycle Manager's queue.

## See Also

- "removeJob()" on page 25
- "QueuedJobInfo Interface" on page 26
- "run_config.xml Reference" on page 35

## getInstance()

```
public static final LifecycleManager getInstance()
```

### Description

Returns a reference to a singleton **LifecycleManager** object. If a **LifecycleManager** object already exists in the JVM memory space, this method returns a reference to that object; otherwise, it creates a new one and returns a reference to the new object.

### Return Value

Returns a singleton instance of a **LifecycleManager** object.

### Example

Example 1 on page 11 shows how to use this method to start the Lifecycle Manager.

## getJobUpdate()

```
public QueuedJobInfo getJobUpdate(int jobID)
public QueuedJobInfo getJobUpdate(QueuedJobInfo info)
```

### Description

Queries the work queue to determine the current status **QUEUED**, **SUCCESS**, **PROCESSING**, etc.) of a queued job. This method has two versions, both of which return status information about a queued job. The difference is how you specify the job:

- You can request status information about a job using its job ticket number. To determine the ticket number, call **QueuedJobInfo.getID()**.
- You can request status information about a job using the **QueuedJobInfo** object you received when you added the job to the queue.

### Parameters

Based on which version of this method you use, specify one of the following parameters:

| | |
|---|---|
| ***jobID*** | The ticket number of a job whose status you want. |
| ***info*** | A job object that describes the job whose status you want. |

### Return Value

Returns a **QueuedJobInfo** object with updated status information about the job.

### Exception

Both versions of this method throw **QueueException** if an exception is encountered while checking the job status; for example, if the job cannot be found.

### Example

Example 2 on page 13 shows both ways of using this method: by specifying a job ID, and by specifying a **QueuedJobInfo** object.

### See Also


- "QueuedJobInfo Interface" on page 26
- "getID()" on page 27


## removeJob()

```
public void removeJob (int jobID)
    throws QueueException

public void removeJob (QueuedJobInfo info)
    throws QueueException
```

### Description

This method has two versions, both of which remove a job from the work queue. The difference is how you specify the job to remove:

- You can remove a job using its job ticket number. To determine the ticket number, call **QueuedJobInfo.getID()**.
- You can remove a job using the **QueuedJobInfo** object you received when you added the job to the queue.

### Parameters

Based on which version of this method you use, specify one of the following parameters:

| | |
|---|---|
| ***jobID*** | The ticket number of a job to remove from the queue. |
| ***info*** | A job object that describes the job to remove from the queue. |

### Exception

Both versions of this method throw **QueueException** if an exception is encountered while removing the specified job from the queue; for example, if the requested job is still being executed.

### Example

Example 3 on page 15 shows how to use this method to remove a job from the Lifecycle Manager's queue.

### See Also


- "addJob()" on page 23
- "QueuedJobInfo Interface" on page 26
- "getID()" on page 27


## QueuedJobInfo Interface

Interface that encapsulates information about a queued job that describes a Modulant Curtana transformation run.

### Syntax

```
package modulant.lifecycle.queue;

public interface QueuedJobInfo
extends modulant.lifecycle.queue.QueuedJobStatus
{
   public java.lang.String getConfigFilePath()
   public java.lang.String getErrorFilePath()
   public int getID()
   public java.util.Date getRunEndTime()
   public java.util.Date getRunStartTime()
   public java.lang.String getStatus()
   public int getStatusCode()
   public java.lang.String getStatusMessage()
   public java.util.Date getSubmitTime()
}
```

### Description

The **QueuedJobInfo** interface provides methods that return information about the status of a queued job and the configuration file that defines the associated transformation run. As a job progresses through the transformation run, its status can take the values listed in Table 2 on page 29.

## getConfigFilePath()

```
public java.lang.String getConfigFilePath()
```

### Description

Returns the location of the XML configuration file that defines the transformation run associated with the current job.

### Return Value

A string that contains the full path name to the configuration file that defines this transformation run.

### See Also

- "run_config.xml Reference" on page 35

## getErrorFilePath()

```
public java.lang.String getErrorFilePath()
```

### Description

Returns the location of a file containing error messages if an error occurs during the transformation run.

### Return Value

Returns a string that contains the full path name to the error log file. Returns an empty string if no errors were encountered.

## getID()

```
public int getID()
```

### Description

Returns a ticket number that indicates the position of the current job in the job queue.

### Return Value

Returns an integer value that represents the job ID number.

### Example

Example 2 on page 13 shows how to use this method to get the job ID of a queued transformation run job.

## getRunEndTime()

```
public java.util.Date getRunEndTime()
```

### Description

Returns the time that a successful transformation run ended. While a job is still in progress, this method returns a null value.

### Return Value

Returns a date value containing the time the transformation run finished, or a null value if the run has not ended.

### See Also


- "getRunStartTime()" on page 28
- "getSubmitTime()" on page 31


## getRunStartTime()

```
public java.util.Date getRunStartTime(
```

### Description

Returns the time that the transformation run associated with a job started. While the job is in the queue waiting to start, this method returns a null value.

### Return Value

Returns a date value containing the time the transformation run started, or a null value if the job has not yet started.

### See Also


- "getRunEndTime()" on page 28
- "getSubmitTime()" on page 31

# getStatus()

```
public java.lang.String getStatus()
```

## Description

Returns the status of the current job, indicating whether any exceptions have occurred, and where the job is in the process of the transformation run.

## Return Value

Returns a status value, using one of the values listed in Table 2.

**Table 2: Queued Job Status Values**

| Status | Code | Indicates |
|---|---|---|
| **QUEUED** | 0 | job has been queued, but has not been executed yet |
| **PROCESSING** | 1 | job is currently being executed |
| **SUCCESS** | 2 | job completed successfully |
| **NO_JOB** | 12 | no job found for the given job ID |
| **EXCEPTION_ATS_LOAD** | 4 | job failed while loading data from a source ATS |
| **EXCEPTION_XMP_LOAD** | 6 | job failed while loading XMP mappings |
| **EXCEPTION_PRE_PROCESS** | 7 | job failed during the preprocessing phase of the Modulant Curtana Data Converter |
| **EXCEPTION_POPULATION** | 9 | job failed while populating data to the Abstract Conceptual Model |
| **EXCEPTION_EXTRACTION** | 10 | job failed while extracting data from the Abstract Conceptual Model |
| **EXCEPTION_POST_PROCESS** | 8 | job failed during the postprocessing phase of the Modulant Curtana Data Converter |
| **EXCEPTION_ATS_EXTRACT** | 5 | job failed while extracting data to a target ATS |
| **EXCEPTION_SCHEMA_REBUILD** | 11 | job failed during database cleanup |
| **EXCEPTION_UNKNOWN** | 3 | some unknown exception has occurred |

### Example

Example 3 on page 15 shows how to use this method to get the status of a queued transformation run job.

### See Also


- "getStatusCode()" on page 30
- "getStatusMessage()" on page 30


## getStatusCode()

```
public int getStatusCode()
```

### Description

Returns the a numeric value showing the status of the current job, including whether any exceptions have occurred, and where the job is in the process of the transformation run.

### Return Value

Returns an integer value representing the status code, using one of the values listed in Table 2 on page 29.

### See Also


- "getStatus()" on page 29
- "getStatusMessage()" on page 30


## getStatusMessage()

```
public java.lang.String getStatusMessage()
```

### Description

Returns any messages associated the status of the current job, including messages generated by exceptions that have occurred.

### Return Value

Returns a text string that contains status messages associated with the current transformation run.

### See Also


- "getStatus()" on page 29
- "getStatusCode()" on page 30


## getSubmitTime()

```
public java.util.Date getSubmitTime()
```

Returns the time that the transformation run associated with a job was added to the job queue.

### Return Value

Returns a date value containing the time the job was queued.

### See Also


- "getRunStartTime()" on page 28
- "getRunEndTime()" on page 28


# LifecycleQueue Class

Class that gives you direct access to the Lifecycle Manager's queue from the command line.

### Syntax

```
package modulant;
import modulant.lifecycle.LifecycleManager;
public class LifecycleQueue
extends java.lang.Object
{
    public static void main(String[] args)
}
```

### Command-Line Syntax

```
java "-Dcurtana.root=curtana.root" modulant.LifecycleQueue ADD
java "-Dcurtana.root=curtana.root" modulant.LifecycleQueue ADD
   config_file

java "-Dcurtana.root=curtana.root" modulant.LifecycleQueue REMOVE
   job_id

java "-Dcurtana.root=curtana.root" modulant.LifecycleQueue STATUS
   job_id
```

## Description

If you do not want to embed the functionality of the Lifecycle Manager into an application, you can use the **LifecycleQueue** class to run transformation jobs directly from the command line.

You can use the **LifecycleQueue** class to manage transformation runs in the Lifecycle Manager's queue. Using the options available for this class, you can:

- add jobs to the queue
- view the status of a queued job
- remove queued jobs that have not been started

To use the **LifecycleQueue**, enter the following command:

```
$> java "-Dcurtana.root=d:\modulant\curtana"
modulant.LifecycleQueue <command> <options>
```

"Options" on page 32 lists the available command-line options. If you run this command without specifying any options, you get a help message describing the syntax of the command. Example 4 on page 18 shows the **LifecycleQueue**'s usage message.

## Options

The **LifecycleQueue** command recognizes the following options:

**ADD *config_file***

Adds a job to the queue, using the information in the specified run configuration file. For information about the syntax of run configuration files, see Appendix A, "run_config.xml Reference."

The **LifecycleQueue** returns an XML DOM object with a description of the queued job.

**ADD**

If you use the **ADD** option without specifying a configuration file name, the **LifecycleQueue** looks for a run configuration file in the default location ***curtana.root*\conf\run_config.xml**.

**STATUS *job_id***

Returns the status of a queued job as an XML DOM object. To specify the job, you provide the job ID from the DOM object you got from the **LifecycleQueue** when you added the job.

For more information about this job object, see "QueuedJobInfo Interface" on page 26.

**REMOVE *job_id***

Removes a job from the queue. To specify the job, you provide the job ID from the DOM object you got from the **LifecycleQueue** when you added the job.

## Examples

- Example 5 on page 19 shows the syntax for adding a job to the queue, and the command-line output that results.
- Example 6 on page 19 shows the syntax for requesting a job's status, and the status information that the **LifecycleQueue** returns.
- Example 7 on page 20 shows the syntax for removing a job from the queue, followed by the **LifecycleQueue**'s confirmation.

# A run_config.xml Reference

The*run_config*.xml file defines the parameters of a transformation run. Youcreate this file yourself using an XML editor, such as XMLSpy, or any text editor.

Example 8 shows a sample run configuration file. This appendix explains each of the sections in the sample file.

**Example 8: Configuration File Syntax**

```
<?xml version = "1.0"?>
<!-- sample run config file -->
<configuration>
    <section name="engine">
        <entry key="integrationSchemaFile">
            e:\runs\acme\schema\acme.exp
        </entry>
        <entry key="directTranslation">false</entry>
        <entry key="forceUnique">true</entry>
        <entry key="enforceATSKeys">true</entry>
    </section>
    <section name="atsSource">
        <section name="atsSource1">
            <entry key="name">acme_1_in</entry>
            <entry key="filePath">
                e:\runs\acme\ats\acme_1_in.xml
            </entry>
            <entry key="dtdPath">
                e:\runs\acme\ats\acme_1_in.dtd
            </entry>
            <entry key="xmpPath">
                e:\runs\acme\xmp\acmeXMP_1.xml
            </entry>
            <entry key="atsSchemaDefinition">
                e:\runs\acme\xmp\acmeXMP_schema_1.xml
            </entry>
            <entry key="format">xml</entry>
            <entry key="createKeys">false</entry>
        </section>
```

```
        <section name="atsSource2">
            <entry key="name">acme_2_in</entry>
            <entry key="filePath">
                e:\runs\acme\ats\acme_2_in.xml
            </entry>
            <entry key="dtdPath">
                e:\runs\acme\ats\acme_2_in.dtd
            </entry>
            <entry key="xmpPath">
                e:\runs\acme\xmp\acmeXMP_2.xml
            </entry>
            <entry key="atsSchemaDefinition">
                e:\runs\acme\xmp\acmeXMP_schema_2.dtd
            </entry>
            <entry key="format">xml</entry>
            <entry key="createKeys">false</entry>
        </section>
    </section>
    <section name="atsTarget">
        <section name="atsTarget1">
            <entry key="name">acme_out</entry>
            <entry key="filePath">
                e:\runs\acme\ats\acme_out.xml
            </entry>
            <entry key="dtdPath">
                e:\runs\acme\ats\acme_out.dtd
            </entry>
            <entry key="xmpPath">
                e:\runs\acme\xmp\acmeXMP_out.xml
            </entry>
            <entry key="atsSchemaDefinition">
                e:\runs\acme\xmp\acmeXMP_schema_out.dtd
            </entry>
            <entry key="format">xml</entry>
        </section>
    </section>
</configuration>
```

## Description

The structure of the configuration file is not complex. The file consists a top-level **configuration** element with nested **section** elements, each of which has a set of **entry** elements. You specify each section using a **name** attribute.

For each **section**, the Lifecycle Manager expects a specific set of entry elements, which you specify using **key** attributes.

Each section of the configuration file triggers a stage in the transformation run. The expected sections are:

- section name="engine"

  This section defines overall parameters that the Lifecycle Manager uses to control the transformation run, including the location of the Abstract Conceptual Model schema file.

- section name="atsSource"

  This section defines each of the source ATSs for the transformation run, including the location of the ATS data file, the DTD that defines the ATS schema, and the XMP file that contains the mapping specification for this ATS. You specify each source ATS in a nested **section** element.

- section name="atsTarget"

  This section defines each target ATS for the transformation run, including the location of the output file into which to extract the ATS data, the DTD that defines the ATS schema, and the XMP file that contains the mapping specification for this ATS. You specify the target ATS in a nested **section** element.

## section name="engine"

The Engine section defines the configuration parameters required by the Lifecycle Manager. The entries in this section parallel the configuration information you specify in the **Configuration** dialog box of the Modulant Curtana Analyst Tool.

Table 3 lists the **entry** elements and the key attributes included in this section.

**Table 3: The Engine Section of the Configuration File**

| key Attribute | Description |
|---|---|
| **entry key="integrationSchemaFile"** | The full path to the location of the Abstract Conceptual Model schema file. The Transformation Engine expects this file to be in EXPRESS format.<br>This entry is optional; if you leave it out, the Lifecycle Manager uses the schema file listed in the system configuration file **conf\config.xml**. |

**Table 3: The Engine Section of the Configuration File (Continued)**

| key Attribute | Description |
|---|---|
| **entry key="directTranslation"** | Determines whether to perform a direct translation on the source ATS data (in which the structure of the ATS schema is considered as the integration schema), bypassing the stage of populating the Abstract Conceptual Model.<br>Specify **true** or **false**. The default is **false**. |
| **entry key="forceUnique"** | If set to **true**, the Transformation Engine enforces the uniqueness rules (that is, the key attribute constraints) defined in the Abstract Conceptual Model while populating source ATS data to the Abstract Conceptual Model tables in the internal database.<br>Specify **true** or **false**. The default is **true**. |
| **key="enforceATSKeys"** | If set to true, the Transformation Engine enforces primary keys to avoid duplicate rows when extracting target ATS data from the Abstract Conceptual Model.<br>Specify **true** or **false**. The default is **true**. |

## section name="atsSource"

The ATS Source section defines parameters for the Population Engine. Define a separate **section** subelement within this section for each source ATS that is part of the transformation run. Make sure the names of the subsections are unique.

If any of your source ATSs require data conversion before running the Population Engine, ensure that the XMP file you specify in this section contains the necessary **dataConverter** elements in a **PREPROCESSOR** list. For more information, see Chapter 9, “Using the Modulant Curtana Data Converter,” in the *Modulant Balisarda Mapping Tool Guide*.

Table 4 lists the **entry** elements and the key attributes included in this section.

**Table 4: The ATS Source Section of the Configuration File**

| key Attribute | Description |
|---|---|
| **entry key="name"** | The name of the ATS. This is the same name you would enter in the **ATS Registry** of the Modulant Curtana Analyst Tool. |
| **entry key="filePath"** | The full path to the file containing the ATS data for this source ATS. |
| **entry key="dtdPath"** | The full path to the DTD that describes the ATS schema for this source ATS. Use FirstSTEP EXML to generate this DTD. |
| **entry key="xmpPath"** | The full path to the XMP file containing the mapping specification for this source ATS. Use the Mapping Tool to export the mapping information to an XMP file. |
| **entry key="format"** | The format of the ATS data for this source ATS. The default value is **xml**.<br><br>**Note:** For this version of the Modulant Curtana platform, the only valid value for this attribute is **xml**. |
| **entry key="createKeys"** | If set to **true**, creates primary keys as specified by the data in the XML file.<br><br>Specify **true** or **false**. The default is **false**. |

## section name="atsTarget"

The ATS Target section defines parameters for the Extraction Engine. Define a **section** subelement within this section for the target ATS for the transformation run.

If your target ATS requires data conversion after running the Extraction Engine, ensure that the XMP file you specify in this section contains the necessary **dataConverter** elements in a **POSTPROCESSOR** list. For more information, see Chapter 9, “Using the Modulant Curtana Data Converter,” in the *Modulant Balisarda Mapping Tool Guide*.

Table 5 lists the **entry** elements and the key attributes included in this section.

**Table 5: The ATS Target Section of the Configuration File**

| key Attribute | Description |
|---|---|
| **entry key="name"** | The name of the ATS. This is the same name you would enter in the **ATS Registry** of the Modulant Curtana Analyst Tool. |
| **entry key="filePath"** | The full path to the file containing the ATS data for this target ATS. |
| **entry key="dtdPath"** | The full path to the DTD that describes the ATS schema for this target ATS. Use FirstSTEP EXML to generate this DTD. |
| **entry key="xmpPath"** | The full path to the XMP file containing the mapping specification for this target ATS. Use the Mapping Tool to export the mapping information to an XMP file. |
| **entry key="atsSchemaDefinition"** | |
| **entry key="format"** | The format of the ATS data for this source ATS. The default value is **xml**.<br>**Note:** For this version of the Modulant Curtana platform, the only valid value for this attribute is **xml**. |

# B Configuring Database Connections

The Modulant Curtana Interoperability Platform uses two files to specify database connection information: **config.xml** and **datasources.xml**. These files reside in the **conf** subdirectory of your Modulant Curtana installation.

- **config.xml** contains general system information for all of the Modulant Curtana platform. It contains a section specific to the Lifecycle Manager that lists names of two data sources: one for internal processing by the Modulant Curtana Transformation Engine, and one for handling the job queue. These data sources can point to the same database, or to different ones.

  For more information, see "Editing config.xml" on page 43.

- **datasources.xml** defines the connection parameters for data sources that the Modulant Curtana platform uses. The Lifecycle Manager expects an entry in this file for each name you list in the **lifecycle** section of **config.xml**.

  For more information, see "Editing datasources.xml" on page 43.

The Lifecycle Manager API uses two databases: one to manage the job queue (known as the *control database*), and one to store internal data for transformation runs (known as the *working database*).

- The control database has a static structure. The Lifecycle Manager does not create or destroy any tables in this database as it runs. The control database maintains a relatively small tablespace. Using a separate database for the job queue provides a logical separation from the dynamic data created during transformation runs.
- The working database is volatile. This database can potentially require a significant amount of tablespace. During a transformation run, the Transformation Engine creates a large number of temporary tables to support population and extraction of ATS data. When a transformation run ends, the Lifecycle Manager removes any non-core Modulant Curtana tables from the working database.

**WARNING:** Do not add any tables of your own to the working database; the Lifecycle Manager will remove them after each transformation run.

If you use the Lifecycle Manager API, you must edit both of the configuration files to specify information about the Lifecycle Manager's databases. This appendix shows examples of the necessary entries in each of these files, and explains each of the sections.

**WARNING:** The Lifecycle Manager requires that you assign only **CONNECT** and **RESOURCE** roles to Oracle database users. Do not set any session idle timeouts for database connections, and do not use Oracle's Multi-Threaded Server (MTS) for connection pooling.

Example 9 shows sample database configuration entries. The appendix explains each of the sections in the sample files.

**Example 9: Database Connection File Syntax**

- In **conf\config.xml**:

```
<section name="lifecycle">
    <entry key="workDataSource">LMWork</entry>
    <entry key="controlDataSource">LMControl</entry>
</section>
```

- In **conf\datasources.xml**:

```
<section name="LMWork">
    <entry key="type">JDBC</entry>
    <entry key="driver">sun.jdbc.odbc.JdbcOdbcDriver</entry>
    <entry key="URL">jdbc:odbc:devz</entry>
    <entry key="user">user7</entry>
    <entry key="password">user7</entry>
    <entry key="dataSourceType">ORACLE</entry>
    <entry key="dataSourceName">devz</entry>
</section>
<section name="LMControl">
    <entry key="type">JDBC</entry>
    <entry key="driver">sun.jdbc.odbc.JdbcOdbcDriver</entry>
    <entry key="URL">jdbc:odbc:devz</entry>
    <entry key="user">user7</entry>
    <entry key="password">user7</entry>
</section>
```

## Description

The master configuration file, **config.xml**, contains settings for various components of the Modulant Curtana platform. If you use the Lifecycle Manager, you only need to be concerned with the **lifecycle** section of this file. In this section, you specify logical names of the two data sources that the Lifecycle Manager uses.

The database connection configuration file, **datasources.xml**, contains a section for each data source used by any of the components of the Modulant Curtana platform. If you use separate data sources for the Transformation Engine's internal database and for the Lifecycle Manager's queue handler database, you need two entries. If you use the same data source for both, you only need one entry.

# Editing config.xml

If you use the Lifecycle Manager, you must specify the logical name of two data sources, one for the Transformation Engine to use for internal processing and one where the Lifecycle Manager processes jobs in the queue.

To specify these names, go to the **lifecycle** section of **config.xml**, and modify the entries as described in Table 6.

**Table 6: The Lifecycle Manager Section of config.xml**

| key Attribute | Description |
|---|---|
| **entry key="workDataSource"** | A logical name that represents the database used to load source ATS data, populate the Abstract Conceptual Model, and extract target ATS data during a transformation run. |
| **entry key="controlDataSource"** | A logical name that represents the database used by the Lifecycle Manager to process the queue of transformation run jobs. |

# Editing datasources.xml

After you specify logical names for the work data source and the control data source in **config.xml**, you must add connection parameters for the associated databases in **datasources.xml**. The Lifecycle Manager expects a separate entry for each logical data source name you specified in **config.xml**.

You can either edit **datasources.xml** in any text editor to create these entries, or you can use the Modulant Curtana Analyst Tool to define a database connection. Using the Analyst Tool lets you automatically encrypt the database password. If you edit the file yourself, the passwords remain in plain text. For more information, see the section on defining database connections in the *Modulant Curtana Analyst Tool Guide*.

## Specifying the Work Data Source

Table 7 lists the **entry** elements and the key attributes included in this section.

**Table 7: The Work Data Source Section of datasources.xml**

| key Attribute | Description |
| --- | --- |
| **entry key="type"** | The type of database. The default value is **JDBC**; do not change this. |
| **entry key="driver"** | The driver used by your database. The default is **sun.jdbc.odbc.JdbcOdbcDriver**, which is provided with the Modulant Curtana platform. |
| **entry key="URL"** | A pointer to your data source. The default is **jdbc:odbc:*dataSourceName***. |
| **entry key="user"** | The user name for logging on to the specified database. |
| **entry key="password"** | The password for logging on to the specified database.<br>Using the Analyst Tool to define a database connection automatically encrypts the database password. If you edit the entry yourself, the password remains in plain text. |
| **entry key="dataSourceType"** | The type database, either **ORACLE** or **ACCESS**.<br>This entry is required by the working database, but not by the control database. |
| **entry key="dataSourceName"** | The name of your data source. This value must match the ***dataSourceName*** portion of the value you specify for the **URL** key.<br>This entry is required by the working database, but not by the control database. |

## Specifying the Control Data Source

If your implementation of the Lifecycle Manager uses a different database for handling the operation of the job queue than for performing transformation runs, you must provide a separate entry for that database in **datasources.xml**.

This entry has the same form as the entry for the work data source, but does not require either the entries **dataSourceType** or **dataSourceName**.

# Index


## A

adding a job 11
- from the command line 18

**addJob()** 23
- example 11

**atsSource** section of ***run_config*.xml** 3
- attributes 38

**atsTarget** section of ***run_config*.xml** 3
- attributes 39

## C

**CLASSPATH** 9

**config.xml** file 41
- Lifecycle Manager entries 8

configuration files 8
- **config.xml** 8
- **datasources.xml** 8
- for transformation run 3
- ***run_config*.xml** 35

**configurationList** element 3

control database 41
- configuring 8

**controlDataSource** entry 43

core tables
- creation 7
- deletion 41

**createKeys** entry 39

## D

Data Converter
- postprocessing 2
- preprocessing 2

data transformation process 2

database connections 8

**dataSourceName** entry 44

**datasources.xml** file 8, 41
- control database entry 8
- work database entry 8

**dataSourceType** entry 44

**directTranslation** entry 38

**driver** entry 44

**dtdPath** entry
- in **atsSource** section 39
- in **atsTarget** section 40

## E

encrypting passwords 44

**enforceATSKeys** entry 38

**engine** section of ***run_config*.xml** 3
- attributes 37

## F

**filePath** entry
- in **atsSource** section 39
- in **atsTarget** section 40

**forceUnique** entry 38

**format** entry
- in **atsSource** section 39
- in **atsTarget** section 40

## G

**getConfigFilePath()** 27

**getErrorFilePath()** 27

**getID()** 27
- example 13

**getInstance()** 24
- example 11

**getJobUpdate()** 24
- example 13

**getRunEndTime()** 28

**getRunStartTime()** 28

**getStatus()** 29
- example 15

**getStatusCode()** 30

**getStatusMessage()** 30

**getSubmitTime()** 31

getting job status 12
- from the command line 19

## I

**integrationSchemaFile** entry 37

internal database, for Transformation Engine 8

## J

job queue 1
- adding a job 11
- database for 8
- lifecycle of a job 4
- removing a job 15
- starting from the command line 18

## L

Lifecycle Manager
- database configuration 8
- invoking 10

Lifecycle Manager API 1
- and Modulant Curtana Analyst Tool 1
- configuration file 21

lifecycle of queued job 4

**LifecycleManager** class 21
- **addJob()** 23
- **getInstance()** 24
- **getJobUpdate()** 24
- overview 22
- **removeJob()** 25

**LifecycleQueue** class 31
- introduced 21
- using 17

## M

**modulant.lifecycle** package 21

## N

**name** entry
- in **atsSource** section 39
- in **atsTarget** section 40

## P

package **modulant.lifecycle** 21

**password** entry 44

**PATH** environment variable 9

**POSTPROCESSOR** conversion list 3

**PREPROCESSOR** conversion list 3

## Q

queue, *see* job queue 8

queued job
- lifecycle 4
- status 12
- status values 29

queued jobs
- removing 15

**QueuedJobInfo** interface 21
- example 12
- **getConfigFilePath()** 27
- **getErrorFilePath()** 27
- **getID()** 27
- **getRunEndTime()** 28
- **getRunStartTime()** 28
- **getStatus()** 29
- **getStatusCode()** 30
- **getStatusMessage()** 30
- **getSubmitTime()** 31
- syntax 26

**QueueException**
- from **addJob()** 23
- from **getJobUpdate()** 25
- from **removeJob()** 26

## R

**removeJob()** 25
- example 15

removing a job 15
- from the command line 20

removing jobs
- job already running 16

run configuration file 3
- sections 3

***run_config*.xml** file 3, 35
- **atsSource** section 3, 38
- **atsTarget** section 3, 39
- **engine** section 3, 37

runtime environment 9

## S

status 12

## T

Transformation Engine
- configuration 3

transformation run 2
    defining 3
    removing 15
    specifying parameters 3, 35
    status 12
    status values 29
**type** entry 44

## U

**URL** entry 44
**user** entry 44

## W

**workDataSource** entry 43
working database 41
    configuring 8

## X

XMP file
    defining data conversions 3
**xmpPath** entry
    in **atsSource** section 39
    in **atsTarget** section 40