OpenText™ Exstream™

Configuring Connectors

Design and Production Documentation

Release 16.6.0

OpenText™ Exstream
Configuring Connectors
Rev.: 2019-Apr-30

This documentation has been created for software version 16.6.0.

It is also valid for subsequent software versions as long as no new document version is shipped with the product or is published at https://knowledge.opentext.com.

Open Text Corporation
275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677

Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Contents

Chapter 1: Dynamic Data Access and Connectors

Exstream provides a document service solution that lets you expand your company's reach to new customers and improve the customer experience by integrating your enterprise architectures with Exstream Design and Production. Integrating Exstream Design and Production and your external systems lets you produce customer communications in real time. The Dynamic Data Access (DDA) module coupled with either Exstream licensed connector modules or custom DDA routines (user-written DDA routines) lets you create on-demand documents on a per-request basis, connect third-party software with Exstream, or query package files for application data to create personalized documents based on customer input. Using the DDA module and DDA routines increases your company's ability to meet customer needs in real time and improve internal processes, such as on-demand reporting.

To communicate with components of an enterprise architecture or existing system in your company's infrastructure, Exstream relies on Application Programming Interface (API) calls configured in a DDA routine. The API must be configured to communicate with the engine so that the engine can process the commands defined in the DDA routine. You can use a licensed connector module, which is a DDA routine provided by Exstream that completes a specific business purpose, or you can provide your own custom code in a DDA routine to complete processes otherwise unavailable in Exstream. The DDA module lets you use DDA routines to circumvent batch processing in Exstream, creating a real-time solution for document creation.

This chapter discusses the following topics:

- “The Dynamic Data Access Module” below
- “Exstream Engine as a Web Service” on the next page
- “Exstream Connectors” on page 12

1.1 The Dynamic Data Access Module

The DDA module lets Exstream communicate with user-written DDA routines and/or predefined connector modules to collect and manipulate data in existing formats across your organization. When deployed together in your organization's existing infrastructure, you can use the DDA module and connectors (user-written or predefined) to accomplish the following goals:

- Access the most recent data available in your enterprise. In other words, you do not have to repackage data into flat files before you start a production run.
- Replace a driver, initialization, reference, or report file to reduce or eliminate flat file input and data retrieval.

- Transform data records to a supported Exstream format on demand before it reaches the engine or message queues.
- Import images or text dynamically for placeholder variables to provide your customers with the most recent marketing communications.
- Complete table lookups as needed and carry out other processes, such as writing data to databases or applications.
- Intercept output data records from the engine to modify the data. A record transform can manipulate data (for tasks such as reordering or encryption) before it reaches an output queue or output file.
- Intercept input data records from traditional data sources for decryption, handling data types that are not supported by Exstream, and reordering records for advanced processing.
- Allow Exstream to communicate with Web, Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), content management, archives, and other enterprise systems and architectures in a Services-Oriented Architecture (SOA).

1.2 Exstream Engine as a Web Service

The Exstream Engine as a Web Service (EWS) serves as a wrapper for the engine that allows the engine to communicate directly with web services primarily through the use of SOAP protocol.

When using EWS, remote users can submit customer driver file data so that the engine can compose output and receive output from Exstream through the Internet. For example, a person at a remote site can send a SOAP request (including a customer driver file and additional information) over HTTP to start an engine run. EWS can then return a SOAP response that contains the necessary SOAP headers and composed output from Exstream.

You must have licensed the EWS module to use EWS. EWS is not associated with the Web Services Interface module.

For more information about EWS version 2.0, see “Exstream Engine as a Web Service” on page 69.

This section discusses the following topics:

- “EWS Specifications” on the next page
- “EWS Production Process” on the next page
- “EWS Design Considerations” on page 11

1.2.1 EWS Specifications

The following table describes the supported platforms and prerequisites for EWS version 2.0. You can use the information in this table to verify that you have met the basic requirements for setting up EWS version 2.0.

Exstream EWS specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only) z/OS • z/OS 2.1 and later	• Exstream engine • EWS module Requirements for J2EE implementations: • Java version 6.0 or later (with packaged XML support) • JAX-WS-compliant web server Requirements for WebSphere and WebLogic implementations: • WebSphere version 6.1 or later (with Webservice feature pack) • WebLogic version 10.3 or later Requirements for .NET implementations: • Microsoft .NET Framework version 4.6.1 or later • Internet Information Service (IIS) • Windows Communication Foundation (WCF)

1.2.2 EWS Production Process

EWS is comprised of the Exstream engine and an engine service. The engine service creates the wrapper that lets the Exstream engine communicate with web services. The engine service performs actions to prepare and send information from a SOAP call to the engine so that it can create documents.

When EWS receives a SOAP call, the following actions are performed:

1. The engine service extracts the driver file from the SOAP request and creates files in the working directory.
2. The engine service creates a control file on demand based on information provided in the

SOAP request. The control file includes a standard set of engine switches, such as the PACKAGEFILE switch and FILEMAP switch (for the customer driver file) as well as any additional engine switches as specified by the `<EngineOptions>` element of the SOAP request.

3. The engine starts and accesses the information in the control file and the information in the package file as specified in the `<PubFiles>` element of the SOAP request.
4. The engine writes the output to the working directory. The output can then be returned in the response or written to a location on the server.

1.2.3 EWS Design Considerations

Keep in mind the following design considerations when setting up an application that uses EWS:

- The engine service and the engine should be on the same machine in order for EWS to successfully handle the request, processing, and delivery of your application.
- The person who creates the SOAP request must know the production data source name of the customer driver file included in the package file, so the data sent as part of the SOAP request can be applied to the correct file in the application. The data included in the SOAP request replaces the data of the original customer driver file within the packaged application.
- EWS allows you to send driver files as attachments in SOAP requests. You can attach a driver file in the following ways:
  - SOAP with Attachment (SWA)
  - Message Transmission Optimization Mechanism (MTOM)

  Note: When you use EWS version 2.0, JAX-WS automatically encodes data as Base64.

- The engine returns the output to the client. The output produced by the engine can be transferred in the following ways:
  - As a part of the SOAP response. The output returned is Base64 encoded.
  - As a file you specify to be placed on the server

Caution: If the customer driver file data is not included with the SOAP request, it can cause a SOAP fault, and the production environment will not produce output at run time.

1.3 Exstream Connectors

To use a connector in your enterprise architecture, you must meet certain requirements and/or specifications depending on the connector. The following sections provide a quick reference of requirements and specifications for individual connectors. You can use the following sections to determine which connector best meets your business needs:

- “Email Delivery Connector” below
- “IBM Content Manager Connector” on the next page
- “IBM WebSphere MQ Connector” on page 14
- “Java Database Connectivity Connector” on page 15
- “Java Messaging Service Connector” on page 16
- “Microsoft MQ Connector” on page 17
- “SOAP Connector” on page 18
- “Watched Directory Connector” on page 19

1.3.1 Email Delivery Connector

The Email Delivery Connector lets you communicate with customers by sending documents through email or by sending Short Message Service (SMS) text messages. A platform-independent utility, the Email Delivery module can be implemented as a DDA connector, or as a standalone executable program.

Email Delivery Connector Specifications

The following table describes the supported platforms and prerequisites for the Email Delivery Connector. You can use the information in this table to verify that you have met the minimum requirements for setting up the connector.

Email Delivery Connector specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only)	• Exstream engine • Dynamic Data Access (DDA) module • Java Enabler (available with the base installation of Exstream Design and Production) • Java Runtime Environment (JRE) version 1.4.2 or later For more information about JRE, see the Oracle Technology Network website. • Access to an SMTP-compliant server

1.3.2 IBM Content Manager Connector

The IBM Content Manager (IBMCM) Connector extracts content (such as, text, TIFF files, or other supported placeholder variable types) from the IBM Content Manager client and then passes it to the engine to populate a placeholder variable so that design pages in an Exstream design, such as marketing campaigns, can be filled with targeted communications.

IBM Content Manager Connector Specifications

The following table describes the supported platforms and prerequisites for the IBMCM Connector. You can use the information in this table to verify that you have met the basic requirements for setting up the connector.

IBM Content Manager Connector specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016	• Exstream engine • Dynamic Data Access (DDA) module • Dynamic Content Import module • Java Enabler (available with the base installation of Exstream Design and Production)

1.3.3 IBM WebSphere MQ Connector

The IBM WebSphere MQ (WSMQ) Connector lets the Exstream engine interact with IBM WebSphere MQ enterprise messaging software using user-written DDA routines. When you use the WSMQ Connector to receive and process requests, the WSMQ Connector dynamically allocates the memory necessary to process incoming requests and then passes those requests to the engine. The dynamic memory allocation that the WSMQ Connector uses lets you avoid potential engine processing errors when large data files are passed through the connector.

IBM WebSphere MQ Connector Specifications

The following table describes the supported platforms and prerequisites for the WSMQ Connector. You can use the following information in this table to verify that you have met the minimum requirements for setting up the connector.

IBM WebSphere MQ Connector specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only) z/OS • z/OS 2.1 and later	• Exstream engine • Dynamic Data Access (DDA) module • Java Enabler (available with the base installation of Exstream Design and Production) • WebSphere MQ • IBM's terse utility • At least one running queue manager

1.3.4 Java Database Connectivity Connector

The Java Database Connectivity (JDBC) Connector lets the engine communicate and interact with databases that use Java Database Connectivity. Java Database Connectivity is Java-based data access technology. The JDBC Connector communicates with a JDBC-compliant database, such as DB2, or from an Oracle stored procedure to retrieve results which are then passed to the engine for document creation and fulfillment.

Java Database Connectivity Connector Specifications

The following table describes the supported platforms and prerequisites for the JDBC Connector. You can use the following information in this table to verify that you have met the minimum requirements for setting up the connector.

Java Database Connectivity Connector specifications

Supported 32-bit platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux (SBCS only) • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only) z/OS • z/OS 2.1 and later	• Exstream engine • Dynamic Data Access (DDA) module • XML/JSON Input module (required if you are using the XML or JSON format) • Java Enabler (available with the base installation of Exstream Design and Production) • Java Runtime Environment (JRE) version 1.4.2 or later For more information about JRE, see the Oracle Technology Network website. • Supported JDBC driver For more information about supported JDBC drivers for the JDBC Connector, see "Preparing the JDBC Connector" on page 154. **Note:** If you run the JDBC Connector on the AIX 64-bit platform, you must add the following path to your LIBPATH environment variable: `<JDK>/java6_64/jre/lib/ppc64`. The `<JDK>` in the example represents the path to the directory that contains your Java Development Kit.

1.3.5 Java Messaging Service Connector

The Java Messaging Service (JMS) Connector lets the engine communicate with JMS-compliant enterprise messaging software, such as Sun Java System MQ software and IBM WebSphere MQ software, using user-written routines. The JMS Connector is part of the open J2EE platform, and is a valuable tool to use with Web applications, particularly transaction processing.

Java Messaging Service Connector Specifications

The following table describes the supported platforms and prerequisites for the JMS Connector. You can use the information in this table to verify that you have met the minimum requirements for setting up the connector.

Java Messaging Service Connector specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only) z/OS • z/OS 2.1 and later	• Exstream engine • Dynamic Data Access (DDA) module • Java Enabler (available with the base installation of Exstream Design and Production) • Java Runtime Environment (JRE) version 1.4.2 or later For more information about JRE, see the Oracle Technology Network website. • JMS-compliant messaging service (such as WebSphere Message Queue, Sun Java System Message Queue, or BEA WebLogic) • Vendor-specific JMS client class libraries • Access to the message queues for the user who runs the engine Additional HP-UX requirements: • PHSS_30965-ld(1) and linker tools cumulative patch • PHSS_30966 and linker tools cumulative patch • HP-UX JDK 1.4 version PA_RISC 2.0 Additional z/OS requirements: • IBM z/OS Persistent Reusable VM build cm142-20040917 JIT enabled: jitc.

1.3.6 Microsoft MQ Connector

The Microsoft MQ (MSMQ) Connector lets the engine communicate with the Microsoft MQ messaging service using user-written applications.

Microsoft MQ Connector Specifications

The following table describes the supported platforms and prerequisites for the MSMQ Connector. You can use the information in this table to verify that you have met the minimum requirements for setting up the connector.

Microsoft MQ Connector specifications

Supported platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016	• Exstream engine • Dynamic Data Access (DDA) module • CLR Enabler (available with the base installation of Exstream Design and Production) • Microsoft .NET Framework version 4.6.1 or later • Microsoft Messaging Service

1.3.7 SOAP Connector

The SOAP Connector lets you use web services as a source for customer data, so that the web service can populate data and images in documents. The SOAP Connector uses an auxiliary layout file to map the web service request (the information contained within the XML <body> tags). The auxiliary layout file is assigned to a report file or reference file that makes up the SOAP response received from the web service. You can map variables, including placeholder variables, to tags in the report or reference files.

In order for the SOAP connector to use web services as data sources, you must license one of the following modules:

- Exstream Engine as a Web Service (EWS)
- Web Services Interface (WSI)

SOAP Connector Specifications

The following table describes the supported platforms and prerequisites for the SOAP Connector. You can use the information in this table to verify that you have met the minimum requirements for setting up the connector.

SOAP Connector specifications

Supported 32-bit platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only)	• Exstream engine • Dynamic Data Access (DDA) module • XML/JSON Input module • Java Runtime Environment (JRE) version 1.4.2 or later For more information about JRE, see the Oracle Technology Network website. This includes the Java Virtual Machine (JVM). • Access to a SOAP-compliant messaging server • HTTP transport

1.3.8 Watched Directory Connector

The Watched Directory Connector lets you dynamically control data access and file processing by specifying the location of a specific directory, the type of files to be read, and the disposition of those files after their data has been accessed. You specify a directory location (a watched directory) to watch for data source files. The Watched Directory Connector polls a directory looking for work to process. When the directory receives a completely written file, the Watched Directory Connector reads the file and sends the appropriate information to the engine. Exstream uses the data to create the required documents and store them in an output file.

Watched Directory Connector Specifications

The following table describes the supported platforms and prerequisites for the Watched Directory Connector. You can use the information in this table to verify that you have met the minimum requirements for setting up the connector.

Watched Directory Connector specifications

Supported 32-bit platforms	Prerequisites
Windows • Microsoft Windows 7 • Microsoft Windows 8 • Microsoft Windows 8.1 • Microsoft Windows 10 • Microsoft Windows Server 2008 R2 • Microsoft Windows Server 2012 • Microsoft Windows Server 2016 Linux • SuSE version 10 (Intel) and later • Red Hat ES 6.0 (Intel) and later UNIX • IBM AIX (RISC) 6.1 and later • Oracle Solaris 10 (SPARC) and later (32-bit only) • HP-UX Itanium 11i v2 and later (64-bit only)	• Exstream engine • Dynamic Data Access (DDA) module

Chapter 2: Configuring Dynamic Data Access

The DDA module acts as the main component in connecting your enterprise architecture to the design and production environments available in Exstream. The DDA module coupled with Exstream connectors provides the conduit by which a business's enterprise systems can communicate with Exstream to use batch or on-demand production, web services, or the interactive document (Live) capabilities that Exstream offers. The DDA module uses API calls to communicate with external routines to complete specific processes for document fulfillment. A routine is a complete set of coded instructions that directs a computer to perform a series of operations. You can create a user-written routine, such as a Dynamic Link Library (DLL), to complete these processes, or use a predefined connector available in the Exstream product suite.

Configuring DDA routines to communicate with the engine is the first step to connecting your enterprise systems with Exstream. While Design Manager provides a user-interface to configure DDA routines, you can also use engine switches to configure certain DDA routine types. To switch between DDA routines either on or off line, use engine switches to configure DDA routine types.

Whether you are using a user-written DDA routine or a predefined connector, the engine requires some information to find and start communicating with the DDA routines that you use with your enterprise architecture. You provide this information to the engine with connector objects. Connector objects define properties and specify the DDA routine that Exstream will use in an engine run.

This chapter discusses the following topics:

- “Configuring Connectors in Design Manager” below
- “Configuring DDA Routines Using a Control File” on page 28
- “Creating an Initialization File” on page 30

2.1 Configuring Connectors in Design Manager

To configure a connector to communicate with the Exstream engine, you must create a connector object in Design Manager so that the engine can identify the connector, and you must customize the properties of the connector object so that the connector can complete specific processes based on the results that you want to achieve.

This section discusses the following topics:

2.1.1 Creating a Connector Object in Design Manager

Whether you use a user-written routine or a predefined connector, you must create a connector object in Design Manager so that the engine can identify the information required to complete the appropriate processes to create documents. A connector object in Design Manager identifies properties, parameters, and other characteristics of a DDA routine.

For more information about creating a connector object for an Exstream connector, see the corresponding connector chapter.

Like other objects in the Design Manager Library, a connector object can be created, updated, cloned, renamed, deleted, and (with workflow enabled) approved and rejected. Since a connector object serves as the interface between Exstream and the user-written routine, you must assign a connector object to another application object, such as a data file, placeholder variable, or an output queue. You can assign the following application objects to a connector object in Design Manager:

- Initialization files
- Driver files
- Reference files
- Report files
- Outputs
- Output Queues
- Record transforms
- Placeholder variables
- Post processors

Before you begin creating connector objects in Design Manager, you must make sure that you have the following information:

- Name and location of the routine
- Name of the user-defined function name
- Maximum buffer size
- Operating parameters

> **Note:** For custom DDA routines, you must obtain the required information from the programmer that wrote the custom DDA routine.

To create a connector object in Design Manager, you must complete the following steps:

1. In Design Manager, in the Library, right-click the **Connectors** heading and select **New Connector**.

   The **New Connector** dialog box opens.
2. In the **Name** box, enter a name.
3. In the **Description** box, enter a description (optional).
4. Click **Finish**.

   The connector object opens in the Property Panel.
5. From the **Program type** drop-down list, select **DLL(C++)**.

   The **DLL(C++)** option can be used for all programming languages.
6. In the **Dynamic Link Library** box, enter the path and name of the DDA routine, or click to go to the file.
7. In the **Function name** box, enter the name of the function for the DDA routine.

   You can enter one of the following functions:

   - `dda_connect`
   - `processRec`
   - `recordtransform`
   - `postprocessor`
8. In the **Maximum buffer size** box, enter the buffer size to accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size that the engine allows at one time. If you set the buffer size to an amount higher than `32767`, the engine processes information 32,767 bytes at a time.
9. In the **Open parameters** box, enter the path to the initialization file that contains the

parameters for the DDA routine, or enter each parameter on a single line.

For more information about using an initialization file for parameters, see "Creating an Initialization File" on page 30.

2.1.2 Assigning a Connector Object to a Data File for Data Input

You can assign a connector to a data file so that a DDA routine can supply data input. At a minimum, you must assign a customer driver file to a DDA routine so that customer information can be supplied to the engine for production. Other data files are not required, but may be useful, such as a reference file, which allows you to access customer information not included in the customer driver file.

To assign a connector object to a data file for data input, you must complete the following steps:

1. In Design Manager, from the Library, drag the data file to the Property Panel.

   For more information about data file properties, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

2. Set properties in the data file to the way that the routine interacts with the engine. For example, on the **Basic** tab, you can identify the data format in the **File format** drop-down list, which lets you specify how the engine receives data from the routine.
3. Click the **Test Data Source** tab.
4. From the **Type** drop-down list, select **Connector**.

   The **Connector** box opens.

5. Click .

   The **Select Connector** dialog box opens.

6. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the data file for test runs.
7. Click **OK**.

   The **Select Connector** dialog box closes.

8. In the **File to use in test** box, click to go to the data file.
9. Click the **Production Data Source** tab.
10. From the **Type** drop-down list, select **Connector**.
11. Click .

    The **Select Connector** dialog box opens.

12. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the data file for production runs. You can use the same connector for test and production runs, or you can select a different connector object if you use separate DDA routines for testing and final production.
13. Click **OK**.

    The **Select Connector** dialog box closes.
14. Save the data file.

2.1.3 Assigning Connector Objects to a Data File for Record Transform

You can assign a connector to a data file that points to a routine that transforms incoming data before it reaches the engine. Assigning a record transform to a data file is optional unless it is necessary for the routine to manipulate the data on a record-for-record basis or in a buffer (for tasks such as changing the record order). You can assign a record transform to the following files:

- Customer driver file
- Reference file
- Initialization file

To assign a connector object to a data file for record transform, you must complete the following steps:

1. In Design Manager, from the Library, drag the data file to the Property Panel.
2. Set properties in the data file to the way that the routine interacts with the engine. For example, on the **Basic** tab, you can identify the data format in the **File format** drop-down list, which lets you specify how the engine receives data from the routine.
3. Click the **Advanced** tab.
4. In the **Test record transform** box, click .

    The **Select Connector** dialog box opens.
5. On the **Select Connector** dialog box, highlight the connector object that you want to assign to all test runs that use this data file.
6. Click **OK**.

    The **Select Connector** dialog box closes and you return to the **Advanced** tab.
7. In the **Production record transform** box, click .

The **Select Connector** dialog box opens.

8. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the data file for production runs. You can use the same connector for test and production runs, or you can select a different connector object if you use separate DDA routines for testing and final production.
9. Click **OK**.

   The **Select Connector** dialog box closes.
10. Save the data file.

2.1.4 Assigning a Connector Object to a Placeholder Variable

You can assign a connector to a placeholder variable to pass information from a DDA routine to the engine. Assigning a record transform to a placeholder variable is optional unless you map a placeholder variable to a customer driver or reference file. The customer driver file or reference file must be associated with the connector object that points to the routine.

To assign a connector object to a placeholder variable, you must complete the following steps:

1. In Design Manager, from the Library, drag the placeholder variable to the Property Panel.

   For more information about placeholder variables, see *Importing External Content* in the Exstream Design and Production documentation.
2. Click the **Placeholder** tab.
3. In the **Test Import connector** box, click 🔌.

   The **Select Connector** dialog box opens.
4. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the variable during test runs.
5. Click **OK**.

   The **Select Connector** dialog box closes.
6. In the **Production Import connector** box, click 🔌.

   The **Select Connector** dialog box opens.
7. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the data file for production runs. You can use the same connector for test and production runs, or you can select a different connector object if you use separate DDA routines for testing and final production.

8. Click **OK**.

   The **Select Connector** dialog box closes.

9. Save the variable.

2.1.5 Assigning a Connector Object to an Output

You can assign a connector to an output object so a DDA output file can perform post processing to manipulate output before it reaches the print stream. Assigning a connector object to an output is optional unless you want your routine to buffer, encrypt, and manipulate in other ways engine-produced output in manageable pieces and return manipulated data back to the engine.

To assign a connector object to an output, you must complete the following steps:

1. In Design Manager, from the Library, drag the output object to the Property Panel.
2. Click the **Advanced** tab.
3. In the **Test post processor** box, click 🔌.

   The **Select Connector** dialog box opens.

4. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the output during test runs.
5. Click **OK**.

   The **Select Connector** dialog box closes.

6. In the **Production post processor** box, click 🔌.

   The **Select Connector** dialog box opens.

7. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the output object for production runs. You can use the same connector for test and production runs, or you can select a different connector object if you use separate DDA routines for testing and final production.
8. Click **OK**.

   The **Select Connector** dialog box closes.

9. Save the output.

2.1.6 Assigning a Connector Object to an Output Queue

You can assign a connector object to an output queue so the DDA routine can route output from the engine. You must also assign a connector object to an output queue so the engine understands what type of output to produce. This type of routine generally routes output as a whole unit to a predefined connector (such as WSMQ Connector). In contrast to assigning DDA routines on an output object, a "routing" DDA routine on an output queue does not process data in chunks or return output to the engine.

To assign a connector object to an output queue, you must complete the following steps:

1. In Design Manager, from the Library, drag the output queue to the Property Panel.
2. On the **Basic** tab, click in the **Test output connector** box.

   The **Select Connector** dialog box opens.
3. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the output queue during test runs.
4. Click **OK**.

   You return to the Property Panel.
5. Next to the **Production output connector** box, click .

   The **Select Connector** dialog box opens.
6. On the **Select Connector** dialog box, highlight the connector object that you want to assign to the data file for production runs. You can use the same connector for test and production runs, or you can select a different connector object if you use separate DDA routines for testing and final production.
7. Click **OK**.

   The **Select Connector** dialog box closes.
8. Save the output queue.

2.2 Configuring DDA Routines Using a Control File

After you have added connector objects in Design Manager, you do not have to continue reopening Design Manager to make adjustments and changes to the properties of the DDA routine that you defined. A quick and efficient alternative to manipulating connector object properties in Design Manager is to use a control file that contains engine switches specific to the DDA module and the connectors that you are using in test and production.

In some cases, it is easier to configure a DDA routine with an engine switch than with the Design Manager interface. For example, suppose you use connectors that use multiple DDA routines to complete processes. Using engine switches to manage multiple DDA routines lets you enable or disable a DDA routine more efficiently than navigating through the Design Manager interface. The ability to enable or disable a DDA routine can reduce errors at run time. Additionally, you can use engine switches to dynamically change parameters and connector object properties at run time. Using engine switches to dynamically change parameters and connector object properties lets you change connector object properties defined in Design Manager without having to repackage the application. Repackaging is a time-consuming task that can often lead to production downtime because you have to reopen Design Manager to complete packaging tasks.

You can configure the following application objects as DDA routines using an engine switch:

- Initialization files
- Driver files
- Reference files
- Report files
- Outputs
- Output queues
- Data files
- Placeholder variables

To use a control file to configure DDA routines, you must complete the following steps:

1. Open your control file in a text editor.
2. Add the following engine switches to your control file as needed for your DDA routine.

Engine switch	Description

CONNECTORMAP	The CONNECTORMAP engine switch dynamically changes connector object properties at run time without repackaging the application.
	Syntax:
	`-CONNECTORMAP=[name of DLL],DLL,`
	`[file path location of DDA routine]`
	For example:
	`-CONNECTORMAP=customerfileDDA,DLL,`
	`C:\Program Files\OpenText\Exstream\`
	`Exstream <version>\javaenabler.dll,dda_connect,32767,`
	`INIFILE=C:\exdemo\Connector\demo\input\`
	`iniFiles\MessageQueueIn.ini`

Engine switch	Description

DDAFILEMAP	The DDAFILEMAP engine switch changes the data file target for a DDA routine.
	Syntax:
	`–DDAFILEMAP=[name of driver file],[program type], [name of DLL],[file path location of DDA routine]`
	For example:
	`–DDAFILEMAP=customerfile.dat, C:\Program Files\OpenText\Exstream\ Exstream <version>\JavaEnabler.dll,dda_connect,32767, INIFILE=C:\Program Files\OpenText\Exstream\ Exstream <version>\dda\MessageQueueOut.ini`

DDAOUTPUT	The DDAOUTPUT engine switch specifies a DDA routine for receiving the buffered output, with one output write per record.
	Syntax:
	`–DDAOUTPUT=[name of dll],[file path location]`
	For example:
	`–DDAOUTPUT=OUTPDF,C:\exdemo\Connector\out\WSMQ_Output.pdf`

DDAOUTPUTFILE	The DDAOUTPUTFILE specifies a location to place temporary output files.
	Syntax:
	`-DDAOUTPUTFILE=[file path location for custom directory]`
	For example:
	`-DDAOUTPUTFILE=C:\WSMQ Output\`

3. Save the control file.

2.3 Creating an Initialization File

Whether you use a user-written DDA routine, or a predefined connector, you must create an initialization file to contain your parameters and reference the initialization file in Design Manager. An initialization file contains operational details using both required and optional parameters. Parameters allow you to specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify the name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager instead of listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

To create an initialization file for connector parameters, you must complete the following steps:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

The following example shows a basic initialization file that can be used with a user-written DDA routine or predefined connector:

```
QUEUEMGR=RTQMGR
INPUTQUEUE=QUEUE.1
OUTPUTQUEUE=QUEUE.2
WAITINTERVAL=2
MAXINACTIVITYCOUNT=3
TRACE=L
INPUTBUFFERSIZE=100000
MSGFORMAT=MQSTR
EXPIRYSHUTDOWN=20
CONVERT=N
RECONNECT=5,3000,5
PROCESS_CONTROL_MSGS
```

After you have created the initialization file and saved the name of the file in Design Manager, you can do any of the following:

- Reference the initialization file at run time by using the `INIFILE` or `UNICODEINIFILE` parameters. You can either reference the initialization file as a part of your control file setup, or you can place the name of the initialization file as an argument for a supported engine switch.
- Open the text file and edit the parameters to dynamically manipulate specific connector properties.
- Reuse the initialization file for different applications.

Note: Reusing the initialization file may require you to change some of the values of parameters depending on the requirements for the connector and application you produce.

Some connector parameters support the use of double-byte characters. Using double-byte characters in a parameter gives you the ability to retain information in a native language that is a value for the parameter, such as file names. If the connector that you are configuring supports parameters that include double-byte characters, you must first specify the type of initialization file that you will be using to define the parameters. For example, if your initialization file for the WSMQ Connector lists the `MESSAGEFILE` parameter with a message file name value that uses double-byte characters, you must first specify a Unicode initialization file by defining the `UNICODEINIFILE` parameter.

To specify the type of initialization file to use, you must specify one of the following parameters.

Parameters for specifying an initialization file

Parameter	Description
`INIFILE`	The `INIFILE` parameter specifies an initialization file containing parameters that support only the use of single-byte characters. When specifying connector behavior with an initialization file, the `INIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. **Example:** `INIFILE=MyINIFile.ini` There is no default value for this parameter.
`UNICODEINIFILE`	The `UNICODEINIFILE` parameter specifies an initialization file that contains one or more supported parameters with Unicode characters. When specifying connector behavior with an initialization file that contains Unicode characters in parameters, the `UNICODEINIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. When the engine does not detect a byte-order marker, you must use the `UNICODEINIFILE` parameter so that the engine can interpret Unicode characters correctly. **Example:** `UNICODEINIFILE=MyUnicodeINIFile.ini` There is no default value for this parameter.

Chapter 3: Creating a User-Written DDA Routine

Exstream lets you use a pre-defined connector or your own custom DDA routine to complete processes in an on-demand environment. A custom DDA routine is a user-written DDA routine that is configured for the engine to interpret. The user-written DDA routine includes custom code that defines specific actions that you want the engine and third-party software to complete.

Using a user-written DDA routine to connect Exstream and your existing enterprise architecture lets you accomplish the following goals:

- Connect software in your existing system with Exstream.
- Manipulate incoming data so that the data is formatted for Exstream.
- Manipulate output so that it is configured to work with a software application in your existing system.

For example, suppose that your organization wants to import customer data records from an existing mainframe system that is being phased out of your enterprise architecture. Your company is integrating Exstream and wants to create a database for Exstream so that the customer records can be used in future Exstream designs. The customer data records that are stored in the mainframe system are formatted in capital letters, with each record being 20 characters long. One of the goals that you want to accomplish is to transform the customer data records so that customer names are camel-case and are divided into two fields (first name and last name). You write a DDA routine configured for the engine and create a connector object in Design Manager that specifies the information that the engine requires. The user-written DDA routine requests customer records from the mainframe and transforms the data before the data reaches the Exstream engine. The engine processes the transformed data and writes the data to the specified database based on the actions specified by the set of commands in the user-written DDA routine.

This chapter discusses the following topics:

3.1 About Creating a User-Written DDA Routine

Before you can connect Exstream to third-party software in your existing system with a user-written DDA routine, you must configure the DDA routine to communicate with the engine and your enterprise architecture. When you configure a licensed connector module in Exstream, the basic format is provided as a part of the pre-defined connector. Before the engine can communicate with your existing enterprise architecture, the user-written DDA routine must provide certain information, such as parameters, functions, and other components. You must also make sure that the language that is used to write your DDA routine is compatible with Exstream and your company's enterprise architecture.

All of the DDA routines that you use with Exstream are made up of the following components:

- “Supported Programming Languages for User-Written DDA Routines” below
- “Parameters for DDA Routine Functions” on the next page
- “DDA Routine Functions” on page 39
- “DDA Routine Return Values” on page 39
- “DDA Routine Output Headers” on page 40

3.1.1 Supported Programming Languages for User-Written DDA Routines

Before you write a DDA routine in Exstream, you must first determine the programming language that you will use to write your DDA routine. Typically, the programming language that you use is the same as the programming language used in the systems in your enterprise architecture. For example, if the enterprise application that you want to connect to Exstream is built using C++, then the programming language used to write your DDA routine is C++. For best results, you can use the following information to determine the programming language for your user-written DDA routine that best fits your existing systems and architectures.

Programming languages used with operating systems

Programming language	Windows	UNIX/Linux	z/OS
C/C++	X	X	X
Java	X	X	
.NET	X		

Because C/C++ is available for all available operating systems, C/C++ is the most common programming language used to write a DDA routine. Using C/C++ to write your DDA routine lets you use the user-written DDA routine in a Java or .NET environment, which allows for more flexibility when using user-written DDA routines in Exstream.

3.1.2 Parameters for DDA Routine Functions

The parameters used for DDA routine functions provides information to the Exstream engine for input and output processing. The engine must know the amount of bytes to process for incoming data and the type of access that is being requested by the DDA routine. The amount of bytes processed are provided in a buffer. For example, if your user-written DDA routine provides customer data records from a customer driver file, the DDA routine must specify the maximum amount of data, in bytes, that are available for the engine to process.

All DDA routines in Exstream export the following four parameters, in sequence:

1. Maximum buffer size
2. Current buffer size
3. Address of buffer
4. Access type

Configuring Parameters for Your User-Written DDA Routine

To configure the parameters for DDA routine functions, you must include values for the following information.

Parameters for DDA routine functions

Parameter	Data type	Description
`ulMaxBufferSize`	Unsigned long	The `ulMaxBufferSize` parameter specifies the maximum amount of data, in bytes, that are processed for the size of incoming (input) and outgoing (output) data. The buffer size amount that you set for the `ulMaxBufferSize` parameter corresponds to the **Maximum buffer size** box on the connector object properties in Design Manager. When you add your custom DDA routine as a connector object in Design Manager, the amount that you set for the `ulMaxBufferSize` parameter must be entered in the **Maximum buffer size** box on the connector object properties. For more information about connector object properties, see "Configuring Dynamic Data Access" on page 21. • **Input**—You must set the `ulMaxBufferSize` parameter to an amount greater than the largest record in the data. For example, of a customer record has 60,000 bytes of data, then the maximum buffer size must be set to an amount greater than 60,000 bytes. • **Output**—You must set the `ulMaxBufferSize` parameter to accommodate all of output produced by the engine. For example, if all of the output produced by the engine is 50,000 bytes, then the maximum buffer size must be set to at least 50,000 bytes.
`ulBufferSize`	Unsigned long	The `ulBufferSize` parameter specifies the current buffer size that you want to supply to the engine. The DDA routine supplies the engine with the current buffer size for input processing following a read or seek operation. The engine supplies the current buffer size during an open or write operation. • **Input**—You must set the `ulBufferSize` parameter to contain the number of bytes that the engine is to read. During the read or seek operation, the engine can process data at a maximum of 32,767 bytes at a time. If your incoming data exceeds 32,767 bytes, then you must buffer the data 32,767 bytes at a time. • **Output**—You must set the `ulBufferSize` parameter to the number of bytes that the engine must write as output. In the event that the engine is processing a record transform or post-processor DDA routine, the value of the parameter can change.

Parameters for DDA routine functions, continued

Parameter	Data type	Description
`szBuffer`	Character array	The `szBuffer` parameter specifies the address value of the I/O buffer to the engine following a read or seek operation. • **Open**—On an open operation, the `szBuffer` parameter specifies the parameters used to define the behavior of the DDA routine. The `szBuffer` parameter for an open operation, corresponds to the **Open parameters** box on the connector properties in Design Manager. When you add your custom DDA routine as a connector object in Design Manager, the amount that you set for the `szBuffer` parameter must be entered in the **Open parameters** box on the connector properties. For more information about connector object property settings, see "Configuring Dynamic Data Access" on page 21. • **Seek**—On a seek operation, depending on the type of call, the `szBuffer` parameter contains one of the following strings: • **Position call**—The `szBuffer` parameter contains the relative position. • **Lookup call**—The `szBuffer` parameter contains the value of the search criteria. When the DDA routine finds the string value, the DDA routine sends the relative position to the buffer. If the seek call is unsuccessful, the DDA routine sends a `0` as the value. • **Output**—The `szBuffer` parameter contains the data to return to the engine. The type of data returned to the engine depends on the type of engine request and the specific function of the DDA routine. • **Read**—On a read operation, the `szBuffer` parameter contains the data from the data file.

Parameters for DDA routine functions, continued

Parameter	Data type	Description
`ulAccessType`	Unsigned long	The `ulAccessType` parameter specifies the type of access operation. The `ulAccessType` parameter uses integer values to define the type of access that the engine requests from the DDA routine. The engine uses one of the following ten integer values to specify the access type: • `0`—The `0` integer value indicates a read access type operation. The engine uses a read access type operation to collect data from a data source. The byte array contains data in the same format specified in the data file that is mapped to the application. • `1`—The `1` integer value indicates a write access type operation. The engine uses a write access type operation to write information from a data source. The byte array contains data in the same format specified in the data file that is mapped to the application. • `2`—The `2` integer value indicates an update access type operation. The engine uses an update access type operation to overwrite the contents of an existing data source. • `3`—The `3` integer value indicates a seek access type operation. The engine uses the seek access type operation to find a particular key or index in the buffer. Typically, only reference files and report files use a seek access type operation. • `4`—The `4` integer value indicates an open access type operation. The engine uses the open access type operation as the first call to any DDA routine. An open access type operation initializes the DDA routine, accesses the parameters, and sets up any other resources that are required. An open access type operation contains the open parameters specified in the **Open parameters** box of the associated connector object in Design Manager. • `5`—The `5` integer value indicates a close access type operation. The engine uses a close access type operation to close, disconnect, release resources the DDA routine uses. A close access type operation allows you to reinitialize any DDA routine previously initialized by an open access type operation. • `7`—The `7` integer value indicates an end of transaction access type operation. The engine uses an end of transaction access type operation as a one-time operation when processing all currently available input data has been completed. • `8`—The `8` integer value indicates an end of customer access type operation. The engine uses the end of customer access type operation as a one-time operation at the end of all customer processing. • `9`—The `9` integer value indicates a transform access type operation. The engine uses the transform access type operation for both input and output transforms and passes data through the byte array similar to a write operation. The routine passes the transformed data back to the engine similar to a read operation. If the DDA routine requires more data from the engine, the DDA routine issues a return value of `8`. If the transformed data exceeds the maximum buffer size, the DDA routine issues a return value of transform operation to retrieve the rest of the data. After the engine receives a `9` to indicate a transform, the engine sends another call to the record transform DDA routine to retrieve the rest of the data. • `10`—The `10` integer value indicates a finalize transform access type operation. The engine uses the finalize transform access type operation for output transforms only. If the transformed data exceeds the specified buffer size, the engine makes a finalize transform call (`9`) to receive the remaining transformed data. The engine calls this operation before it calls a close operation.

3.1.3 DDA Routine Functions

Any user-written DDA routine that you write must use the `getRec` function to retrieve data records and the `writeRec` function to write data records. The type of user-written DDA routine that you write determines the type of function that your DDA routine exports.

As an optional method for defining the appropriate DDA routine function, you can use the DDAFileMap built-in function available in Design Manager.

For example:

```
getRec(unsigned long * ulMaxBufferSize,
unsigned long * ulBufferSize, char * szBuffer,
unsigned long * ulAccessType);
writeRec(unsigned long * ulMaxBufferSize,
unsigned long * ulBufferSize, char * szBuffer,
unsigned long * ulAccessType);
```

For more information about the DDAFileMap built-in function, see *Using Logic to Drive an Application* in the Exstream Design and Production documentation.

Each DDA routine type requires a specific DDA routine function. Use the following information to determine the DDA routine function that your type of DDA routine exports.

Required DDA routine functions

Type of DDA routine	Required DDA routine function
Customer driver file DDA routine	`getRec`
Reference file DDA routine	`getRec`
Placeholder file DDA routine	`getRec`
Report file DDA routine	`writeRec`
Output file DDA routine	`writeRec`

3.1.4 DDA Routine Return Values

When you write a custom DDA routine, you must specify the access type operation that you want the engine to perform. The engine uses a set of integer values to indicate the access type operation to perform. Likewise, your DDA routine must return an integer value that indicates the type of operation performed.

The following table identifies the integer values that your DDA routine can return to the engine after an access type operation is performed.

Return type definitions and integer values

Return Value Type	Definition	Integer Value
`USERPROGRAMDATARETURNED`	Data returned (indicates success)	`0`
`USERPROGRAMBUFFEROVERFLOW`	Buffer overflow (the number of bytes read exceeds the maximum)	`1`
`USERPROGRAMEOF`	End of file	`2`
`USERPROGRAMUNABLETOOPENFILE`	Unable to open file	`3`
`USERPROGRAMSHUTDOWNENGINE`	Shut down engine (Transaction mode only)	`4`
`USERPROGRAMFLUSHREPORTS`	Flush report files (Transaction mode only)	`5`
`USERPROGRAMEOFFILESCLOSED`	Files closed	`7`
`USERPROGRAMMOREAVAILABLE`	A record transform or output processor has more data to return	`8`
`USERPROGRAMMOREREQUESTED`	A record transform wants more data passed to it	`9`
`USERPROGRAMCAPTUREON`	Tracefile input capture starts	`10`
`USERPROGRAMCAPTUREOFF`	Tracefile input capture ends	`11`
`USERPROGRAMRELOAD`	Unloads package file(s), waits 60 seconds, and reloads them	`12`
`USERPROGRAM_COMMAND`	Returns commands to the engine for processing	`17`

3.1.5 DDA Routine Output Headers

Whether you assign your DDA routine to an output queue in Design Manager or add the DDAOUTPUT engine switch to your control file, the engine produces DDA output. The output is buffered and presented to your DDA routine with each normal output break. The output appears in the buffer preceded by a header. If you use the APPEND value for the DDAMESSAGE engine switch, the output is followed by the engine messages for the transaction. The value of the 'SYS_DDAOutputUserData' variable appears at the end of the buffer.

For more information about assigning a DDA routine to an output queue, see “Assigning a Connector Object to an Output Queue” on page 28.

The following table describes the layout of the output user data with header information.

DDA routine output header information

Offset	Contents
0	Header size (currently 32)
4	Header version (currently 101)

DDA routine output header information, continued

Offset	Contents
8	Output Driver code: • 1—AFP • 2—PostScript • 3—Metacode • 4—PDF • 5—PCL • 6—IJPDS • 7—VPS • 8—RTF • 9—HTML • 10—XML (composed) • 11—PPML • 12—TIFF • 13—Line Data • 14—VIPP • 15—PowerPoint • 16—VDX • 17—MIBF • 18—XML (content) • 19—Live • 20—PDF/A • 21—EDGAR HTML • 22—Screen (TOP) • 23—XML (Multi-channel) • 24—Word (2007/2010) • 26—HTML (email) • 27—ZPL • 28—Empower • 29—PDF/VT • 401—Report files
12	Size of output
16	Size of messages
20	Size of user data
24	Page count

DDA routine output header information, continued

Offset	Contents
28	Maximum return code: • 0—Normal exit with no errors or warnings • 2—Informational • 4—Normal exit with warnings • 8—Normal exit with errors • 12—Severe error—program stop • 16—Abnormal termination
[Header size]	Composed output
[Header size] + [size of output]	Messages
[Header size] + [size of output] + [size of messages]	Contents of the 'SYS_DDAOutputUserData' variable

The brackets in the table represent actual values, not offset values. The eight offset values in DDA output are all big-endian 32-bit long words.

3.2 Configuring a User-Written DDA Routine in Design Manager

You configure user-written DDA routines for Exstream in the same manner that you configure a pre-defined connector. First, you must add the user-written DDA routine as a connector object in Design Manager. Then, you must assign the connector object to other objects in Design Manager according to the goals that you want to accomplish with your user-written DDA routine.

To configure a user-written DDA routine in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**. The **DLL(C++)** option can be used for all programming languages.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to your DDA routine.
Specify the type of function name	In the **Function name** box, enter the main function that you want the engine to call when connecting to the DDA routine.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size to accommodate data handling requirements.
Specify parameters	Set up your initialization file and enter each parameter on a single line. The parameters that you enter in your initialization file are the parameters that define the actions of your DDA routine, and should be provided in the user-written DDA routine. For more information about setting up your initialization file to define parameters, see "Creating an Initialization File" on page 30.

   For more information about creating a connector object in Design Manager, see "Configuring Connectors in Design Manager" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning your DDA routine to a data file for data input, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to a data file for record transforms.

   For more information about assigning your DDA routine to a data file for record transforms, see "Assigning Connector Objects to a Data File for Record Transform" on page 25.

4. Assign the connector to a placeholder variable.

   For more information about assigning your DDA routine to a placeholder variable, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

5. Assign the connector to an output.

   For more information about assigning your DDA routine to an output, see "Assigning a Connector Object to an Output" on page 27.

6. Assign the connector to an output queue.

   For more information about assigning your DDA routine to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

3.3 About User-Written DDA Routines in a Java Environment

Exstream offers a pre-defined DDA routine that you can configure in Design Manager for your Java-based, user-written DDA routines. The Java Enabler lets the Exstream engine communicate with enterprise architectures that follow a Java-based protocol. With the Java Enabler, the engine can communicate with routines written in Java to complete custom data pre-processing, post-processing, retrieval, and storage.

The Java Enabler is a C++ routine that uses the Java Invocation Interface, part of the Java Native Interface (JNI), to load a Java Virtual Machine (JVM) in the engine. Based on user-specified arguments, the Java Enabler uses the JVM to start and call methods in user-written Java classes. The Java Enabler supports routines in the Java environment. The `JavaEnabler.dll` and `JavaEnabler.jar` files that make up the Java Enabler DDA routine are available as part of the base installation of Exstream Design and Production.

Because Exstream offers both a 32-bit and 64-bit production engine, you must use the correct version of the Java Enabler. For example, if you upgrade your production engine to the 64-bit version, you must use the 64-bit version of the Java Enabler.

For more information about the 64-bit version of the Exstream production engine, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

This section discusses the following topics:

- “Configuring a User-Written DDA Routine for the Java Environment” on the next page
- “Required Java Enabler Parameters” on page 46
- “Java Class Methods for the Java Enabler” on page 48

3.3.1 Configuring a User-Written DDA Routine for the Java Environment

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the Java Enabler. For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, the buffer size is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than `32767`, then the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Creating an Initialization File" on page 30.

   For more information about creating a connector object in Design Manager, see "Creating a Connector Object in Design Manager" on page 22.

2. Assign the connector to a data file for data input.

   For more information about assigning your DDA routine to a data file for data input, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to a data file for record transforms.

   For more information about assigning your DDA routine to a data file for record transforms, see "Assigning Connector Objects to a Data File for Record Transform" on page 25.

4. Assign the connector to a placeholder variable.

   For more information about assigning your DDA routine to a placeholder variable, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

5. Assign the connector to an output.

   For more information about assigning your DDA routine to an output, see "Assigning a Connector Object to an Output" on page 27.

6. Assign the connector to an output queue.

   For more information about assigning your DDA routine to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

3.3.2 Required Java Enabler Parameters

To configure the parameters for the Java Enabler, you must include the following parameters in addition to any parameters that you include in your user-written DDA routine.

Open parameters for the Java Enabler

Parameter	Description
`CLASS`	The `CLASS` parameter specifies the name of a Java class that implements the Java Enabler interface. This user-written Java class contains the custom code to use in place of the data file. The class name is case-sensitive.
`CLASSPATH`	The `CLASSPATH` parameter indicates the directory where the Java class resides. The default value for this parameter is the working directory. Although multiple data files can use the Java Enabler, only one JVM is launched. The shared JVM is set to use the `CLASSPATH` indicated in the open parameters for the associated connector object. Consequently, subsequent `CLASSPATH` parameters are ignored. To avoid ignoring subsequent `CLASSPATH` parameters, set the `CLASSPATH` parameter for all data files that use the Java Enabler in an application to a common path.
`INSTANCE`	The `INSTANCE` parameter indicates whether this instance of the class is shared or not. Select from the following values: • `UNIQUE`—If you set the parameter to `UNIQUE`, the data file uses a unique instance of the specified `CLASS`. The `UNIQUE` value is the default value for the `INSTANCE` parameter. • `SHARED`—If you set the parameter to `SHARED`, other data files of the same class that also have their `INSTANCE` parameter set to `SHARED` can share the same instance of the class. As a result, custom code for different data files (even different types of data files) can share data.
`JVMOPT`	The `JVMOPT` parameter lets you customize the actions of the JVM. For example, you can set JVM parameters that control the `CLASSPATH` parameter, output information, or maximum heap size. Setting JVM options with the `JVMOPT` parameter for the file path to the Java class or maximum heap size overrides settings you set for the `JVMSIZE` and `CLASSPATH` parameters. For more information about using the JVMOPT parameter, see "Using the JVMOPT Parameter to Customize the Java Virtual Machine" on the next page.
`JVMSIZE`	The `JVMSIZE` parameter sets the maximum memory allocation for the preview engine's JVM, usually in response to receiving out-of-memory exceptions within that JVM. The number you supply must be a multiple of 1024. The `JVMSIZE` parameter is an optional parameter since the preview engine's JVM size is not typically an issue even with fairly large documents. The engines are single threaded and the default JVM size is generally 64 MB (but the JVM size varies by operating system, hardware, and JVM).

Open parameters for the Java Enabler, continued

Parameter	Description
`TRACE`	The `TRACE` parameter specifies whether debugging information is written to the console at run time. Select from one of the following values: • Y—The `Y` value enables debugging information. • N—The `N` value disables debugging information. The `N` value is the default value for the `TRACE` parameter.

Using the JVMOPT Parameter to Customize the Java Virtual Machine

When defining parameters for the Java Enabler, you must set a file path to the Java class to indicate the directory where the Java class resides. You can set the file path to the Java class either with the `CLASSPATH` open parameter or with the `JVMOPT` parameter, but the `JVMOPT` parameter lets you set additional arguments that give you more control over the JVM in use.

The following example uses the `JVMOPT` parameter to define specific actions for the JVM in use:

```
CLASS=com/exstream/connector/JMSConnector
JVMOPT=-Djava.class.path=D:\Projects\misc_
progs\JMSConnector.jar;C:\dev\Sun\MessageQueue\lib\imq.jar
JVMOPT=-Xrunhprof:heap=sites,cpu=samples,depth=10,monitor=y,thread=y,doe=y
JVMOPT=-verbose
JVMOPT=-Xrs
JVMOPT=-Xms32m
JVMOPT=-Xmx256m
TRACE=Y
SERVERTYPE=SunONE
MODE=RW
USERNAME=admin
PASSWORD=admin
HOSTNAME=localhost
PORT=6767
QUEUENAME=testq
TIMEOUT=30
INSTANCE=SHARED
```

In the previous example, the `JVMOPT` parameter is defined to perform the following actions:

- The `JVMOPT` parameter sets the file path to the Java class. (The `JVMOPT` parameter operates in the same way as the `CLASSPATH` parameter.)
- Next, the `JVMOPT` parameter sets the JVM to dump usage statistics.
- The `JVMOPT` parameter then sets the JVM to produce information on the actions of the JVM.
- After the JVM is set to produce information on the actions of the JVM, `JVMOPT` parameter disables certain signals usually handled by the JVM.

- The `JVMOPT` parameter then defines the initial and maximum heap size. (The `JVMOPT` parameter defines the initial and maximum heap sizes in the same way as the `JVMSIZE` parameter.)

3.3.3 Java Class Methods for the Java Enabler

A user-written Java class specifies the methods that the engine calls through the Java Enabler.

Even if a method does not contain a command, any class that implements the Java interface must have every method specified.

DDA routine methods for the Java Enabler

Name	Method	Description
Close	`public void close() throws IOException;`	After the engine completes the processing of a customer, it calls the close method. The close method requires no arguments and has no return value.
End of Customer	`public void onEndOfCustomer();`	After the engine completes the processing of a customer, it calls the end of customer method. The end of customer method requires no arguments and has no return value.
End of File	`public boolean isEOF();`	If a read call returns null, the engine calls the end of file method. The end of file method requires no arguments and returns a Boolean value indicating whether or not there is more data on subsequent read calls. For a customer driver file, you can use the `isEOF` method to set the engine to transaction mode. As long as this method returns `false`, the engine continues to poll the read method for more data. If this method returns `true`, the engine shuts down.
End of Transaction	`public void onEndOfTransaction();`	After the engine completes the processing of a customer, it calls the end of customer method. The customer run ends when the read call returns `null`. The end of customer method requires no arguments and has no return value.

DDA routine methods for the Java Enabler, continued

Name	Method	Description
Open	`public void open(String config) throws IOException;`	Before any other methods, the engine calls the open method once to initialize the instance of the class. The open method requires one argument: the configuration string. The configuration string consists of multiple lines of text in `key=value` format, either in the **Open parameters** box on the connector properties or in an initialization file. The open method implements the functionality necessary to open, connect, or set up any resources used by the class. If there is an error in opening the required resources, the open method throws an `IOException`. The Java Enabler catches the `IOException`, notifies the user, and shuts down the engine.
Read	`public byte[] read() throws IOException;`	The engine calls the read method to collect data. The read method requires no arguments and returns a byte array containing the read data. If no more data is available, the read method returns null. If an error occurs, the read method throws an `IOException`. The Java Enabler catches the `IOException`, notifies the user, and shuts down the engine.
Seek	`public uint seek(String key) throws IOException;`	The engine calls the seek method to find a specific key or index. The seek method requires a string key value as an argument. The return value indicates the size of the data record found. If an error occurs, the seek method throws an `IOException`. The Java Enabler catches the `IOException` and notifies the user.
Transform	`public byte[] transform(byte[] bytes) throws IOException;`	You can use the transform method for both input and output transforms. The engine passes data in through the byte array and the DDA routine passes back the transformed data to the engine. The transform method requires no arguments and has no return value. If you are writing a Java connector for use with the Java Enabler, you must provide a body for the transform method. Otherwise, the method can be empty.

DDA routine methods for the Java Enabler, continued

Name	Method	Description
Finalize Transform	`public byte[] transformFinalize() throws IOException;`	The engine calls the finalize transform method right before it calls the close method. You use the finalize transform method for output transforms only. If the DDA routine buffers the transformed data from previous transform calls instead of immediately returning the data to the engine, you can use this method to retrieve the contents of that buffer. The bytes returned are added to the end of data that was returned from each transform method call. If you are writing a Java connector for use with the Java Enabler, you must provide a body for the finalize transform method. Otherwise, the method can be empty.
Write	`public void write(byte[] bytes) throws IOException;`	The engine calls the write method to write data. It takes one argument: a byte array containing the data to write. If an error occurs, the write method returns an `IOException`. The Java Enabler catches the `IOException` and notifies the user.

Required Java Class Methods for Data File Types

Depending on the type of user-written DDA routine that you use, certain Java class methods are required. For example, a customer driver file DDA routine requires the open, close, read, and end of file Java class methods.

In certain cases, a Java class provides functionality for multiple data file types. You can use Java classes that provide functionality for multiple data file types to enable a single class to serve as both a customer driver file and report file, and share data between engine read and write calls.

The following table indicates which Java class methods are required for different data file types.

Required Java class methods for different data files

Methods	Initialization file	Driver file	Reference file	Report file	Placeholder variable
Close	required	required	required	required	required
End of Customer				optional	
End of File	required	required			
End of Transaction				optional	
Open	required	required	required	required	required
Read	required	required	required		required

Required Java class methods for different data files, continued

Methods	Initialization file	Driver file	Reference file	Report file	Placeholder variable
Seek			required		required
Transform	optional	required	optional	optional	optional
Finalize Transform	optional	required	optional	optional	optional
Write				required	

3.4 About User-Written DDA Routines in a .NET Environment

Exstream offers a pre-defined DDA routine that you can configure in Design Manager for your .NET-based user-written DDA routines. The CLR Enabler is an optional DDA routine that is included with Exstream to help streamline the deployment of DDA applications in the .NET Framework environment. The `CLREnabler.dll` and `CLRConnector.dll` files that make up the CLR Enabler are available as part of the base installation of Exstream Design and Production.

The `CLREnabler.dll` file (written in C++) supports DDA routines that are written as a .NET assembly that is on a Windows 2000 or later platform. The `CLRConnector.dll` (a .NET assembly that is contained in a COM Callable Wrapper (CCW) directs data between a user-written .NET assembly and the CLR Enabler.

Because Exstream offers both a 32-bit and 64-bit production engine, you must use the correct version of the CLR Enabler. For example, if you upgrade your production engine to the 64-bit version, you must use the 64-bit version of the CLR Enabler.

For more information about the 64-bit version of the Exstream production engine, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

This section discusses the following topics:

- “Completing Pre-Configuration Tasks” on the next page
- “Configuring a User-Written DDA Routine for the .NET Environment” on page 53
- “Required CLR Enabler Parameters” on page 54
- “Specifying .NET Classes for the CLR Enabler” on page 54
- “Adding a Reference to the CLR Connector” on page 57

3.4.1 Completing Pre-Configuration Tasks

Before you can run user-written DDA routines in a .NET environment, you must complete the following steps:

1. Verify that you are using .NET Framework 2.0 to build your .NET DDA project. If you use a different version of the .NET Framework, you receive an error when you load the project. For more information about creating your .NET DDA project, see "Adding a Reference to the CLR Connector" on page 57.
2. Verify that the following files are available with the Exstream engine:
   - `CLRConnector.dll`—The dynamic linked library (DLL) file that allows access to the connector
   - `CLREnabler.dll`—The DLL file that contains the CLR Enabler

   The CLR Enabler and CLR Connector DLLs are packaged and distributed with the Exstream engine. To build and run user-written DDA routines in a .NET environment, you must have the CLR Enabler and CLR Connector DLLs.
3. Register the `CLRConnector.dll` file with the .NET Framework.

   For example:

   ```
   C:\Windows\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe CLRConnector.dll
   ```
4. For best results, make sure that any applications with test files package and run in batch mode with no errors.

3.4.2 Configuring a User-Written DDA Routine for the .NET Environment

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	Enter the path and file name to the CLR Enabler. For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\CLREnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, the buffer size is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than `32767`, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Creating an Initialization File" on page 30.

   For more information about creating a connector object in Design Manager, see "Configuring Connectors in Design Manager" on page 21.

2. Assign the DDA routine to a data file for data input.

   For more information about assigning your DDA routine to a data file for data input, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the DDA routine to a data file for record transforms.

   For more information about assigning your DDA routine to a data file for record transforms, see "Assigning Connector Objects to a Data File for Record Transform" on page 25.

4. Assign the DDA routine to a placeholder variable.

   For more information about assigning your DDA routine to a placeholder variable, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

5. Assign the DDA routine to an output.

   For more information about assigning your DDA routine to an output, see "Assigning a Connector Object to an Output" on page 27.

6. Assign the DDA routine to an output queue.

For more information about assigning your DDA routine to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

3.4.3 Required CLR Enabler Parameters

To configure the parameters for the CLR Enabler, you must include the following parameters in addition to any parameters that you include in your user-written DDA routine.

Open parameters for the CLR Enabler

Parameter	Description
`CLASS`	The `CLASS` parameter specifies the name of the class in the .NET assembly. If the `CLASS` parameter is omitted, the CLR Enabler attempts to load the first class it finds that implements the ICLR Connector interface. If more than one DDA routine is defined in a given assembly, use the `CLASS` parameter.
`DLL`	The `DLL` parameter specifies the full file path to your .NET routine.
`INSTANCE`	The `INSTANCE` parameter indicates whether this instance of the class is shared or not. Select from one of the following values: • `UNIQUE`—If you set the parameter to `UNIQUE` the data file uses a unique instance of the specified `CLASS`. • `SHARED`—If you set the parameter to `SHARED`, other data files of the same class that also have their `INSTANCE` parameter set to `SHARED` can share the same instance of the class. As a result, custom code for different data files (even different types of data files) can share data. The `SHARED` value is the default value for the `INSTANCE` parameter.
`TRACE`	The `TRACE` parameter specifies whether to write debugging information to the console at run time. Select from one of the following values: • Y—The Y value enables debugging information. • N—The N value disables debugging information. The N value is the default value for `TRACE` parameter.

3.4.4 Specifying .NET Classes for the CLR Enabler

Because the CLR Enabler is designed to work specifically within the Windows environment, you must use a user-written .NET class to specify the methods that the engine calls through the CLR Enabler.

The .NET user-written classes must implement the ICLRConnector interface (included with the `CLRConnector.dll` file). This interface specifies the methods the engine calls through the CLR Enabler.

Any class that implements the ICLR interface must have every method specified below. However, certain methods can be left with empty implementations, depending on the type of data file associated with the DDA routine.

DDA routine methods for the CLR Enabler

Name	Method	Description
Close	`public void Close()`	When the engine finishes with the instance of the class, it calls the close method. It takes no arguments and has no return value.
End of Customer	`public void OnEndOfCustomer()`	After the engine completes the processing of a customer, it calls the end of customer method. It takes no arguments and has no return value.
End of File	`public bool IsEOF()`	If a read call returns `null`, the engine calls the end of file method. It takes no arguments and returns a Boolean value indicating whether or not there is more data on subsequent read calls. For a customer driver file, the `isEOF` method can be used to set the engine to transaction mode. As long as this method returns `false`, the engine continues to poll the read method for more data. If this method returns true, the engine shuts down.
End of Transaction	`public void OnEndOfTransaction()`	After the engine completes the processing of a customer (a customer run ends when the read call returns `null`), it calls the end of transaction method. It takes no arguments and has no return value.
GetID	`public int GetID()`	After a successful open call, the engine calls the GetID method. Always set the `GetID` function in your routine to return a zero, as in this example: `public int GetID() { return 0;}`
Open	`public bool Open(string configuration)`	The engine calls the open method once to initialize the instance of the class, before any other methods. It takes one argument: the configuration string. The configuration string consists of multiple lines of text in `key=value` format, either in the **Open parameters** box on the connector properties or in an initialization file. The open method implements the functionality necessary to open, connect, or set up any resources used by the class. If there is an error in opening the required resources, the open method returns `False`. The CLR Enabler catches the `IOException`, notifies the user, and shuts down the engine.
Read	`public byte[] Read()`	The engine calls the read method to collect data. It takes no arguments and returns a byte array containing the read data. If no more data is available, the read method returns `null`. If an error occurs, the read returns `null`.

DDA routine methods for the CLR Enabler, continued

Name	Method	Description
Seek	`public uint Seek(string key)`	The engine calls the seek method to find a specific key or index. It takes one argument: the key. The return value indicates the size of the data record found. If an error occurs, the seek method returns a zero and the CLR Enabler notifies the user.
Transform	`public byte[] Transform(byte[] bytes)`	You can use the transform method for both input and output transforms. The engine passes data in through the byte array and the DDA routine passes back the transformed data to the engine. It takes no arguments and has no return value. If you are writing a CLR connector for use with the CLR Enabler, you must provide a body for the transform method. Otherwise, the method can be empty.
Finalize Transform	`public byte[] FinalizeTransform()`	The engine calls the finalize transform method right before it calls the close method. You use the finalize transform method for output transforms only. If the DDA routine buffers the transformed data from previous transform calls instead of immediately returning the data to the engine, you can use this method to retrieve the contents of that buffer. If you are writing a CLR connector for use with the CLR Enabler, you must provide a body for the transform finalize method. Otherwise, the method can be empty. The bytes returned are added to the end of the data record that was returned from each transform method call.
Write	`public bool Write(byte[] bytes)`	The engine calls this method to write data. It takes one argument: a byte array containing the data to write. If an error occurs, the write method returns a zero and the CLR Enabler notifies the user.

Required .NET Class Methods for Different Driver File Types

Depending on the type of user-written DDA routine that you use, certain .NET class methods are required. For example, a customer driver file DDA routine requires the open, close, read, and end of file .NET class methods.

In certain cases, a class provides functionality for multiple data file types. You can use .NET classes that provide functionality for multiple data file types to enable a single class to serve as both a customer driver file and report file and share data between engine read and write calls.

The following table indicates the methods for DDA routines you intend to use as replacements for various types of data files:

.NET class methods with different data file types

Methods	Initialization file	Driver file	Reference file	Report file	Placeholder file
Close	required	required	required	required	required
End of Customer				optional	
End of File	required	required			
End of Transaction				optional	
GetID	required	required	required	required	required
Open	required	required	required	required	required
Read	required	required	required		required
Seek			required		required
Transform	optional	required	optional	optional	optional
Finalize Transform	optional	required	optional	optional	optional
Write				required	

3.4.5 Adding a Reference to the CLR Connector

After creating a new Class Library .NET project, you must add a reference to the `CLRConnector.dll`.

To add a reference to the CLR Connector, you must complete the following steps:

1. In the Solution Explorer, right-click the **References** node in your project tree and select **Add Reference**.
2. Go to the location of the `CLRConnector.dll` and click **Open**.

   The folder that contains the `CLRConnector.dll` file opens.
3. Click **OK** to add the reference to your project.
4. Save the file.

Chapter 4: Generating On-Demand Application Data

The Application Query Service (AQS) module is a library service that lets on-demand applications query a package file to generate application object data. The AQS module generates an XML-based manifest that presents a hierarchical, property-centric overview of all objects contained within the package file. You can use the AQS manifest to drive user interactions and document processes. In addition, you can track data across multiple requested documents without having to open and interact with the Design Manager interface.

For example, suppose that your company wants to provide new customers with an interview webpage that the customer completes when applying for a new line of credit. The finance department wants to develop personalized letters to send to the customer using the information provided in the interview webpage. Your company uses Exstream for their on-demand document creation needs and licenses the AQS module as a tool to develop more personalized customer communications. Based on the information provided by each customer, the AQS module generates a manifest that your company can use to determine what user information you need from the interview webpage. Your company then builds a driver file based on the information in the AQS manifest and uses Exstream to create a personalized letter for the customer.

In order to generate an XML manifest of package file data, you must have licensed the AQS module. The `AQS.zip` file that supplements the AQS module installation ZIP file (`AQS_6_0_001.zip`) is included in the base installation of Exstream Design and Production. The `AQS.zip` file includes a schema for every object referenced in the AQS manifest. The object schemas provide information that the AQS module uses to create the framework of the XML manifest. To understand the structure of the XML that the AQS module generates, review the object schemas.

You can view object schemas using word processing programs such as Notepad, in Internet browsers such as Internet Explorer, or in XML editors such as oXygen.

For information about package files, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

This chapter discusses the following topics:

- “Enabling the AQS Module” on the next page
- “Generating the AQS Manifest” on the next page
- “Contents of an AQS Manifest” on page 61

4.1 Enabling the AQS Module

Before you can generate the AQS manifest, you must enable the AQS module in the System Settings of Design Manager.

To enable the AQS module, you must complete the following steps:

1. In Design Manager, in the Library, select **Environment > System > System Settings**.
2. Drag the **System Settings** heading to the Property Panel.
3. On the **Basic** tab, click **System Configuration**.

   The **System Configuration** dialog box opens.
4. Click the **Workflow** tab.
5. Select the **Generate manifest for AQS** check box.

When you package and produce the application, the AQS module generates the AQS manifest.

4.2 Generating the AQS Manifest

After you enable the AQS Module in Design Manager, each package file that you produce in Exstream generates an XML manifest of application object data. To generate an XML manifest from the package file, the AQS module must first call a function to retrieve the manifest, and then call another function to store the manifest data in a string value that you can parse.

To generate the AQS manifest, the AQS module must call the following two functions:

- `PackageFileManager`—The `PackageFileManager` function retrieves the list containing the data in each package file.
- `DumpXML`—The `DumpXML` function places the data in a string for easy reference after you collect each manifest.

This section discusses the following topics:

- “Retrieving the AQS Manifest” on the next page
- “Storing the AQS Manifest” on the next page

4.2.1 Retrieving the AQS Manifest

To retrieve the AQS manifest, the AQS module calls the `PackageFileManager` function using the following syntax based on your platform:

- J2EE— `public PackageFileManager(DataInputStream datainputstream)`
- .NET— `public PackageFileManager(Stream datainputStream)`

The `PackageFileManager` function initializes and reads the package file to retrieve the AQS manifest. It then stores the package file in a string to be returned by the `DumpXML` function. This function uses the `java.io.DataInput-Stream` construct.

The data input stream of the package file contains the parameters.

The `PackageFileManager` method throws the `PkgQueryException`, which can include the following messages:

- `"Package file version is lower than what this api supports";`
- `"Package file signature is corrupt";`
- `"Package file does not contain the AQS manifest";`
- `"Package file AQS manifest is empty";`

4.2.2 Storing the AQS Manifest

To store the AQS manifest, the AQS module calls the `DumpXML` function. The `DumpXML` function returns a string that contains the manifest. You can then parse the string or store it.

The entire requested manifest is populated into the `DumpXML` string. You can make use of the string as needed. For example, you can parse the XML string, or you can place the XML string in a file.

Note: The string returns as ASCII. If you are using z/OS, convert the string to EBCDIC.

The `DumpXML` function has the same syntax in both .NET and J2EE:

```
public String DumpXML
```

4.3 Contents of an AQS Manifest

The AQS module returns a string of XML that details the contents of the package file. The AQS module catalogs the following objects:

- Applications
- Barcodes
- Campaigns
- Connectors
- Data files
- Dictionaries
- Documents
- Fonts
- Functions
- Inserters
- Languages
- Library rules
- Locales
- Message types
- Messages
- Metadata
- Non-library rules in Designer
- Non-library rules in Design Manager
- Output queues
- Output devices
- Pages
- Paper types
- Paragraphs
- Sections
- Style sheets

- Styles
- Templates
- Variables

The manifest query string always includes the following header information:

Header values of the AQS manifest

Attribute	Information provided	Example
`xml version`	The `xml version` attribute specifies the XML version used to compose this string.	`xml version="1.0"`
`version`	The `version` attribute specifies the AQS version used to create this manifest.	`version="2.0"`
`exstream-version`	The `exstream-version` attribute specifies the serialization version of the package file.	`exstream-version="1602009"`
`program-version`	The `program-version` attribute specifies the version of Exstream Design and Production that was used to create the package file.	`program-version="16.2.0"`
`dbcs-package`	The `dbcs-package` attribute specifies whether or not the package file was created for double-byte character set output.	`dbcs-package="Yes"`

After the header, all design objects in the package file are cataloged. All design objects include the following values:

Information available for all objects in the AQS manifest

Attribute	Information provided	Example
`oid`	The `oid` attribute provides the database identification number for the object.	`oid="3"`
`folder`	The `folder` attribute specifies the folder identification number for each object.	`folder="4"`
`Name`	The `name` attribute provides the name of the variable.	`Bank Statement`
`Description`	The `description` attribute provides a description of the variable.	`User name`
`vid`	The `vid` attribute specifies the version number of the object.	`vid="1"`
`status`	The `status` attribute specifies the status of the object.	`status="wip"`

This section discusses the following topics:

- "Object Property Information" on the next page
- "Object Relationship Information" on the next page
- "Code Information for the Rule, Function, and Variable List Elements" on page 68

4.3.1 Object Property Information

All `<ObjectList>` elements in the AQS manifest have an `<ObjectProperties>` tag that lists the properties of the referenced object. An element references the objects in the AQS manifest and each element contains a tag that lists the properties of that object. For each object catalogued in the AQS manifest, there is a corresponding string of XML that references all of the properties and assigns them an identifying value. Any property set on a design object has a corresponding attribute and value in the AQS manifest.

The syntax for property tags is as follows:

```
<ObjectNameProperties "property attribute=value" />
```

For example:

```
<ApplicationProperties locale-oid="2" page-count=method="Normal, fronts and
backs"
         report-key-oid="2" tracking-key-oid="2" allow-must-go="1" max-
pages="0"
         max-add-pages="0" max-weight="0.000000" break-type="Weight"
         num-weight-breaks="3"/>
```

For more information about the specific properties available for a particular design object, see *Designing Customer Communications* in the Exstream Design and Production documentation.

4.3.2 Object Relationship Information

The AQS manifest describes the relationship between objects in a package file and gives a representation of which variables and rules are used in the design. The `<ObjectList>` element in the AQS manifest represents objects in the package file. To identify the object relationship, the AQS manifest assigns an object identification number to the `<ObjectUse>` tag. For example, the `<DocumentList>` element contains `<PageUse>` and `<SectionUse>` tags, with each having a specific object identification number that corresponds to a specific page and section used in the document. The following table shows how certain `<ObjectList>` elements reference other objects using the `<ObjectUse>` tag:

Object element and tag relationship

Object element	Object tag	Example

`<ApplicationList>`	The `<ApplicationList>` tag references the following object use tags:
	• `<DocumentUse>`
	• `<CampaignUse>`
	• `<DataFileUse>`
	• `<QueueUse>`
	• `<MetadataUse>`
		`<Documents>`
		   `<DocumentUse oid="1"/>`
		`</Documents>`
		`<Campaigns>`
		   `<CampaignUse oid="1"/>`
		   `<CampaignUse oid="2"/>`
		   `<CampaignUse oid="3"/>`
		`</Campaigns>`
		`<DataFiles>`
		   `<DataFileUse oid="1"/>`
		   `<DataFileUse oid="1"/>`
		   `<DataFileUse oid="3"/>`
		`</DataFiles>`
		`<Queues>`
		   `<QueueUse oid="4"/>`
		   `<QueueUse oid="2"/>`
		`</Queues>`
		`<MetadataObjects>`
		   `<MetadataUse oid="501"/>`
		   `<MetadataUse oid="502"/>`
		`</MetadataObjects>`

`<DataFileList>`	The `<DataFileList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="632"/>`
		   `<VariablesUse oid="625"/>`
		   `<VariablesUse oid="624"/>`
		   `<VariablesUse oid="634"/>`
		`</Variables>`
		`<MetadataObjects>`
		   `<MetadataUse oid="515"/>`
		   `<MetadataUse oid="525"/>`
		`</MetadataObjects>`

`<DocumentList>`	The `<DocumentList>` tag references the following object use tags:
	• `<PageUse>`
	• `<SectionUse>`
	• `<MetadataUse>`
		`<Pages>`
		   `<PageUse oid="30"/>`
		`</Pages>`
		`<Sections>`
		   `<SectionUse oid="2"/>`
		   `<sectionUse oid="1"/>`
		`</Sections>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`<MetadataObjects>`

Object element and tag relationship, continued

Object element	Object tag	Example

`<MetadataList>`	The `<MetadataList>` tag references the following objects:
	• `<ApplicationList>`
	• `<DataFileList>`
	• `<DocumentList>`
	• `<PageList>`
	• `<SectionList>`
	• `<CampaignList>`
	• `<MessageList>`
	• `<PageTemplateList>`
	• `<BarcodeList>`
	• `<InserterList>`
	• `<FunctionList>`
	• `<LanguageList>`
	• `<VariableList>`
		`<Messages>`
		   `<MessageUse oid="1"/>`
		   `<MessageUse oid="2"/>`
		   `<MessageUse oid="3"/>`
		   `<MessageUse oid="4"/>`
		   `<MessageUse oid="5"/>`
		`</Messages>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<Sections>`
		   `<SectionUse oid="2"/>`
		   `<sectionUse oid="1"/>`
		`</Sections>`
		`<Pages>`
		   `<PageUse oid="30"/>`
		`</Pages>`
		`<Documents>`
		   `<DocumentUse oid="1"/>`
		`</Documents>`
		`<Campaigns>`
		   `<CampaignUse oid="1"/>`
		   `<CampaignUse oid="2"/>`
		   `<CampaignUse oid="3"/>`
		`</Campaigns>`
		`<DataFiles>`
		   `<DataFileUse oid="1"/>`
		   `<DataFileUse oid="1"/>`
		   `<DataFileUse oid="3"/>`
		`</DataFiles>`
		`<Queues>`
		   `<QueueUse oid="4"/>`
		   `<QueueUse oid="2"/>`
		`</Queues>`

`<PageList>`	The `<PageList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<RuleUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<Rules>`
		   `<RuleList oid="1870533674"/>`
		   `<RuleList oid="1785862743"/>`
		`<Rules>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

Object element and tag relationship, continued

Object element	Object tag	Example

`<SectionList>`	The `<SectionList>` tag references the following object use tags:
	• `<SectionUse>`
	• `<MessageUse>`
	• `<MetadataUse>`
		`<Sections>`
		   `<SectionUse oid="2"/>`
		   `<sectionUse oid="1"/>`
		`</Sections>`
		`<Messages>`
		   `<MessageUse oid="1"/>`
		   `<MessageUse oid="2"/>`
		   `<MessageUse oid="3"/>`
		   `<MessageUse oid="4"/>`
		   `<MessageUse oid="5"/>`
		`</Messages>`
		`<MetadataObjects>`
		   `<MetadataUse oid="515"/>`
		   `<MetadataUse oid="525"/>`
		`</MetadataObjects>`

`<CampaignList>`	The `<CampaignList>` tag references the following object use tags:
	• `<MessageUse>`
	• `<MetadataUse>`
		`<Messages>`
		   `<MessageUse oid="1"/>`
		   `<MessageUse oid="2"/>`
		   `<MessageUse oid="3"/>`
		   `<MessageUse oid="4"/>`
		   `<MessageUse oid="5"/>`
		`</Messages>`
		`<MetadataObjects>`
		   `<MetadataUse oid="515"/>`
		   `<MetadataUse oid="525"/>`
		`</MetadataObjects>`

`<MessageList>`	The `<MessageList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<RuleUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<Rules>`
		   `<RuleList oid="1870533674"/>`
		   `<RuleList oid="1785862743"/>`
		`<Rules>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

`<PageTemplateList>`	The `<PageTemplate>` tag references the following object use tags:
	• `<VariableUse>`
	• `<RuleUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<Rules>`
		   `<RuleList oid="1870533674"/>`
		   `<RuleList oid="1785862743"/>`
		`<Rules>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

Object element and tag relationship, continued

Object element	Object tag	Example

`<BarcodeList>`	The `<BarcodeList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

`<InserterList>`	The `<InserterList>` tag references the following object use tags:
	• `<BarcodeUse>`
	• `<MetadataUse>`
		`<Barcodes>`
		   `<BarcodeUse oid="423757320"/>`
		   `>BarcodeUse oid="254891001"/>`
		`</Barcodes>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

`<FunctionList>`	The `<FunctionList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="580"/>`
		   `<VariablesUse oid="581"/>`
		`</Variables>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

`<LanguageList>`	The `<LanguageList>` tag references the following object use tags:
	• `<DictionaryUse>`
	• `<MetadataUse>`
		`<Dictionaries>`
		   `<DictionaryUse oid="1"/>`
		   `<DictrionaryUse oid="2"/>`
		`</Dictionaries>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

`<RuleList>`	The `<RuleList>` tag references the following object use tag:
	`<VariableUse>`
		`<Variables>`
		   `<VariablesUse oid="632"/>`
		   `<VariablesUse oid="625"/>`
		   `<VariablesUse oid="624"/>`
		   `<VariablesUse oid="634"/>`
		`</Variables>`

`<VariableList>`	The `<VariableList>` tag references the following object use tags:
	• `<VariableUse>`
	• `<MetadataUse>`
		`<Variables>`
		   `<VariablesUse oid="632"/>`
		   `<VariablesUse oid="625"/>`
		   `<VariablesUse oid="624"/>`
		   `<VariablesUse oid="634"/>`
		`</Variables>`
		`<MetadataObjects>`
		   `<MetadataUse oid="626"/>`
		   `<MetadataUse oid="625"/>`
		`</MetadataObjects>`

4.3.3 Code Information for the Rule, Function, and Variable List Elements

Most applications produced from the Exstream platform use logic to define specific rules and functions. When you generate an AQS manifest, the code for these rules and functions appears in the `<Formula>` tag contained in the `<RuleList>`, `<FunctionList>`, and `<VariableList>` elements. The following example shows how the code used to create a Library rule appears in the AQS manifest.

For more information about using logic for rules, formulas, and functions, see *Using Logic to Drive an Application* in the Exstream Design and Production documentation.

```
<Rule>
   <Scope oid="15" folder="3">
      <Name>Letter Code is 0100</Name>
      <Description />
   </Scope>
   <Version vid="1" status="wip" />
   <RuleProperties in-Library="1">
      <Formula>
      <![CDATA[
         IF(Letter_Code = '0100') THEN
            INCLUDE
         ENDIF
      ]]
      </Formaula>
   </RuleProperties>
</Rule>
<Rule>
   <Scope oid="16" folder="3">
      <Name>Letter Code is 0200</Name>
      <Description>
   </Scope>
   <Version vid="1" status="wip" />
   <RuleProperties in-library="1">
      <Formula>
      <![CDATA[
         IF(Letter_Code = '0200') THEN
            INCLUDE
         ENDIF
      ]]
      </Formula>
   </RuleProperties>
</Rule>
```

Chapter 5: Exstream Engine as a Web Service

EWS is a variation of the Exstream production engine that you can use to deliver documents through a web service using the web-based communication protocol, Simple Object Access Protocol (SOAP). SOAP is an XML-based protocol that allows a set of software applications to exchange information over HTTP. EWS uses the SOAP protocol to deliver customer documents in an on-demand environment by way of a request/response process. For example, suppose you provide your customers with the ability to request and access their documents over the web, and a customer wants to access an insurance statement. The customer accesses his or her account online and submits a request for the insurance statement. EWS receives the request, the Exstream Design and Production engine generates the document, and EWS delivers the document to the customer in real time.

EWS requires a special set of configurations for both the server side and client side applications. This chapter discusses the requirements for configuring EWS version 2.0 using SOAP to complete a request and response process for delivering customer documents.

The following table provides an overview of some of the key features of EWS version 2.0.

Functionality and configuration highlights of EWS version 2.0

Feature	Description
Product key specification	In EWS 2.0, you have the option to send product key information either using the KEY switch as specified in your SOAP request, or by specifying the product key string as part of the configuration file (`ews-config.xml` for J2EE or `Web.config` for .NET). If you specify product key information in both areas, the product key used in the request overrides the product key information provided in the configuration file. When you specify product key information in the configuration file, you must use the `<ProductKeySBCS>` tag for SBCS applications and the `<ProductKeyDBCS>` tag for DBCS applications. For more information about specifying product key information in your configuration file, see "Configuring EWS for Deployment to the Application Server" on page 71.

Functionality and configuration highlights of EWS version 2.0, continued

Feature	Description
Output	EWS 2.0 allows you to control the behavior of output files in the following ways: • Output files can either be written to the server or returned in the SOAP response. • You can specify multiple output types using the `<OutputHeaderExtensions>` and `<OutputExtensions>` tag in your configuration file. • You can specify the output types you want returned in your response using the `<FileReturnRegEx>` tag in your SOAP request. For more information about specifying multiple output types, see "Configuring EWS for Deployment to the Application Server" on the next page. For more information about specifying specific output types, see "Configuring SOAP Requests to Return EWS Responses" on page 86.
Customer driver files	In EWS 2.0, all customer driver files sent in your SOAP request as inline XML must be Base64 encoded.

This chapter discusses the following topics:

- "Preparing the Exstream Engine as a Web Service" below
- "Configuring the Exstream Engine as a Web Service" on the next page
- "Testing the Exstream Engine as a Web Service" on page 95

5.1 Preparing the Exstream Engine as a Web Service

EWS is available on multiple platforms. Depending on the platform that you want to use, you must download the appropriate ZIP file from My Support at http://support.opentext.com

EWS is packaged with the SOAP Connector.

The naming convention for the EWS installation ZIP files is as follows:

- On J2EE:

  ```
  EWS_Connector_2_0_[Version release number]_J2EE.zip
  ```

- On .NET:

  ```
  EWS_Connector_2_0_[Version release number]_DotNet.zip
  ```

To prepare EWS for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download EWS version 2.0.
2. Extract the contents of the installation ZIP file to a directory on your machine.
3. Verify that you have met all of the prerequisites for EWS on your platform.
4. Install and configure the application server that you will use with EWS. For example, if you are using WebSphere application server, you must have installed and set up the WebSphere application server before you begin configuring EWS and testing EWS.
5. For best results, make sure any applications that include test files package and run in batch mode with no errors.

5.2 Configuring the Exstream Engine as a Web Service

Configuring EWS requires multiple tasks for both the application server that you will use and an EWS client that handles the requests and response process.

To fully configure EWS for production, you must complete the following tasks:

1. “Configuring EWS for Deployment to the Application Server” below
2. “Configuring the EWS Client” on page 84
3. “Configuring SOAP Requests to Return EWS Responses” on page 86

5.2.1 Configuring EWS for Deployment to the Application Server

Before you can configure the client side application, you must set up the server side of EWS. Because EWS is comprised of two parts: the Exstream engine and the engine service, both parts of EWS must be installed on the same machine. You will first deploy the web service and then edit the configuration file to point EWS to the location of the files that EWS requires to process requests.

Server -side configuration files for EWS 2.0

From the appropriate ZIP file for your server, make sure you have the following files:

- For .NET:
  - `Web.config`—The `Web.config` file is an editable file that you must configure to complete the server side setup for EWS on .NET.

    You access this file from the `EWSService.zip` file.

    For example:

    ```
    EWS_Connector_2_0_0003_DotNet.zip\EWSSerivce.zip\Web.config
    ```

- For J2EE:
  - `ews-config.xml`—The `ews-config.xml` file is an editable file that you must configure to complete the server side set up for EWS on J2EE.

    For WebSphere application servers only, you access the `ews-config.xml` file from the `ExstreamServices.ear` file located in the `ExstreamServices.zip` file.

    For example:

    ```
    EWS_Connector_2_0_0003_J2EE.zip\EngineService.zip\ExstreamServices.ear\
       EngineService.war\WEB-INF\ews-config.xml
    ```

    Additionally, on WebSphere application servers, you must set permissions for the web service by editing the `was.policy` file located in the `ExstreamServices.ear` file.

    For example:

    ```
    EWS_Connector_2_0_0003_J2EE.zip\EngineService.zip\ExstreamServices.ear\
       META-INF\was.policy
    ```

    For all other web application servers you access the `ews-config.xml` file from the `EngineService.war` file located in the `ExstreamServices.zip` file.

    For example:

    ```
    EWS_Connector_2_0_0003_J2EE.zip\EngineService.zip\EngineService.war\
       WEB-INF\ews-config.xml
    ```

This section discusses the following topics:

- “Configuring EWS for the J2EE Environment” below.
- “Configuring EWS for the .NET Environment” on page 79.
- “Using SOAP Headers for Requests Sent to Intermediary Web Services” on page 83.

Configuring EWS for the J2EE Environment

If you are configuring EWS 2.0 for the J2EE environment, you must first verify the type of web application server that you use. For example, if you configure the server side of EWS 2.0 using

WebSphere as your application server, you must use the `ExstreamServices.ear` file. For all other web application servers, you must use the `EngineService.war` file when configuring the server side of EWS.

Additionally, you must make sure that your Web server supports the JAX-WS framework. On WebSphere, the minimum version that you can use is WebSphere 6.1, which must include an installed web service feature pack. WebSphere 7.0 supports JAX-WS.

For EWS 2.0, you must use one of the following files, depending on your application server:

- **WebSphere application server only**— `ExstreamServices.ear`
- **All other application servers**— `EngineService.war`

  The `ExstreamServices.ear` file and `EngineService.war` file contain all of the necessary components to deploy the service.

To configure EWS 2.0 for use in the J2EE environment, you must complete the following tasks as needed:

1. “Preparing Your Machine and Server for EWS Deployment on J2EE” below
2. “Customizing Elements for the EWS Server Configuration on J2EE” on the next page
3. “Modifying Access Privileges” on page 78

Preparing Your Machine and Server for EWS Deployment on J2EE

To prepare your machine and server for EWS deployment on J2EE, you must complete the following steps:

1. Select the appropriate file, depending on the server type that you use.

   For more information about selecting the appropriate file and determining the correct file path location for server configuration files, see “Configuring EWS for Deployment to the Application Server” on page 71.
2. Extract the selected file to a directory on your local machine.
3. Deploy the selected file as required for your server.
4. Make sure that you locate and edit the following files:
   - `ews-config.xml`—This file configures paths and files for the service.

     For more information about editing and configuring the `ews-config.xml` file, see “Customizing Elements for the EWS Server Configuration on J2EE” on the next page.
   - `EngineService.wsdl`—This file lets you define the web service and the operations the web service can perform.
   - `was.policy`—This file sets permissions for the web service and is available only in the `ExstreamServices.ear` file when you use WebSphere as your application server.

Although the `was.policy` file is an editable file, Exstream recommends that you accept the default permissions defined in the file.

For more information about setting permissions using the `was.policy` file, see "Modifying Access Privileges" on page 78.

5. Write the WSDL portion of you server setup using the following syntax:

   ```
   http://<server name>:<port>/<contextRoot>/EngineService?wsdl
   ```

   The `<contextRoot>` portion of the WSDL is the web application context root. On WebSphere, the `<contextRoot>` portion of the WSDL is the default of EWS.

Customizing Elements for the EWS Server Configuration on J2EE

The `ews-config.xml` file contains the element structure that you must use to configure the server side of EWS. The `ews-config.xml` file lets you configure all of the necessary elements required to complete a successful line of communication between the application server and Exstream Design and Production.

The following task explains how to customize the elements in the `ews-config.xml` file.

To customize the elements for the EWS server configuration on J2EE, you must complete the following steps:

1. Go to the `WEB-INF` folder and open the `ews-config.xml` file.
2. Using the syntax `<Tag>Absolute Path Name<Tag>`, customize the following tags:

To	Do this
Customize the `<WorkingDirectory>` element	Specify the absolute path name to the folder that you want the engine to use to store temporary files. For example: `<WorkingDirectory>C:\ExstreamWork\scratch </WorkingDirectory>`

Customize the `<EngineExecutableSBCS>` or `<EngineExecutableDBCS>` elements	Specify the absolute path name to the engine.
	
	If you operate in SBCS, customize the `<EngineExecutableSBCS>` element.
	
	If you operate in DBCS, customize the `<EngineExecutableDBCS>` element.
	
	For example:
	
	`<EngineExecutableSBCS>C:\ExstreamWork\engine\Engine.exe </EngineExecutableSBCS>`
	
	`<EngineExecutableDBCS>C:\ExstreamWork\engine\Engine_ DBCS.exe </EngineExecutableDBCS>`

To	Do this
Customize the `<ProductKeySBCS>` or `<ProductKeyDBCS>` elements	Specify the product key that you want to use to run the engine. If you operate in SBCS, customize the `<ProductKeySBCS>` element. If you operate in DBCS, customize the `<ProductKeyDBCS>` element. Any product key specified in the request using the KEY switch passed in the `<EngineOptions>` element overrides any product key specified in the `<ProductKeySBCS>` or `<ProductKeyDBCS>` elements.
Customize the `<MessageResourceFile>` element	Specify the absolute path name to the message resource file. For example: `<MessageResourceFile>C:\ExstreamWork\engine\ MsgResource_en-us.dat</MessageResourceFile>` The installation file for the Exstream production engine contains the files for each localized language.
Customize the `<PackageFileDirectory>` element	Specify the absolute path name of the folder or directory that contains the package files. This folder or directory must exist and must contain all of the package files that you use with this service. For example: `<PackageFileDirectory>C:\ExstreamWork\packages </PackageFileDirectory>`
Customize the `<OutputHeaderExtension>` element	Specify the output extension that you want to use for output headers. If you do not specify an extension for this element, the default specification is `"hdr"`. For example: `<OutputHeaderExtension>hdr</OutputHeaderExtension>` If EWS requires information about the output file, you can use the WRITEOUTPUTHEADER engine switch to specify a separate location in which to write the header information. You must specify the appropriate output extension with the WRITEOUTPUTHEADER engine switch. For more information about the WRITEOUTPUTHEADER switch, see *Preparing Applications for Production* in the Exstream Design and Production documentation.
Customize the `<DDADirectory>` element	Specify the absolute path name of the folder where any DDA binaries have been installed. This folder must exist in order for you to use DDA routines. For example: `<DDADirectory>C:\ExstreamWork\dda</DDADirectory>`

To	Do this

Customize the `<OutputDirectory>` element	Specify the path name to a folder where output is saved. The specified folder must exist in order for output files to be saved on the host server, unless you use the `<CreateOutputDirectories>` element, and it is set to `True`.
	
	For example:
	
	`<CreateOutputDirectories>true</CreateOutputDirectories>`
	
	You use the `<OutputDirectory>` element so you can request that EWS save output files on the server instead of returning the output files in the request. Customization of this element is optional.
	
	If you customize the `<OutputFile>` element as part of your SOAP request, EWS places all of the generated output in the directory that you specify in the `<OutputDirectory>` element in the `ews-config.xml` file.
	
	For more information about the `<OutputFile>` element, see "Configuring SOAP Requests to Return EWS Responses" on page 86.
	
	For example:
	
	`<OutputDirectory>C:\ExstreamWork\output</OutputDirectory>`

To	Do this

Customize the `<OutputExtensions>` element	Specify the output extension for your output format.
	
	The `ews-config.xml` file lists the default extensions for each output in the `<OutputExtensions>` element. You can change the default extension attribute listed for your output, but the output name attribute is internal and is used to determine the extension of a given output type. To prevent errors, the output name should remain unchanged.
	
	The `<OutputExtensions>` element is defined in the `ews-config.xml` file as follows:
	
	`<OutputExtensions>`
	   `<add output="afp" extension="afp"/>`
	   `<add output="postscript" extension="ps"/>`
	   `<add output="metacode" extension="meta"/>`
	   `<add output="pdf" extension="pdf"/>`
	   `<add output="pcl" extension="pcl"/>`
	   `<add output="ijpds" extension="ijp"/>`
	   `<add output="vps" extension="vps"/>`
	   `<add output="rtf" extension="rtf"/>`
	   `<add output="html" extension="htm/>`
	   `<add output="dxf" extension="dxf"/>`
	   `<add output="ppml" extension="ppml"/>`
	   `<add output="tiff" extension="tiff"/>`
	   `<add output="3211LD" extension="xml"/>`
	   `<add output="vipp" extension="vipp"/>`
	   `<add output="powerpoint" extension="ppt"/>`
	   `<add output="vdx" extension="vdx"/>`
	   `<add output="mibf" extension="mibf"/>`
	   `<add output="content-xml" extension=xml"/>`
	   `<add output="live" extension="dlf"/>`
	   `<add output="pdf-a" extension="pdf"/>`
	   `<add output="edgar-html" extension="htm"/>`
	   `<add output="screen" extension="top"/>`
	   `<add output="multi-channel" extension="xml"/>`
	   `<add output="docx" extension="docx"/>`
	   `<add output="zpl" extension="zpl"/>`
	   `<add output="report" extension="rpt"/>`
	`</OutputExtensions>`
	
	Report file output is available only on EWS version 2.0.
	
	Note: When using output queues, the engine writes the output files to the working directory that you specified in the `ews-config.xml` file. Although the engine uses the file name specified on the output queue as the output file name, the engine ignores the file path specified on the output queue. In the case that output is written to the server and not returned in the request, the engine writes the output to the location specified in the `<OutputDirectory>` element of the `ews-config.xml` file.

To	Do this

Customize the `<RequestFilter>` element	You use the `<RequestFilter>` element to enable SOAP header processing for an intermediary web service.
	
	For more information about using SOAP headers for requests sent to an intermediary web service, see "Using SOAP Headers for Requests Sent to Intermediary Web Services" on page 83.
	
	Although the `<RequestFilter>` element is a required tag for EWS 2.0, specifying a value for the `<RequestFilter>` element is optional. If a value is not specified, EWS ignores the `<RequestFilter>` element.
	
	Specify how the `<RequestFilter>` element is used by modifying the Boolean values of the following attributes:
	
	• `headers-required`—If set to `True`, incoming requests to EWS are required to have header tags. A SOAP fault is generated if the required header tags are missing.
	• `send-request-body`—If set to `True`, the `send-requestbody` tag specifies that the SOAP body should be sent to the intermediary, where it can be processed and then returned to EWS.
	• `use-response-body`—If set to `True`, the `use-responsebody` tag specifies that the SOAP body returned by the intermediary should be used by the engine.
	
	For example:
	
	`<RequestFilter>http://localhost:8080/application/`
	   `header-service</RequestFilter>`
	`<RequestFilter headers-required="false"`
	   `send-request-body="false"`
	   `use-response-body="false">`
	`</RequestFilter>`

Customize the `<ProductionMode>` element	Set the `<ProductionMode>` element to `True` if you want EWS to run in production mode and to produce final output. The system deletes working files that you created during a production run. Set the `<ProductionMode>` element to `False` if you want EWS to run in test mode. The system does not delete working files that you produced during a production run. Typically, the `False` setting is used only for troubleshooting.

Modifying Access Privileges

For WebSphere application servers only, you can modify access privileges for users. Depending on your business needs, you can edit this file for specific installation requirements, but you must first enable read, write, and delete access to the file.

To modify access privileges, you must complete the following steps:

1. Go to the EAR file and open the `was.policy` file.
2. Edit the file to grant read, write, and delete privileges to the engine service file.

Note: Although the `was.policy` file is an editable file, Exstream recommends that you accept the default permissions defined in the file.

Configuring EWS for the .NET Environment

To configure the server side of EWS for the .NET environment, you must edit the `Web.config` XML file. This file controls several options for the service, including the following:

- Directories that you want to use for the service
- Locations of files and programs that EWS must access
- Whether to use an intermediary web service to process requests

To configure EWS 2.0 for use in the .NET environment, you must complete the following tasks:

- “Preparing Your Machine and Server for EWS Deployment on J2EE” on page 73
- “Customizing Elements for the EWS Server Configuration on J2EE” on page 74

Preparing Your Machine and Server for EWS Deployment on .NET

To prepare your machine and server for EWS deployment on .NET, you must complete the following steps:

1. Select the archive files in the EWS installation ZIP file that you have installed on your machine.
2. Extract the files to a directory on your local machine.
3. Create a virtual directory for the Internet Information Service (IIS). For example, the `SOAPService`.
4. Copy the files to the virtual directory.

Customizing Elements for the EWS Server Configuration on .NET

The following task explains how to customize the elements in the `Web.config` file.

To customize elements for the EWS server configuration on .NET, you must complete the following steps:

1. Open the `Web.config` XML file.
2. Using the syntax `<add key="Key Attribute" value="Absolute Path Name"/>`, customize the following elements:

To	Do this
Customize the `WorkingDirectory` element	Specify the absolute path name to a folder that you want the engine to use for storing temporary files. For example: `<add key="WorkingDirectory" value="C;\EngineService\Scratch"/>`

Customize the `EngineExecutableSBCS` or `EngineExecutableDBCS` elements	Specify the absolute path name to the engine.
	
	If you operate in SBCS, customize the `EngineExecutableSBCS` element.
	
	If you operate in DBCS, customize the `EngineExecutableDBCS` element.
	
	For example:
	
	`<add key="EngineExecutableSBCS"`
	`value="C;\EngineService\engine\Engine.exe`
	`<add key="EngineExecutbaleDBCS"`
	`value="C:\EngineService\engine\Engine_DBCS.exe/>`

Customize the `MessageResourceFile` element	Specify the absolute path name to the message resource file.
	
	For example:
	
	`<add key="MessageResourceFile" value="C:\Program`
	`Files\Exstream\DialogueMsgResource_en-us.dat"/>`
	
	The installation file for the Exstream production engine contains the files for each localized language.

Customize the `OutputHeaderExtension` element	Specify the output extension that you want to use for output headers. If you do not specify an extension for this element, the default specification is `"hdr"`. If EWS requires information about the output file, you can use the WRITEOUTPUTHEADER engine switch to specify a separate location to write the header information. You must specify the appropriate output extension with the WRITEOUTPUTHEADER engine switch. For example: `<add key="OutputHeaderExtension" value="hdr"/>` For more information about the WRITEOUTPUTHEADER switch, see *Switch Reference* in the Exstream Design and Production documentation.

Customize the `PackageFileDirectory` element	Specify the absolute path name of the folder or directory that contains the package files. This folder or directory must exist and must contain all package files you use with this service.
	
	For example:
	
	`<add key="PackageFileDirectory"`
	`value="C:\EngineService\packages"/>`

Customize the `DDADirectory` element	Specify the path name of the folder where any DDA binary files have been installed. This folder must exist in order for you to use DDA routines. For example: `<add key="DDADirectory" value="C:\EngineService\dda"/>`

To	Do this

Customize the `OutputDirectory` element	Specify the path name to a folder where output is saved. The specified folder must exist in order for output files to be saved on the host server, unless you use the `<CreateOutputDirectories>` element, and it is set to `True`.
	
	You use this key attribute to request that EWS save output files on the host server.
	
	If you customize the `<OutputFile>` element as part of your SOAP request, EWS places all of the generated output in the directory that you specify in the `<OutputDirectory>` element in the `Web.config` file.
	
	For more information about the `<OutputFile>` element, see "Configuring SOAP Requests to Return EWS Responses" on page 86.
	
	For example:
	
	`<add key="OutputDirectory" value="C:\EngineService\output"/>`
	`<add key="CreateOutputDirectories" value="true"/>`

Customize the `ProductKeySBCS` or `ProductKeyDBCS` elements	Specify the product key that you want to use to run the engine.
	
	If you operate in SBCS, customize the `ProductKeySBCS` element.
	
	If you operate in DBCS, customize the `ProductKeyDBCS` element.
	
	Any product key specified in the request using the KEY switch passed in the `<EngineOptions>` element overrides any product key that is specified in the `<ProductKeySBCS>` or `<ProductKeyDBCS>` elements.
	
	For example:
	
	`<add key="ProductKeySBCS" value=" "/>`
	`<add key="ProductKeyDBCS" value=" "/>`

Customize the `ProductionMode` element	Set `ProductionMode` element to `True` if you want EWS to run in production mode and to produce final output. The system deletes working files created during a production run. Set `ProductionMode` element to `False` if you want EWS to run in test mode. The system does not delete working files produced during a production run. This setting is typically used only for troubleshooting. For example: `<add key="ProductionMode" value="false"/>`

Customize the `RequestFilter`element	Specify as a fully-qualified URL endpoint. You use this key attribute to enable SOAP header processing for an intermediary web service.
	
	Although the `RequestFilter` element is a required attribute for EWS 2.0, you are not required to specify a value for the attribute. If a value is not specified, EWS ignores the `RequestFilter` element.
	
	The `headers-required`, `send-request-body`, and `use-response-body` tags must be configured in conjunction with the `RequestFilter` tag.
	
	For example:
	
	`<add key="RequestFilter"`
	`value="http://localhost:1626/RequestFilter.asmx"/>`
	
	For more information about using SOAP headers for requests sent to an intermediary web service, see "Using SOAP Headers for Requests Sent to Intermediary Web Services" on page 83.

To	Do this
Customize the `headers-required` tag	Set the Boolean value of the `headers-required` tag to `True` only if you want to enable SOAP header processing for an intermediary web service. Incoming requests to EWS require elements. For example: `<add key="headers-required" value="false"/>` For more information about using SOAP headers for requests sent to an intermediary web service, see "Using SOAP Headers for Requests Sent to Intermediary Web Services" on the next page.
Customize the `send-request-body` tag	Set the Boolean value of the `headers-required` tag to `True` only if you want to enable SOAP header processing for an intermediary web service. Incoming requests to EWS require header elements. For example: `<add key="send-request-body" value="false"/>` For more information about using SOAP headers for requests sent to an intermediary web service, see "Using SOAP Headers for Requests Sent to Intermediary Web Services" on the next page.
Customize the `use-response-body` tag	Set the Boolean value of the `send-request-body` tag to `True` only if you want to enable SOAP header processing for an intermediary Web service. If this Boolean value is set to `True`, it specifies that the SOAP body should be sent to the intermediary, where it can be processed and then returned to EWS. For example: `<add key="use-response-body" value="false"/>` For more information about using SOAP headers for requests sent to an intermediary web service, see "Using SOAP Headers for Requests Sent to Intermediary Web Services" on the next page.

To	Do this

Customize the `OutputExtensions` element	Specify the output extension for your output format.
	
	The `Web.config` file lists the default extensions for each output in the `<OutputExtensions>` element. You can change the default extension attribute listed for your output, but the output name is internal and is used to determine the extension of a given output type. To prevent errors, the output name attribute should remain unchanged.
	
	The `<OutputExtensions>` element is defined in the `Web.config` file as follows:
	
	`<OutputExtensionGroup>`
	   `<OutputExtensions>`
	      `<add output="afp" extension="afp"/>`
	      `<add output="postscript" extension="ps"/>`
	      `<add output="metacode" extension="meta"/>`
	      `<add output="pdf" extension="pdf"/>`
	      `<add output="pcl" extension="pcl"/>`
	      `<add output="ijpds" extension="ijp"/>`
	      `<add output="vps" extension="vps"/>`
	      `<add output="rtf" extension="rtf"/>`
	      `<add output="html" extension="htm/>`
	      `<add output="dxf" extension="dxf"/>`
	      `<add output="ppml" extension="ppml"/>`
	      `<add output="tiff" extension="tiff"/>`
	      `<add output="3211LD" extension="xml"/>`
	      `<add output="vipp" extension="vipp"/>`
	      `<add output="powerpoint" extension="ppt"/>`
	      `<add output="vdx" extension="vdx"/>`
	      `<add output="mibf" extension="mibf"/>`
	      `<add output="content-xml" extension=xml"/>`
	      `<add output="live" extension="dlf"/>`
	      `<add output="pdf-a" extension="pdf"/>`
	      `<add output="edgar-html" extension="htm"/>`
	      `<add output="screen" extension="top"/>`
	      `<add output="multi-channel" extension="xml"/>`
	      `<add output="docx" extension="docx"/>`
	      `<add output="zpl" extension="zpl"/>`
	      `<add output="report" extension="rpt"/>`
	   `</OutputExtensions>`
	`</OutputExtensionsGroup>`
	
	Report file output is available only on EWS version 2.0.
	
	Note: When using output queues, the engine writes the output files to the working directory specified in the `Web.config` file. Although the engine uses the file name specified on the output queue as the output file name, the engine ignores the file path specified on the output queue. In the case that output is written to the server and not returned in the request, the engine writes the output to the location specified in the `<OutputDirectory>` element of the `Web.config` file.

Using SOAP Headers for Requests Sent to Intermediary Web Services

You can use the `RequestFilter` settings to include SOAP headers in your request. Including SOAP headers in your SOAP requests allows the request to be processed by an intermediary web service. The intermediary web service you use must be able to receive a SOAP request. SOAP headers for intermediary web services are commonly used for two purposes:

- **Security**—Some SOAP requests might require authentication and/or authorization from an intermediary. SOAP headers are forwarded to the intermediary for security processing.

  The SOAP body must be authorized, and then the body is sent on to the intermediary. The intermediary response is sent to the engine in place of the original SOAP body. EWS does not require any additional security specification since it passes the header directly to the intermediary service.

- **Pre-processing**—Some SOAP requests might require pre-processing on the body of the SOAP request. Preprocessing might or might not be done in conjunction with security. The SOAP headers and the SOAP body are forwarded to the intermediary.

  The intermediary response is sent to the engine, replacing the original SOAP body. For example, the intermediary could be used to process encrypted requests, and return the (decrypted) SOAP body to EWS for engine processing.

  The intermediary is required to return a response to EWS. The lack of a response is processed as an error condition. If an error occurs during intermediary processing, the intermediary must respond with a SOAP fault. The SOAP fault must be sent to the original sender.

Note: For security handling errors, including authentication failure, refer to the fault conventions found in Chapter 12 of the *Oasis WS-Security 1.0* specification.

5.2.2 Configuring the EWS Client

After you configure the server side of EWS, you must program a client to handle the request and response process that comprises the engine service portion of EWS. Although you can program a client to meet your individual needs, you can apply information provided in the following sections to develop a generic client for EWS.

Depending on the platform that you use, the EWS installation files (`EWSJ2EE_<version #>.zip` and `EWSDotNet_<version #>.zip`) contain several folders and files where you can reference sample client code to assist you in the development of the client side that best fits your needs.

If you are configuring the client side of EWS on the J2EE platform, you can access these samples in the `EWSJ2EE_<version #>.zip` file:

- `EngineClient.zip`
- `Client.zip`

If you are configuring the client side of EWS on the .NET platform, you can access these samples in the `EWSDotNet_<version #>.zip` file:

- `EWSClient.zip`
- `Client.zip`

This section discusses the following topics:

- "EWS Proxy Client Service for J2EE" below
- "Configuring an EWS Proxy Client for .NET" below

EWS Proxy Client Service for J2EE

The EWS proxy is a framework that applications can use to request preview documents from EWS. The proxy provides a simple API that you use to specify the preview request parameters required for generating previews. The proxy performs the service request and response handling. In addition, the proxy makes the preview document and associated metadata available to the application.

The EWS proxy requires the following Java 2 platform libraries:

- **Java API for XML Processing (JAXP) Version 1.3.1**—This library is a part of the Sun Java Web services development pack.
- **SOAP with Attachments API for Java (SAAJ) Version 1.2.2**—This library is a part of the Sun Java Web services development pack.
- **Java APIs for WSDL (JWSDL) Version 1.4**—This library is a separate download.
- **A JAX-WS compatible client creation tool**—You can use wsimport or any other JAX-WS compatible client-creation tool to create the client-side of EWS.

After you have acquired the Java 2 platform libraries, you can establish an EWS proxy client in the J2EE environment.

Configuring an EWS Proxy Client for .NET

To configure an EWS proxy client for .NET, you must complete the following steps:

1. Add a web reference to EWS.

   The `EWSComposeRequest`, `EWSComposeRequestHeader`, and `EWSComposeResponse` primary data classes, along with their supporting data classes, are now available.

2. Create an instance of `EWSComposeRequest` and populate the required properties. These can include properties such as `DriverFile`, `EngineOptions`, and so on.

3. If you want to send headers, create an instance of `EWSComposeRequestHeader` and populate it as in step 2.

4. Associate the instance of `EWSComposeRequestHeader` that you created in step 3 with EWS.
5. Invoke the service passing the `EWSComposeRequest` object that you created in step 2. This returns a `EWSComposeResponse` instance.

Now you can access properties of interest from the returned `EWSComposeResponse` object, such as `EngineMessage`, `Headers`, and so on.

5.2.3 Configuring SOAP Requests to Return EWS Responses

EWS communicates over the web using SOAP protocol and a request/response process. A valid request contains all of the information that the engine service and the engine require to compose and deliver documents to customers. As is true for the production of other applications in Exstream, the engine must have the required information to produce an EWS application, including driver file information, package file information, and output information. The SOAP request contains a set of specific and optional tags that contain all of the information necessary to produce the requested output. The engine service uses the information contained within the SOAP request to create a control file on-demand.

EWS uses a document call type for both input and output. Both the input and the output to the web service are Base64 encoded. The customer driver file used with this service can be any valid data format. As you create your SOAP request, remember that when you submit the SOAP request, line feeds and carriage returns are typically removed automatically upon submission. If requests still contain line feeds or carriage returns after submission, the request fails and a SOAP fault is generated.

For information about document call type parameters, see “Configuring a Call Type” on page 232.

Configuring the SOAP Request With Specific Request Tags

You must include specific tags in a SOAP request. Missing required tags result in an exception and the engine run is not completed. Tags are described in the Types section of the SOAP service's WSDL. The following table describes the tags you can configure.

Note: If you upgrade to Exstream version 7.0 or later and use .NET, make sure that you change the `<Driver>` tag content from XML content to a Base64 encoded string. SOAP requests that contain XML content and that are sent to newer versions of Exstream cause the request to fail.

SOAP request tags for J2EE

To	Use this tag	Tag Attributes
Add the root tag	`<Compose>`	None
Begin a SOAP request	`<EWSComposeRequest>`	You must include the following child tags in the `<EWSComposeRequest>` tag: • `driver` • `engineOptions` • `pubFiles` • `pubFile` Optionally, you can include the following child tags: • `outputFile` • `includeHeader` • `includeMessageFile` • `FileReturnRegEx`
Identify the customer driver file you want to use with your SOAP response	`<driver>`	The `<driver>` tag requires the `driver` and `fileName` child tags. The value of the driver attribute is a Base64 encoded string. The value of the `fileName` child tag is the exact name of the customer driver file packaged in the package file. The exact name of the customer driver file is the Production Data Source name given in Design Manager during the setup of the application. You must make sure the value of the `fileName` child tag does not contain characters that EWS considers invalid for file names. If the value of the `fileName` child tag contains invalid characters, the SOAP request fails. After the `<driver>` tag, the engine on both .NET and J2EE uses the Base64 encoded string to pass content to the engine, replacing the customer driver file.

SOAP request tags for J2EE, continued

To	Use this tag	Tag Attributes
Send engine switches as a part of your request to be included in the control file created at run time (optional)	`<engineOptions>`	You can send as many engine switches as required. For each switch, include the `<engineOption>` tag. To define the switch and options to use, use the following child tags: • `name`—The name of the engine switch • `value`—Specify options needed for the engine switch (optional). The DDAOUTPUT switch causes EWS not to issue its own DDAOUTPUT switch and EWS behaves as if the preview request is `<EWSComposeRequest include-header=" false">`. When you use the DDAFILEMAP switch, it modifies the path to the DDA binary and appends the path to the DDA directory used by your environment. Environment DDA directories are specified at the following location: • J2EE— `ews-config.xml` EWS ignores the following engine switches: • MESSAGEFILE • XMLMESSAGEFILE • MESSAGELEVEL • MSGSYMBOLICS • PACKAGEFILE • CONTROLFILE • FILEMAP that attempts to map the `fileName` attribute of the `<driver>` tag • DDAFILEMAP that attempts to map the `fileName` attribute of the `<driver>` tag
Specify the output type to be returned in the response	`<FileReturnRegEx>`	You can use regular expression syntax to specify the types of files you want to be returned. For example, if you want to return all files with PDF and AFP output type extensions, you would specify the following regular expression syntax: `<FileReturnRegEx>^.*.(pdf|afp)$</FileReturnRegEx>`
Specify whether to include header information in the response	`<includeHeader>`	If the value of `<includeHeader>` is set to `True`, SOAP headers are returned with the response. If the value of `<includeHeader>` is set to `False`, SOAP headers are not returned in the response.

SOAP request tags for J2EE, continued

To	Use this tag	Tag Attributes
Specify whether to include message file information in the response	`<includeMessageFile>`	If the value of `<includeMessageFile>` is set to `True`, the request returns the message file as inline Base64 encoded data in the response. If the value of `<includeMessageFile>` is set to `False`, the message file is not returned. If the message file is requested, but not generated, a SOAP fault is not issued.
Write a generated document to a file location instead of including it in the response (optional)	`<outputFile>`	You must include the following child tags with or without values for each • `fileName`—The name of the created file. If no file name is requested, EWS generates a file name. If you specify the name of an existing file, it is overwritten. • `directory`—The directory where you want the file to be created. EWS appends the argument for the `directory` attribute to the configured output file directory (`OutputDirectory`) to determine the final directory. If the final output file directory does not exist, EWS either creates it or returns a SOAP fault, depending on the `CreateOutputDirectories` setting. When the `<outputFile>` tag is used, all output is written to the specified location. You cannot write some output to the server and return some output in the response. When output is written to the server, all that will be returned in the response are references to the location of the output.
Identify the package file(s) that you want the engine to use	`<pubFiles>`	Use the `<pubFiles>` tag to specify the package file (s) that you want to include in the SOAP request. The `<pubFiles>` tag is a parent tag of the `<pubFile>` tag. EWS supports the following package file objects: • Applications • Campaigns • Documents • Non-Design objects • Single campaigns • Single documents

SOAP request tags for J2EE, continued

To	Use this tag	Tag Attributes
Identify the package file(s) name that you want the engine to use	`<pubFile>`	Use the `<pubFile>` tag to specify each package file that you want to include in the SOAP request. The engine reads package file names sequentially from the SOAP request. If you specify multiple package files in the SOAP request, then you must specify the main package file that you want the engine to use before you specify additional package file names. You must specify only package file names with the `.pub` file extension. Package file names are appended to the path for the package file directory.

SOAP request tags for .NET

To	Use this tag	Tag Attributes
Add the root tag	`<Compose>`	None
Begin a SOAP request	`<request>`	You must include the following child tags in the `<request>` tag: • `Driver` • `EngineOptions` • `PubFiles` • `PubFile` Optionally, you can include the following child tags: • `OutputFile` • `IncludeHeader` • `IncludeMessageFile` • `FileReturnRegEx`
Identify the customer driver file you want to use with your SOAP response	`<Driver>`	The `<Driver>` tag requires the `Driver` and `FileName` child tags. The value of the driver attribute is a Base64 encoded string. The value of the `FileName` child tag is the exact name of the customer driver file packaged in the package file. The exact name of the customer driver file is the Production Data Source name given in Design Manager during the setup of the application. You must make sure the value of the `FileName` child tag does not contain characters that EWS considers invalid for file names. If the value of the `FileName` child tag contains invalid characters, the SOAP request fails. After the `<Driver>` tag, the engine on both .NET and J2EE uses the Base64 encoded string to pass content to the engine, replacing the customer driver file.

SOAP request tags for .NET, continued

To	Use this tag	Tag Attributes
Send engine switches as a part of your request to be included in the control file created at run time (optional)	`<EngineOptions>`	You can send as many engine switches as required. For each switch, include the `<EngineOption>` tag. To define the switch and options to use, use the following child tags: • `Name`—The name of the engine switch • `Value`—Specify options needed for the engine switch (optional). The DDAOUTPUT switch causes EWS not to issue its own DDAOUTPUT switch and EWS behaves as if the preview request is `<Request include-header=" false">`. When you use the DDAFILEMAP switch, it modifies the path to the DDA binary and appends the path to the DDA directory used by your environment. Environment DDA directories are specified at the following location: • .NET— `Web.config` EWS ignores the following engine switches: • MESSAGEFILE • XMLMESSAGEFILE • MESSAGELEVEL • MSGSYMBOLICS • PACKAGEFILE • CONTROLFILE • FILEMAP that attempts to map the `FileName` attribute of the `<Driver>` tag • DDAFILEMAP that attempts to map the `FileName` attribute of the `<Driver>` tag
Specify the output type to be returned in the response	`<FileReturnRegEx>`	You can use regular expression syntax to specify the types of files you want to be returned. For example, if you want to return all files with PDF and AFP output type extensions, you would specify the following regular expression syntax: `<FileReturnRegEx>^.*.(pdf|afp)$</FileReturnRegEx>`
Specify whether to include header information in the response	`<IncludeHeader>`	If the value of `<IncludeHeader>` is set to `True`, SOAP headers are returned with the response. If the value of `<IncludeHeader>` is set to `False`, SOAP headers are not returned in the response.

SOAP request tags for .NET, continued

To	Use this tag	Tag Attributes
Specify whether to include message file information in the response	`<IncludeMessageFile>`	If the value of `<IncludeMessageFile>` is set to `True`, the request returns the message file as inline Base64 encoded data in the response. If the value of `<IncludeMessageFile>` is set to `False`, the message file is not returned. If the message file is requested, but not generated, a fault is not issued.
Write a generated document to a file location instead of including it in the response (optional)	`<OutputFile>`	You must include the following child tags with or without values for each • `FileName`—The name of the created file. If no file name is requested, EWS generates a file name. If you specify the name of an existing file, it is overwritten. • `Directory`—The directory where you want the file to be created. EWS appends the argument for the `Directory` attribute to the configured output file directory (`OutputDirectory`) to determine the final directory. If the final output file directory does not exist, EWS either creates it or returns a SOAP fault, depending on the `CreateOutputDirectories` setting. When the `<OutputFile>` tag is used, all output is written to the specified location. You cannot write some output to the server and return some output in the response. When output is written to the server, all that will be returned in the response are references to the location of the output.
Identify the package file(s) that you want the engine to use	`<PubFiles>`	Use the `<PubFiles>` tag to specify the package file (s) that you want to include in the SOAP request. The `<PubFiles>` tag is a parent tag of the `<PubFile>` tag. EWS supports the following package file objects: • Applications • Campaigns • Documents • Non-Design objects • Single campaigns • Single documents

SOAP request tags for .NET, continued

To	Use this tag	Tag Attributes
Identify the package file(s) name that you want the engine to use	`<PubFile>`	Use the `<PubFile>` tag to specify each package file that you want to include in the SOAP request. The engine reads package file names sequentially from the SOAP request. If you specify multiple package files in the SOAP request, then you must specify the main package file that you want the engine to use before you specify additional package file names. You must specify only package file names with the `.pub` file extension. Package file names are appended to the path for the package file directory.

The following example shows a sample request in EWS 2.0 for the J2EE environment:

```
<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:eng="urn:hpexstream-services/Engine">
   <soapenv:Header/>
   <soapenv:Body>
      <eng:Compose>
         <EWSComposeRequest>
         <driver>
         <driver>PERyaXZlckxpc3Q+PEN1c3RvbWV</driver>
         <fileName>DRIVER</fileName>
         <driver>
         <engineOptions>
            <name>RUNMODE</name>
            <value>PRODUCTION</value>
         </engineOptions>
         <FileReturnRegEx>^.*\.(pdf|afp)$</FileReturnRegEx>
         <includeHeader>True</includeHeader>
         <includeMessageFile>True</includeMessageFile>
         <outputFile>
            <directory>Universal</directory>
            <fileName>Statement</fileName>
         </outputFile>
         <pubFiles>
            <pubFile>Main.pub</pubFile>
            <pubFile>Sub1.pub</pubFile>
            <pubFile>Sub2.pub</pubFile>
         </pubFiles>
         <EWSComposeRequest>
      </eng:Compose>
   </soapenv:Body>
</soapenv:Envelope>
```

The following example shows a sample request in EWS 2.0 for the .NET environment:

```
<soapenv:Envelopexmlns:soapenv="http://schemas.xmlnssoap.org/soap/envelope/"
xmlns:tem="http://tempuri.org/"

xmlns:com="http://schemas.datacontract.org/2004/07/com.hpexstream.services.ew
s">
<soapenv:Header/>
   <soapenv:Body>
      <tem:Compose>
         <tem:request>
            <com:Driver>
               <com:Driver>RvbWVyRGF0YT4=</com:Driver>
               <com:FileName>DRIVER</com:Driver>
            <com:Driver>
            <com:EngineOptions>
               <com:EngineOption>
                  <com:Name>RUNMODE</com:Name>
                  <com:Value>Production</com:Value>
               </com:EngineOption>
                  <com:NamePRODUCTION</com:Value>
               </com:EngineOption>
                  <com:Name>FILEMAP</com:Name>
                  <com:Value>EMAILINIT,
                     C:\Projects\EWSTemp\80\Universal_email_
init.txt</com:Value>
               </com:EngineOption>
            </com:EngineOptions>
            <com:FileReturnRegEx>^.*\.(pdf|afp)$</com:FileReturnRegEx>
            <com:IncludeHeader>1</com:IncludeHeader>
            <com:IncludeMessageFile>1</com:IncludeMessageFile>
            <com:OutputFile>
               <com:Directory>Universal</com:Directory>
               <com:FileName>Statement</com:FileName>
            </com:OutputFile>
            <com:PubFiles>
               <com:PubFile>Main.pub</com:PubFile>
               <com:PubFile>Sub1.pub</com:PubFile>
               <com:PubFile>Sub2.pub</com:PubFile>
            </com:PubFiles>
         </tem:request>
      </tem:Compose>
   </soapenv:Body>
</soapenv:Envelope>
```

5.3 Testing the Exstream Engine as a Web Service

EWS acts as a wrapper for the Exstream engine. The wrapper makes the Exstream engine act as a Web service. You can run sample applications through EWS to test your configuration before using EWS in production. Testing EWS requires that you validate the responses that EWS returns based on your configuration.

5.3.1 Validating EWS Responses

EWS generates output in two parts: header information and composed engine output. To test your configuration, you can validate the header information and engine output results.

This section discusses the following topics:

- “Validating SOAP Header Information in the EWS Response” below
- “Validating PDL Return Codes” on page 97
- “Composed Engine Output” on page 98

Validating SOAP Header Information in the EWS Response

The content of the SOAP response is the composed output headers. The SOAP request tag attributes below represent the J2EE environment. The tag order of the output is not guaranteed, nor is the inclusion or exclusion of any tag. Clients must parse the output using explicit tag names.

- If the `include-header` tag is set to `false`, the output contains empty header tag sets.
- If the `include-header` tag is set to `true`, then you receive complete header information in the response.

The following table explains the tags used in the SOAP header in the J2EE environment.

J2EE SOAP header tags

Tag	Description
`<ComposeResponse>`	The root tag
`<version>`	The header structure version number

J2EE SOAP header tags, continued

Tag	Description
`<PDL>`	The output driver used For more information about `<PDL>` tag return codes, see "Validating PDL Return Codes" on the next page.
`<outputLength>`	The length in bytes of the output
`<messageLength>`	The length in bytes of the message (this placeholder is currently not used)
`<userData>`	The value of 'SYS_DDAOutputUserData' For more information about 'SYS_DDAOutputUserData', see "DDA Routine Output Headers" on page 40.
`<userDataLength>`	The length in bytes of the user data
`<pageCount>`	The number of pages in the composed output
`<returnCode>`	The return code from the engine run
`<fileType>`	The file type of the composed output
`<defaultExtension>`	The file extension of the composed output

The following table explains the tags used in the SOAP header in the .NET environment.

.NET SOAP header tags

Tag	Description
`<ComposeResponse>`	The root tag
`<ComposeResult>`	The container tag for the entire response
`<Header>`	The root tag for run results
`<Version>`	The header structure version number
`<PDL>`	The output driver used For more information about `<PDL>` tag return codes, see "Validating PDL Return Codes" on the next page.
`<OutputLength>`	The length in bytes of the output
`<MessageLength>`	The length in bytes of the message (this placeholder is currently not used)
`<UserData>`	The value of 'SYS_DDAOutputUserData' For more information about 'SYS_DDAOutputUserData', see "DDA Routine Output Headers" on page 40.
`<UserDataLength>`	The length in bytes of the user data
`<PageCount>`	The number of pages in the composed output

.NET SOAP header tags, continued

Tag	Description
`<ReturnCode>`	The return code from the engine run
`<FileType>`	The file type of the composed output
`<DefaultExtension>`	The file extension of the composed output

Validating PDL Return Codes

The `<PDL>` tag returns a code to represent the output driver used. The following table lists the return codes and their meanings.

PDL return codes

PDL code	Output driver
1	AFP
2	PostScript
3	Metacode
4	PDF
5	PCL
6	IJPDS
7	VPS
8	RTF
9	HTML
10	Composed XML
11	PPML
12	TIFF
13	Line Data
14	VIPP
15	PowerPoint
16	VDX

PDL return codes, continued

PDL code	Output driver
17	MIBF
18	XML Content
19	DLF
20	PDF/A
21	Edgar HTML
22	Screen (TOP)
23	XML (Multi-channel)
24	Word (2007/2010)
26	HTML (email)
27	ZPL
28	Empower
29	PDF/VT
401	Report files

Composed Engine Output

Composed output is always returned in the body of the SOAP response. In the J2EE environment, you have the option to send the customer driver file to the service as an attachment. In the .NET environment, the customer driver file is always sent inline. The output is returned in a Base64 format; consequently, the client is required to complete Base64 decoding on this output.

Chapter 6: The Email Delivery Connector

The Email Delivery Connector lets you communicate with customers by sending documents through email or by sending Short Message Service (SMS) text messages. A platform-independent utility, the Email Delivery module can be implemented as a DDA connector, or as a standalone executable program.

You can customize production workflow to allow for automated recovery, multiple sending attempts, and the capture of extensive auditing information in communication logs. For example, suppose multiple customers need updated bank statements with varying off-cycle dates (a 15-day period as opposed to a traditional 30-day period). Customers log in to their online banking websites and enter the necessary information into the appropriate fields on the user interface. Additionally, customers request that their documents be sent as emails with attached PDFs of the bank statement along with an additional document to be sent with a regular monthly bank statement by way of a separate email. Your company uses Exstream with the Email Delivery Connector installed as a DDA connector and as a standalone executable. With each customer request, the Email Delivery Connector lets you complete processing by circumventing traditional output channels and using your enterprise system and the production environment to dynamically produce one bank statement on demand (attached in the email as a PDF) while the other bank statement is produced in batch mode and sent to the customer as an email during the normal delivery cycle.

This chapter discusses the following topics:

- “Preparing the Email Delivery Connector” below
- “Configuring the Email Delivery Connector” on the next page
- “Email Delivery Connector Parameters” on page 108
- “Testing the Email Delivery Connector” on page 117

6.1 Preparing the Email Delivery Connector

To prepare the Email Delivery Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the `EmailConnector.zip` file.
2. Extract the contents of the `EmailConnector.zip` file to the directory that contains the Exstream engine.
3. Set the appropriate environment variables for your operating system.

For example, if you are operating on Windows, then you must set the following environment variables:

- PATH—Reference the path name for the directory that contains `jvm.dll`.
- CLASSPATH—Reference the path name for the Email Delivery Connector installation directory.

4. For best results, make sure any applications that include test files package and run in batch mode with no errors.

6.2 Configuring the Email Delivery Connector

To configure the Email Delivery Connector, you must complete the following tasks:

1. "Configuring the Email Delivery Connector in Design Manager" on the next page
2. "Setting Up the Configuration File" on page 103
3. "Configuring the Customer Report File" on page 103
4. "Setting Up Your Log File" on page 105
5. "Sending File Attachments Using Output Queues" on page 106
6. "Creating an Email Message Body Using an Initialization File" on page 107

6.2.1 Configuring the Email Delivery Connector in Design Manager

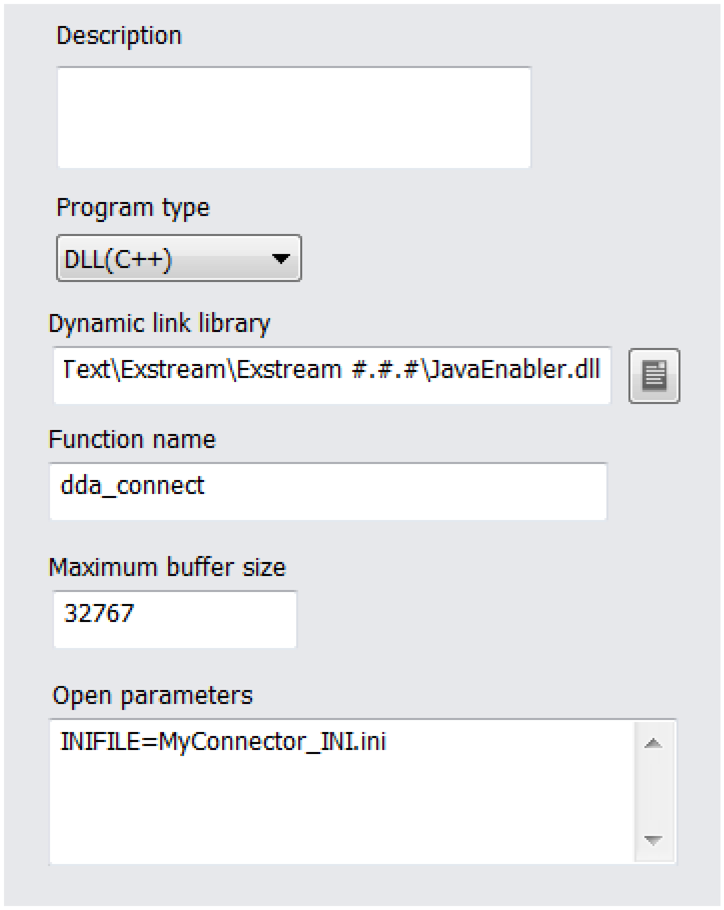
When you configure the Email Delivery Connector in Design Manager, you must first create a connector object in the Design Manager Library. Then, you must assign the Email Delivery Connector to all applicable data files, outputs, and other objects that relate to the applications that you will use with the connector.

To configure the Email Delivery Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the Java Enabler, or click 🗎 to go to the file: For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Email Delivery Connector Parameters" on page 108.

Example of the completed connector object properties for the Email Delivery Connector



Description

Program type
DLL(C++)

Dynamic link library
Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name
dda_connect

Maximum buffer size
32767

Open parameters
INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Dynamic Data Access" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning a connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to an output.

   For more information about assigning a connector to an output, see "Assigning a Connector Object to an Output" on page 27.

4. Assign the connector to an output queue.

   For more information about assigning a connector to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

6.2.2 Setting Up the Configuration File

After you have created a connector object in Design Manager, you must set up the configuration file. The configuration file is named `emaildelivery.defaults` and is available in the following location of the installation directory: `\META-INF\emaildelivery.defaults`.

To set up the configuration file, you must complete the following steps:

1. In your CLASS parameter, enter `com/exstream/emaildelivery/application/EmailDeliveryDDA`.
2. Open the `emaildelivery.defaults` in a text editor.
3. In your CLASS parameter, enter `com/exstream/emaildelivery/application/EmailDeliveryDDA`.
4. In your CLASSPATH parameter, load the file `/META-INF/emaildelivery.properties.defaults`.
5. In your working directory, load the file `emaildelivery.properties`.
6. Set the parameters for the configuration file.

For more information about Email Delivery Connector parameters, see "Email Delivery Connector Parameters" on page 108.

6.2.3 Configuring the Customer Report File

After you set up the Email Delivery Connector configuration file, you must set up a report data file in each application that you will use with the Email Delivery Connector. In the report data file that you create, the report file fields must be specified in the same order as you specified in the `emailDelivery.report.map` parameter of the configuration file.

The information in the report data file must represent a one-to-one correspondence to the definitions of the `emailDelivery.report.map` parameter of the configuration file. The following table shows how the definitions of the `emailDelivery.report.map` parameter match the definitions of the report file.

Email Delivery Connector report file mapping definitions

emailDelivery.report.map Parameter Definitions	Report Data File Definitions
`smtp.sender`	`"From"`
`smtp.rcpt`	`"To"`
`smtp.subject`	`"Subject"`
`smtp.body`	`"Text"`

Email Delivery Connector report file mapping definitions, continued

emailDelivery.report.map Parameter Definitions	Report Data File Definitions
`smtp.attachments`	`"Attachments"`

For more information about the `emailDelivery.report.map` parameter, see "System Configuration Parameters" on page 108.

To configure the customer report file, you must complete the following steps:

1. In Design Manager, in the Library, go to **Data Files** heading.
2. In the Library, right-click the type of data file that you want to create. For example, select **SBCS Data File** for a new SBCS data file.

   The **New Data File** dialog box opens.
3. In the **Name** box, enter a name.
4. In the **Description** box, enter a description (optional).
5. From the **File type** drop-down list, select **Report file**.
6. From the **File format** drop-down list, select **Delimited data file**.

   The report file opens in the Property Panel for you to define.
7. Click the **Advanced** tab.
8. From the **IO time** drop-down list, select **At end of customer, before campaigns**.
9. Click the **Production Data Source** tab.
10. From the **Type** drop-down list, select **Connector**.
11. Next to the **Connector** box, click .
12. Select the name of the DDA connector that you created.
13. Click **OK**.
14. Specify additional settings in the Property Panel as necessary.
15. In the Edit Panel, open the report data file that you want to use with the Email Delivery Connector.
16. Map the appropriate variables you want to provide to the fields of the report.
17. Click to save the report data file settings.
18. Add the report data file to your application.

For more information about defining and mapping data files, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

6.2.4 Setting Up Your Log File

Logging lets you configure a record log that you can use to monitor the behavior of applications produced by the engine and the Email Delivery Connector. You can set up logging to prevent errors in your output. To use logging, configure the standard `log4j` configuration file provided by the Apache community. All parameter definitions follow the standard `log4j.properties` file specifications. You must extract, modify, and then compress the `log4j.properties` file back to the JAR file.

To set up your log file, you must configure the standard `log4j.properties` file located inside the JAR executable archive:

1. Extract the contacts of the `log4j.properties` file.

   The `log4j.properties` file is located in the `EmailConnector.jar` file of the `EmailConnector.zip` file.

2. Modify the following `log4j.properties` parameters according as necessary:

   log4j.properties configuration file parameters for the Email Delivery Connector

Parameter	Description
`log4j.rootLogger`	The `log4j.rootLogger` parameter specifies which loggers to use in the logging process and the minimal threshold of the loggers (specified as a first parameter).
`log4j.appender.stdout.threshold`	The `log4j.appender.stdout.threshold` parameter specifies the threshold for the console (`stdout`) logger and which messages to show on the console.
`log4j.appender.debugFile.File`	The `log4j.appender.debugFile.File` parameter specifies the file name of the debug-logger file that contains all the messages from the module).
`log4j.appender.debugFile.threshold`	The `log4j.appender.debugFile.threshold` parameter specifies the threshold of the debug-logger file.
`log4j.appender.errorFile.File`	The `log4j.appender.errorFile.File` parameter specifies the name of the file in which to log the error and warning messages. This file must already exist.
`log4j.appender.errorFile.threshold`	The `log4j.appender.errorFile.threshold` parameter specifies the threshold of the error-logger file.

3. Compress the `log4j.properties` file back to the `EmailConnector.jar` file.

4. Back up your `log4j.properties` file before you upgrade the Email Delivery Connector module, so you can reuse it with the later version.

For more information about the `log4j.properties` file specifications, refer to the log4j documentation at http://logging.apache.org.

6.2.5 Sending File Attachments Using Output Queues

The Email Delivery Connector lets you create an attachment, typically in the form of a PDF, that you can use to send documents to customers in an email. To send file attachments, you must configure an output queue for each attachment. The Email Delivery Connector can collect data files from any accessible location on the file system, so you can define where to place the generated files.

To send file attachments using output queues, you must complete the following steps:

1. In Design Manager, create a variable to control the file naming of the attachment file and the output queue. Use a variable that provides unique file names (for example, a file name based on the 'SYS_CustomerInRun' variable).

   For information about defining variables to control file naming, see *Creating Output* in the Exstream Design and Production documentation.

2. In Design Manager, in the Library, create a new output queue for each attachment that you will use with the Email Delivery Connector.

   For more information about creating and defining output queues, see *Creating Output* in the Exstream Design and Production documentation.

3. From the Library, drag the output queue to the Property Panel.
4. On the **Basic** tab, in the **Variable for file naming** box, click V .
5. Select the attachment file name variable you created, and click **OK**.
6. Click the **Breaks** tab.
7. In the **Customers** box, enter `1`.
8. Click 💾 to save the output queue settings.
9. Add the output queue to your application. If you have multiple output queues for multiple attachments, you must place the output queues in your application in the same order that the attachments are listed in the report data file.

Note: Output files must be properly named and output queues for attachments must be broken after each customer so the application logic can determine the correct path to the attachment.

6.2.6 Creating an Email Message Body Using an Initialization File

Using Exstream and the application template that comes with the Email Delivery Connector, you can dynamically compose message body text (or any textual information) using predefined functions and an initialization file. The initialization file that you use to create the email message body is a plain text file that contains simple placeholder tags to be replaced during composition by actual variable content.

For example:

```
Hello <1>,
I'm very happy to inform you that you've been selected by the jury to
participate
in an annual conference about <2>. I'm sending you details in the attached
message.
Best regards,
<3>
```

The special marks `<1>`, `<2>`, and `<3>` are placeholder tags to be replaced by variable content during composition.

To create an email message body using an initialization file, you must complete the following steps:

1. Create an initialization file that contains the email message body and simple placeholder tags that you will define using variables.
2. In Design Manager, in the Library, go to **Data Files** heading.
3. In the Library, right-click the type of data file that you want to create. For example, select **SBCS Data File** for a new SBCS data file.

   The **New Data File** dialog box opens.
4. In the **Name** box, enter a name.
5. In the **Description** box, enter a description (optional).
6. From the **File type** drop-down list, select **Report file**.
7. From the **File format** drop-down list, select **Delimited data file**.
8. From the Library, drag the initialization data file to the Property Panel.
9. On the **Basic** tab, from the **How data is identified in the data file** drop-down list, select **Each record in the file has the same mapping**.
10. Specify additional settings in the Property Panel as necessary.
11. In the Edit Panel, open the initialization file.

12. Map the content of the file to a growing array with line length set to a value that is more than the expected maximum length of the email message body.
13. Click 💾 to save the initialization file settings.
14. Add the initialization file to the application.
15. Create a formula string variable for the body text.
16. Create an array variable to hold the replacement values. Each item in the array corresponds to one replacement mark.
17. Create a string variable from the string array in the initialization file.
18. Test the new string variable and map it in the customer report file layout.

6.3 Email Delivery Connector Parameters

You must define the parameters for the Email Delivery Connector when you set up the configuration file. The installation ZIP file for the Email Delivery Connector contains the configuration file (`emaildelivery.defaults`) that you can use to configure the parameters for the Email Delivery Connector. The `emaildelivery.defaults` configuration file is available in the installation directory: `\META-INF\emaidelivery.defaults`.

To define basic configuration, error queue configuration, and SMTP delivery configuration in the Email Delivery Connector, use the parameters in the following sections:

- "System Configuration Parameters" below
- "Customer Report File Mapping Symbols" on page 110
- "Error Queue Parameters" on page 114
- "SMTP Configuration Parameters" on page 116

6.3.1 System Configuration Parameters

The following table identifies the system configuration parameters.

Email Delivery Connector system configuration parameters

Parameter	Description
`emailDelivery.attachmentsHandling=move|copy|reference`	The `emailDelivery.attachmentsHandling` parameter specifies how to handle email attachments. Use one of the following values: • `move`: Move attachments to the error queue. • `copy`: Copy attachments to the error queue. • `reference`: Reference the original attachment in the error queue. **Example:** `emailDelivery.attachmentsHandling=copy` The default value for this parameter is `move`.
`emailDelivery.encoding`	The `emailDelivery.encoding` parameter specifies the default character set encoding to use if you do not specify separate encoding for the email address or body. **Example:** `emailDelivery.encoding=file.encoding` The default value for this parameter is `file.encoding`. For more information about these encoding options, see “SMTP Address String” on page 112 or “Customer Report File Mapping Symbols” on the next page.
`emailDelivery.agent.name`	The `emailDelivery.agent.name` parameter specifies the mail delivery agent name to use for delivery. The only valid value for the Email Delivery Connector is `SMTPDelivery`. **Example:** `emailDelivery.agent.name=delivery agent name` There is no default value for this parameter.
`emailDelivery.report.map`	The `emailDelivery.report.map` parameter specifies the customer report file mapping string. **Example:** `emailDelivery.report.map=smtp.sender,smtp.rcpt,std.subject,smtp.htmlfile,smtp.attachments` There is no default value for this parameter. For information about customer report file mapping options, see “Customer Report File Mapping Symbols” on the next page.
`emailDelivery.customerFile`	The `emailDelivery.customerFile` parameter specifies the customer report file used when delivering messages. **Example:** `emailDelivery.customerFile=C:\Demos\EmailDDA\OUT\111111111.html` There is no default value for this parameter.

6.3.2 Customer Report File Mapping Symbols

You must use the `emailDelivery.report.map` system parameter to specify the mapping string in the customer report file.

Customer report file mapping describes the file format (and entries contained in it) of a report file created by the engine run and used by Email Delivery to compose email messages. The mapping string is a comma-delimited text row that contains mapping symbols. It can be longer than the actual report file mapping, but cannot be shorter.

The following table identifies the generic mapping symbols.

Note: Each mail delivery agent has its own set of mapping symbols. Mail delivery agents other than Email Delivery might not support the generic symbols.

Generic mapping symbols used for the Email Delivery Connector

Symbol	Description
`std.sender`	The `std.sender` parameter specifies the entry as a sender. It can contain any information that the target mail delivery agent understands as a sender (for example, email address for email delivery, fax number for fax delivery, and so on). There is no default value for this parameter.
`std.recipient`	The `std.recipient` parameter specifies the entry as a recipient. It can contain any information that the target mail delivery agent understands as a recipient (for example, email address for email delivery, fax number for fax delivery, and so on). There is no default value for this parameter.
`std.subject`	The `std.subject` parameter specifies the entry as a subject (text). There is no default value for this parameter.
`std.text`	The `std.text` parameter specifies the entry as the text of the message. There is no default value for this parameter.
`std.encoding`	The `std.encoding` parameter specifies the entry as the character set encoding to use for the message body. This value overrides the default character set encoding specified in `emailDelivery.encoding`. There is no default value for this parameter.
`std.ignore`	The `std.ignore` parameter specifies the entry as a field to ignore and use as a placeholder for reserved, empty, or unused fields. There is no default value for this parameter.

The following table identifies the special mapping strings for customer report file mapping included with the Email Delivery Connector.

Email Delivery Connector mapping symbols

Symbol	Description
`smtp.sender`	The `smtp.sender` parameter contains the SMTP address string for the sender's email address for the current message. Although it can contain a list of multiple addresses, only the first entry is considered to be the sender's address. There is no default value for this parameter. For more information about configuring SMTP address strings, see "SMTP Address String" on the next page.
`smtp.replyto`	The `smtp.replyto` parameter contains the SMTP address string for the email address that populates in the "To" field when the recipient replies to the current message. In most cases, the SMTP address string contains the sender's email address. The SMTP address string can contain a list of multiple addresses. There is no default value for this parameter.
`smtp.rcpt`	The `smtp.rcpt` parameter contains the SMTP address string for the recipient's email address. This string can contain a list of multiple addresses. There is no default value for this parameter.
`smtp.htmlbody`	The `smtp.htmlbody` parameter contains the HTML body of the email message. If you set this mapping symbol, the email message is sent with text/HTML body content-type, and the client email software can show it as an HTML page. There is no default value for this parameter.
`smtp.htmlfile`	The `smtp.htmlfile` parameter contains the file path to the HTML file that contains the HTML body of the email message. There is no default value for this parameter.
`smtp.attachments`	The `smtp.attachments` parameter contains the SMTP attachment string for the message attachment(s). The string can be a list of multiple attachments. There is no default value for this parameter. For more information about configuring SMTP attachment strings, see "SMTP Attachment String" on the next page.
`smtp.cc`	The `smtp.cc` parameter contains the SMTP address string for the email address of recipients who receive a carbon-copy of the current message. The recipients who receive a carbon-copy are listed on the email header. This string can contain a list of multiple addresses. There is no default value for this parameter.
`smtp.bcc`	The `smtp.bcc` parameter contains the SMTP address string for the email address of recipients who receive a blind carbon-copy of the current message. The recipients who receive a blind carbon-copy are masked in the email header. This string can contain a list of multiple addresses. There is no default value for this parameter.

You can specify the value of the `emailDelivery.report.map` parameter using a combination of the generic and specific Email Delivery Connector mapping symbols, such as `std.ignore`, which specifies an entry as a field to ignore. For example, the `std.ignore` mapping symbol lets you use a placeholder for a customer's account number.

The following example uses a combination of mapping symbols, some of which are required to properly map a customer report file:

```
emailDelivery.report.map=std.ignore,smtp.sender,smtp.replyto,smtp.rcpt,
std.ignore,std. subject,smtp.attachments,std.text,std.ignore,smtp.cc,smtp.bcc
```

The value of the `emailDelivery.report.map` parameter has the following matching customer report file layout:

```
"Num","Sender","ReplyTo","Recipient","Client name","Subject",
"Attachments","Body","Encoding","CC","BCC"
```

SMTP Address String

Address strings are specially formatted strings that contain all of the information required for an individual or email addresses. You use SMTP address strings to specify the values of the `smtp.sender`, `smtp.replyto`, `smtp.rcpt`, `smtp.cc`, and `smtp.bcc` mapping symbols. The format of a single address string is as follows:

```
[Name]$[Address]$[Encoding]
```

SMTP address string fields

Field	Description
`Name`	The `Name` field specifies the sender's name, which can contain any characters except carriage returns and dollar signs.
`Address`	The `Address` field specifies the RFC822-correct email address.
`Encoding`	The `Encoding` field specifies the character set encoding to use for the full name in the email address (optional). This value overrides the default character set encoding specified in `emailDelivery.encoding`.

To specify more than one address in the string, use two dollar signs ($$) as the delimiter. For example:

```
John Doe$jdoe@company.com$$Jane Doe$jane@othercompany.com$utf-8
```

The address string above contains the following two email addresses:

- John Doe `<jdoe@company.com>`, to be encoded using default encoding
- Jane Doe `<jane@othercompany.com>`, to be encoded using "utf-8" encoding

SMTP Attachment String

Attachment strings are specially-formatted strings that contain all of the information required for email attachment files. If you need to attach files to the message, you use attachment strings to specify the values of the `smtp.attachments` mapping symbol. The format of an attachment string is as follows:

```
[MIME];[FileSystemName];[AttachmentName];[AttachmentsHandling]
```

SMTP attachment string fields

Field	Description
`MIME`	The `MIME` field specifies the MIME type of the attachment (for example, use "`application/pdf`" for PDF documents).
`FileSystemName`	The `FileSystemName` field specifies the path to the attachment file on the file system. The file must already exist and the Email Delivery Connector must have read access.
`AttachmentName`	The `AttachmentName` field specifies the file name to be presented to the recipient as the file name of the attachment. Using this option, you can set the same file name for all clients, even if their `FileSystemName` fields are different. For example, if the file attached from the server is "`inv065782.pdf`," the recipient receives an attachment named "`invoice.pdf`."
`AttachmentsHandling=move|copy|reference`	The `AttachmentsHandling` field specifies how to handle email attachments. Use one of the following values: • `move`: Move attachments to the error queue. • `copy`: Copy attachments to the error queue. • `reference`: Reference the original attachment in the error queue.

To specify more than one attachment, use two semicolon signs (;;) as the delimiter. For example:

```
application/pdf;c:\output\cli_02.pdf;order.pdf;;application/jpeg;
c:\output\cli_02.jpg;picture.jpg
```

The string above contains the following two attachments:

- A PDF attachment located at `c:\output\cli_02.pdf`, to be attached to the message with the file name `order.pdf`
- A JPEG attachment located at `c:\output\cli_02.jpg`, to be attached to the message with the file name `picture.jpg`

6.3.3 Error Queue Parameters

The Email Delivery Connector error queue encapsulates error recovery procedures. You can configure the Email Delivery Connector so that when the delivery of a message to the SMTP server fails, the message, its file attachments, and information about the message move to the error queue directory.

If you configure the Email Delivery Connector to retry delivery, error queue items are redelivered in chronological order. The Email Delivery Connector retries delivery for each message in the error queue until delivery is successful or the message reaches the maximum limit for delivery retries.

After reaching the maximum retry limit, the Email Delivery Connector considers the message delivery to have failed. If you have a logging system configured, then the Email Delivery Connector records the failure state in the log file, and deletes the message entry and all attachments from the error queue.

Email Delivery Connector error queue configuration parameters

Parameter	Description
`emailDelivery.errorQueue.use`	The `emailDelivery.errorQueue.use` parameter specifies whether to move composed, undelivered messages to the error queue for later redelivery attempts. Select from one of the following options: • `True`—The `True` value moves the messages to an error queue. • `False`—The `False` value deletes the messages. **Example:** `emailDelivery.errorQueue.use=False` The default value for this parameter is `False`.

Email Delivery Connector error queue configuration parameters, continued

Parameter	Description
`emailDelivery.errorQueue.retry`	The `emailDelivery.errorQueue.retry` parameter specifies whether to redeliver messages stored in the error queue. Select from one of the following options: • `True`—The `True` value attempts to redeliver messages in the error queue. • `False`—The `False` value does not attempt to redeliver messages. **Example:** `emailDelivery.errorQueue.retry=False` The default value for this parameter is `False`.
`emailDelivery.errorQueue.maxRetries`	The `emailDelivery.errorQueue.maxRetries` specifies the maximum number of retries to make before the message is considered to fail. **Example:** `emailDelivery.errorQueue.maxRetries=3` The default value for this parameter is `3`.

For information about configuring the logging system for the Email Delivery Connector, see "Setting Up Your Log File" on page 105.

Error Queue Maintenance

The Email Delivery Connector error queue is an automated part of the delivery system that contains data in binary format. It does not store human readable files. You can, however, purge the error queue or delete directories with message records. To completely clean the error queue, delete the content of the error queue directory root.

Format of Message Record Directories in the Error Queue

The error queue directory root can contain multiple directories with names in a specific format that matches the message Serial ID (unique identifier of each message) reported in the log file by the Email Delivery Connector. In SMTP delivery, these directories have the following naming convention:

```
SMTP-1169507266531-1
```

The name is divided into three parts by hyphens (-):

- The short identifier of the mail delivery agent; in this case, SMTP for SMTP delivery
- The UNIX time stamp with the date and time of the first delivery attempt encoded
- The incremental run number in the batch

This name is also used as a serial ID of the message, and can be found in the header of each delivered email message under the key: `X-EXS-SerialID`.

6.3.4 SMTP Configuration Parameters

The Email Delivery Connector provides RFC822-compatible SMTP delivery functionality, which lets you deliver messages using this protocol to various targets that support SMTP protocol, such as email servers and SMTP-enabled SMS gateways.

SMTP configuration parameters used with the Email Delivery Connector

Parameter	Description
`SMTPDelivery.auth`	The `SMTPDelivery.auth` parameter specifies whether to use SMTP authentication when communicating with the SMTP server. Select from one of the following values: • `True`—The `True` value uses SMTP authentication. • `False`—The `False` value does not use SMTP authentication. **Example:** `SMTPDelivery.auth=False` The default value for this parameter is `False`.
`SMTPDelivery.hostname`	The `SMTPDelivery.hostname` parameter specifies the host name of the SMTP server. **Example:** `SMTPDelivery.hostname=localhost` The default value for this parameter is `localhost`.
`SMTPDelivery.port`	The `SMTPDelivery.port` parameter specifies the SMTP listener port. **Example:** `SMTPDelivery.port=25` The default value for this parameter is `25`.
`SMTPDelivery.username`	The `SMTPDelivery.username` parameter specifies the user name, if required, used to establish authenticated connection to SMTP server. **Example:** `SMTPDelivery.username=empty string` The default value for this parameter is `empty string`.
`SMTPDelivery.password`	The `SMTPDelivery.password` parameter specifies the password, if required, used to establish authenticated connection to SMTP server. **Example:** `SMTPDelivery.password=empty string` The default value for this parameter is `empty string`.

SMTP configuration parameters used with the Email Delivery Connector, continued

Parameter	Description
`SMTPDelivery.error.path`	The `SMTPDelivery.error.path` parameter specifies the file system path for the SMTP error queue that contains the messages and attachments that failed during delivery. This path must already exist and must allow read/write access for the module. **Example:** `SMTPDelivery.error.path=C:\Program Files\ Documents\Email Delivery\Error queue` There is no default value for this parameter.
`SMTPDelivery.delivery.cleanup`	The `SMTPDelivery.delivery.cleanup` parameter manages cleanup after the successful delivery of a message. Select from one of the following values: • `True`—The `True` value deletes all attachment files related to message. • `False`—The `False` value keeps the attachment files related to the message. **Example:** `SMTPDelivery.delivery.cleanup=False` The default value for this parameter is `False`.
`SMTPDelivery.file.completion.interval`	The `SMTPDelivery.file.completion.interval` parameter specifies the polling interval used to verify the attachment file is complete. The value is the specified interval (in milliseconds) after which the system checks the file size for change. **Example:** `SMTPDelivery.file.completion.interval=500` The default value for this parameter is `500`. **Note:** Do not change this parameter from its default value.

6.4 Testing the Email Delivery Connector

For best results, before using the Email Delivery Connector in production, you should test your configuration with the Email Delivery Connector in standalone mode. Testing the configuration of the Email Delivery Connector helps to prevent errors that can occur during production runs.

To test the Email Delivery Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - Customer report file
   - Output queues for attachments, in the order the attachments are listed in the report data file (optional)
   - Initialization file for body message text (optional)
   - A customer driver file with sample customer data
2. Package your application and run the engine.
3. Run the Email Delivery Connector in standalone mode by calling the Java Virtual Machine (JVM) using the following command:

   ```
   java -cp ./activation.jar;./EmailConnector.jar;./log4j.jar;./mail.jar
      com.exstream.emaildelivery.application.EmailDeliveryExec
      --use-properties "C:\DDA E-mail\Setup\bin\emaildelivery.properties"
   ```
4. Verify that your output is correct based on the configuration of the Email Delivery Connector.

Chapter 7: The IBM Content Manager Connector

The IBM Content Manager (IBMCM) Connector extracts content (such as, text, TIFF files, or other supported placeholder variable types) from the IBM Content Manager client and then passes it to the engine to populate a placeholder variable so that design pages in an Exstream design, such as marketing campaigns, can be filled with targeted communications.

For example, suppose your company uses the IBM Content Manager client to store customer data and you want to improve the customer experience by offering customer-specific coupons with outgoing documents. An Exstream designer creates a document with both image and text placeholders. Exstream developers package the application, including information in the customer driver file for the placeholder variables in the document. The engine processes the package file to produce the application and, using information in the customer driver file, sends a request for data, such as a company logo (image) and a marketing message (text). Acting as the interface point between the engine and the IBM Content Manager client, the IBMCM Connector extracts the requested data from the IBM Content Manager client and sends that data to the engine for processing. The engine processes the image and text and, based on the variable definitions specified on the design page, populates the image placeholder with the company logo and the marketing message for the text placeholder.

This chapter discusses the following topics:

- “Preparing the IBMCM Connector” below
- “Configuring the IBMCM Connector” on the next page
- “Testing the IBMCM Connector” on page 126

7.1 Preparing the IBMCM Connector

To prepare the IBMCM Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the `CMConnector.zip` file.
2. Extract the contents of the `CMConnector.zip` file to the directory that contains the Exstream engine.
3. For best results, make sure any applications that include test files package and run in batch mode with no errors.

7.2 Configuring the IBMCM Connector

The IBMCM Connector is a simple placeholder DDA connector that can receive requests for content from the engine, extract that content from the IBM Content Manager client, and then pass that content to the engine to populate a placeholder variable.

This section discusses the following topics:

- “Configuring the IBMCM Connector in Design Manager” below
- “Setting Up Your Initialization File” on page 123
- “IBMCM Connector Parameters” on page 123

7.2.1 Configuring the IBMCM Connector in Design Manager

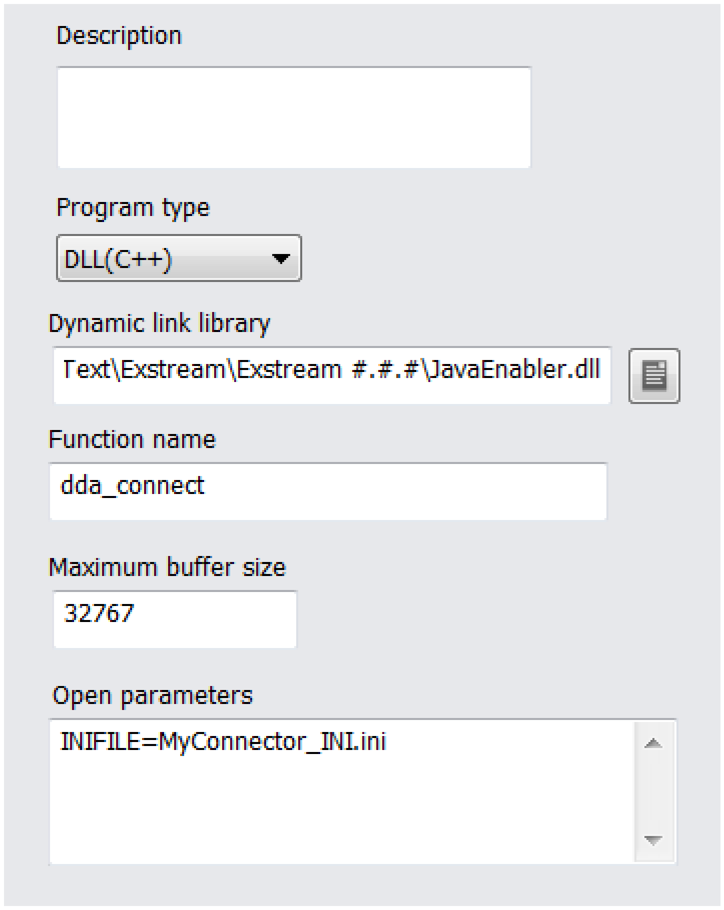
When you configure the IBMCM Connector in Design Manager, you must first create a connector object in the Design Manager Library. Then, you must assign the IBMCM Connector to all applicable data files and placeholder variables that relate to the applications that you will use with the connector.

To configure the IBMCM Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	In the **Program Type** list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the IBMCM Connector DLL, or click 🗎 to go to the file: For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on page 123.

Example of the completed connector object properties for the IBMCM Connector


Description

Program type
DLL(C++)

Dynamic link library
Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name
dda_connect

Maximum buffer size
32767

Open parameters
INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Dynamic Data Access" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning a connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to a placeholder variable.

   For more information about assigning a connector to a placeholder variable, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

7.2.2 Setting Up Your Initialization File

After you configure the IBMCM Connector in Design Manager and set additional configurations, you must complete the following:

- Assign the connector object to a placeholder variable. For the placeholder variable value, assign the `SEARCH_ATTRIBUTE` parameter value that you are searching for in the IBMCM.
- Set up the initialization file specified in the connector object.

  The initialization file is a text file that contains the remaining configuration settings and control parameters. It consists of operational details using both required and optional parameters. These parameters act as containers in which you specify certain definitions that control the overall behavior of your connector.

  You specify the name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager instead of listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

Note: Make sure the value of the placeholder variable in Design Manager is the same as the value of the `SEARCH_ATTRIBUTE` parameter in the connector initialization file.

For more information about referencing a connector initialization file in Design Manager, see "Configuring the IBMCM Connector in Design Manager" on page 120

To set up your initialization file for the IBMCM connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For more information about parameters for the IBMCM Connector, see "IBMCM Connector Parameters" below.

7.2.3 IBMCM Connector Parameters

You must define the parameters for the IBMCM Connector when you set up the initialization file.

To configure the parameters for the IBMCM Connector, you must include the following initialization parameters.

Initialization parameters for the IBMCM Connector

Parameter	Description

`CLASS`	The `CLASS` parameter specifies the name of the IBMCM Connector class that is used by the IBM Content Manager client.
	
	For all of the supported messaging queues, you must set the `CLASS` parameter to the following path:
	
	`com/exstream/connector/CMConnector`
	
	Example:
	
	`CLASS=com/exstream/connector/CMConnector`
	
	There is no default value for this parameter.

`CLASSPATH`	The `CLASSPATH` parameter specifies the system paths to the class libraries required by the IBMCM Connector. The system path values depend on the messaging queue.
	
	Example:
	
	`CLASSPATH=<path>\JavaEnabler.jar;<path>\CMConnector.jar;<path>\cmbcm81.jar;`
	`<path>\cmbicm81.jar;<path>\cmbsdk81.jar;<path>\Common.jar;`
	`<path>\db2jcc.jar;<path>\db2jcc_license_cu.jar;<path>\db2java.zip;`
	`<path>\cmbcmenv.properties`
	
	Separate each path with the correct CLASSPATH separator for the platform (a colon or semicolon).
	
	There is no default value for this parameter.

`TRACE`	The `TRACE` parameter specifies whether to record connector activity in a file. Select from one of the following values: • `NO`—The `NO` value indicates that connector activity is not recorded. • `YES`—The `YES` value indicates that connector activity is recorded to the standard DDA output location. • `LOG`—The `LOG` value indicates the name of the file where traced connector activity is recorded. If a file name is not specified for the `LOG` value, the default `CMConnector.log` file is used. **Example:** `TRACE=NO` The default value for this parameter is `NO`.
`LIB_SERVER_NAME`	The `LIB_SERVER_NAME` parameter specifies the name of the library server that you use to store your data. The value of this parameter is the datastore name that the IBM Content Manager client recognizes. **Example:** `LIB_SERVER_NAME=ICMNLSDB` The default value for this parameter is `ICMNLSDB`.

Initialization parameters for the IBMCM Connector, continued

Parameter	Description
`USERID`	The `USERID` parameter specifies the user ID that you use to connect to the IBM Content Manager client. The value for this parameter is the user name that the IBM Content Manager client recognizes. **Example:** `USERID=ICMADMIN` The default value for this parameter is `ICMADMIN`.
`USER_ PASSWORD`	The `USER_PASSWORD` parameter specifies the password that you use to connect to the IBM Content Manager client. The value for this parameter is the user password that the the IBM Content Manager client recognizes. **Example:** `USER_PASSWORD=admin` There is no default value for this parameter.
`ITEMTYPE`	The `ITEMTYPE` parameter specifies the item type that the IBM Content Manager client uses to identify specific content requested by the engine. The item type must match an item type stored in the IBM Content Manager client. The item type is used by the IBM Content Manager client to identify the attributes of specific content and to refer to the content itself. **Example:** `ITEMTYPE=TEXT` There is no default value for this parameter.
`OBJECT_TYPE`	The `OBJECT_TYPE` parameter specifies the type of object stored in the IBM Content Manager client as an item type. The value of this parameter must match the type of object stored in the selected item type. Select from one of the following values: • `TEXT`—The `TEXT` value indicates a text file object type. • `TIFF`—The `TIFF` value indicates a TIFF file object type. **Example:** `OBJECT_TYPE=TEXT` The default value for this parameter is `TEXT`.

Initialization parameters for the IBMCM Connector, continued

Parameter	Description
SORT_ ATTRIBUTE	The SORT_ATTRIBUTE parameter specifies the name of an attribute. The attribute must be an attribute of the selected item type. The SORT_ATTRIBUTE parameter is used to select content based on the sort attribute. For example, if the sort attribute is a date, specifying the DESCENDING value would result in the extraction of the most recent content stored for the selected item type. In addition to specifying the name of the item attribute you are searching for, specify one of the following parameters: • DESCENDING—The DESCENDING value indicates that the IBMCM Connector extracts all content stored for the selected item type beginning with the most recently stored content. • ASCENDING—The ASCENDING value indicates that the IBMCM Connector extracts all content stored for the selected item type beginning with the content stored preceding the most recent content. **Example:** `SORT_ATTRIBUTE=<item_attribute_name>,DESCENDING` The default value for this parameter is DESCENDING.
EFFECTIVE_ DATE_ ATTRIBUTE	The optional EFFECTIVE_DATE_ATTRIBUTE parameter specifies the name of the effective date attribute. The attribute must be associated in the IBM Content Manager client with the selected item type. This attribute type is used to select content prior to and including the effective date. If no effective date is specified, the IBMCM Connector will select the item with the most recent date. **Example:** `EFFECTIVE_DATE_ATTRIBUTE=XX/XX/XXXX` There is no default value for this parameter.
SEARCH_ ATTRIBUTE	The SEARCH_ATTRIBUTE parameter specifies the name of the attribute associated with the specific content that you want to extract. For example, if the attribute is called filename (referencing the name of a file), the IBMCM Connector extracts content that has that file name. **Example:** `SEARCH_ATTRIBUTE=TEXT` There is no default value for this parameter.

7.3 Testing the IBMCM Connector

For best results, before using the IBMCM Connector in production, you should test your configuration. Testing the configuration of the IBMCM Connector helps to prevent errors that can occur during production runs.

To test the IBMCM Connector, you must complete the following steps:

1. Verify that you have added the placeholder variable to your application.
2. Verify that you have added the initialization file to the connector object.
3. Verify that the connector object is assigned to all relevant placeholder variables in the application.
4. Package your application and run the engine with the ONDEMAND switch in your control file.
5. Verify that your output is correct.

If the placeholder content does not appear as expected, perform a search in the IBMCM client for the test search value. Make sure that the test search value is correctly assigned to the placeholder variable during the engine run. Also ensure that the `SEARCH_ATTRIBUTE` value is spelled correctly in the connector initialization file and that the value matches what is in the IBMCM client.

Chapter 8: The WebSphere MQ Connector

The IBM WebSphere MQ (WSMQ) Connector lets the Exstream engine interact with IBM WebSphere MQ enterprise messaging software using user-written DDA routines. When you use the WSMQ Connector to receive and process requests, the WSMQ Connector dynamically allocates the memory necessary to process incoming requests and then passes those requests to the engine. Files that contain a large amount of data, such as financial statements, can cause engine processing errors when memory allocation is not handled dynamically. The dynamic memory allocation that the WSMQ Connector uses lets you avoid potential engine processing errors when large data files are passed through the connector. The WSMQ Connector has the following capabilities:

- Correlate input and output messages based on IDs (Correlation mode)
- Send messages to remote queues
- Access multiple instances of the Exstream engine simultaneously
- Pass messages synchronously and asynchronously

For example, suppose your company uses IBM WebSphere MQ software to receive customer requests for on-demand financial portfolios. Financial portfolios can be transaction-intensive, which means they can produce a file that requires more memory than is normally allowed with an on-demand solution. Since the WSMQ Connector dynamically allocates memory, you are not forced to monitor a buffer size setting. Instead, you can configure the WSMQ Connector so that the engine can process all customer requests without having to shut down. Using the WSMQ Connector, you can connect the messaging service with the Exstream engine to produce customer-requested documents on demand. Additionally, you can configure the connector to send customer-requested documents through multiple correlated outputs. Configuring the WSMQ Connector to use multiple correlated outputs lets you accommodate customer requests by providing a specific type of output format (for example, PDF) to customers.

This chapter discusses the following topics:

- “Preparing the WSMQ Connector” on the next page
- “Configuring the WSMQ Connector” on page 132
- “Testing the WSMQ Connector” on page 153

8.1 Preparing the WSMQ Connector

The WSMQ Connector is available on multiple platforms. Depending on whether you are installing the WSMQ Connector on a mainframe or non-mainframe platform, you must download the appropriate ZIP file from My Support.

The naming convention for the ZIP file is `[NAME]_Connector_[VERSION]_[PLATFORM].zip`.

For example, `MQ_Connector_1_5_008_AIX_64.zip` is WSMQ Connector version 1.5.008 for the 64-bit AIX platform.

To prepare the WSMQ Connector for configuration, testing, and use in production, you must complete one of the following tasks:

1. “Installing the WSMQ Connector on a Non-Mainframe Platform” below
2. “Installing the WSMQ Connector on a Mainframe Platform” on the next page

8.1.1 Installing the WSMQ Connector on a Non-Mainframe Platform

To install the WSMQ Connector on a non-mainframe platform, you must complete the following steps:

1. From the ZIP file, extract the connector file for your platform:

To	Do this
Extract the connector file from the installation ZIP file to your Windows platform	Select the `ExMQConnect.dll` file. The `ExMQConnect.dll` file contains a test database with two sample applications as well as the required files to run test cases for the connector.
Extract the connector file from the installation ZIP file to your UNIX or Linux platforms	From the `MQSeries.tar.Z` file, select the `libExMQConnector.so` file. The `libExMQConnector.so` file contains two sample applications as well as the required files to run test cases for the connector.

2. Set the following environment variables as required for your operating system platform:

   ```
   MQSERVER=CHANNELNAME<TRANSPORTTYPE><CONNECTIONNAME>
   ```

   For example:

   ```
   MQSERVER=SYSTEM.DEF.SVRCONN/TCP/localhost(1415)
   ```

   For HP-UX, you must indicate the connector directory in the `LD_PRELOAD`environment variable.

Caution: Values specified in the initialization file for the `CHANNELNAME`, `TRANSPORTTYPE`, and `CONNECTIONNAME` parameters override the values specified in the `MQSERVER` environment variable.

8.1.2 Installing the WSMQ Connector on a Mainframe Platform

In addition to the base installation task for the WSMQ Connector on a mainframe platform, you must complete the following post-installation tasks:

1. "Editing Terse Files for the WSMQ Connector" below
2. "Editing the Mainframe JCL for the WSMQ Connector" on the next page
3. "Setting the POSIX Option to Prevent Engine Errors" on page 132

Editing Terse Files for the WSMQ Connector

To edit Terse files, you must complete the following steps:

1. Open the `sendtrs.txt` file in a text editor.
2. In the `sendtrs.txt` file, edit the following values:

Original value	New value
`IBM6000`	Use the URL or IP address of your mainframe.
`P390A`	Use your FTP user name.
`P390A6`	Use your FTP password.
`SCPMV3`	Use the volume where you want the Terse files to be placed. You must change the directory to the location where the message queue will reside.

3. Double-click the `sendtrs.bat` file.
4. In the mainframe, open the `VERUNP` in a text editor.
5. Add or edit your job card as appropriate.
6. In the z/OS definition of `UNPACK`, change the `USERID` value to the high-level qualifier where you loaded your Terse files. For example:

```
C P390A <new high level qualifier> ALL
```

7. Change the `VOLSER` symbolic parameter default value from `SCPMV3` to the appropriate parameter value.
8. Change the `JOBLIB` DSN to the library containing the Terse utility.
9. Change `EXSTREAM` value in the `INFILE` Data Definition (DD) statement and the `OUTFILE` DD statement to your specified data set.
10. Save the changes to the `VERUNP` file.
11. Save the `VERUNP` file.
12. Verify that your changes work in your installation.

Editing the Mainframe JCL for the WSMQ Connector

After you have edited the Terse files and verified that the edited files work in your installation, you must edit the mainframe JCL to add the values for your partitioned data set attributes. The high-level qualifiers indicate the locations for the main components of your integrated systems, such as the locations of your Terse files, the Exstream engine, and the name of the MQ Queue Manager.

To edit the mainframe JCL, `JCL(VEREXMQ)`, you must complete the following steps:

1. Open the `JCL(VEREXMQ)` file in a text editor.
2. Edit the `JCL(VEREXMQ)` file and set the following values:

   Partitioned data set values

Attribute	Description
`HLQ=P390A`	The `HLQ=P390A` data set value indicates the high-level qualifier where you loaded your Terse files.
`EHLQ=P390A`	The `EHLQ=P390A` data set value indicates the high-level qualifier where the engine is installed.
`MQHLQ=CSQ520`	The `MQHLQ=CSQ520` data set value indicates the high-level qualifier where MQ Series is installed.
`QMGR='CSQ1'`	The `QMGR='CSQ1'` data set value indicates the name of your MQ Queue Manager.
`STEPLIB`	The `STEPLIB` data set value indicates the location of your IBM MQ Series installation. **Note:** On the `VERIFY` parameter of the `STEPLIB` data set value, retain the `CSQ520.SCSQANLE` default value.

3. On the `VERDATA (EXMQSINI)` line, change the `QUEUEMGR=CSQ1` value to the name of your MQ Queue Manager.
4. Comment out the line that contains `MSGSYM`.
5. On the lines `STEPLIB` and `DLMSGRES`, change `EXSTREAM` to the location under the high-level qualifier where the engine has been installed.

6. Save the `JCL(VEREXMQ)` file and verify installation.
7. Verify that your changes work in your installation.

Setting the POSIX Option to Prevent Engine Errors

When running the WSMQ Connector on a mainframe platform, you must set the z/OS POSIX run-time option in your JCL file so that you can use a `sleep ()` function call. Using this parameter option prevents engine errors, such as an abend, when the MQ Queue Manager shuts down.

To enable the POSIX option in the parameter field of your JCL file, you must use the following command:

```
PARM=' POSIX(ON) / -USECONTROL=YES'
```

8.2 Configuring the WSMQ Connector

The WSMQ Connector lets the engine communicate with the IBM WebSphere MQ system to generate on-demand documents. The engine generates documents when a message is sent to the message queue and a request is sent to the engine for fulfillment. During fulfillment, the resulting output is then sent back to the message queue to fulfill the request. The WSMQ Connector is responsible for managing the flow of data to the engine, and the delivery of the resulting output back to the message queue. You must configure the WSMQ Connector so that it can be identified by both the message queue and the engine and identify the data to be processed. Additionally, you must make sure that the configuration includes a set of parameters that defines how the WSMQ Connector handles specific processes.

In addition to the basic configuration requirements for the WSMQ Connector, you can also use the WSMQ Connector with a specific set of engine commands to control the workflow process, use multiple engines simultaneously (correlation mode), or use a reference file DDA routine to search for specific messages in the message queue. To use the WSMQ Connector with these additional configurations, you must define additional parameters in your initialization file.

This section discusses the following topics:

1. “Configuring the WSMQ Connector in Design Manager” on the next page
2. “Setting Up Your Initialization File” on page 134
3. “WSMQ Connector Parameters” on page 136
4. “Setting Engine Commands for WSMQ Connector Output Queues” on page 149
5. “Enabling Correlation Mode for the WSMQ Connector and Multiple Engines” on page 150
6. “Supporting Reference File Lookups in the Messaging Service” on page 152

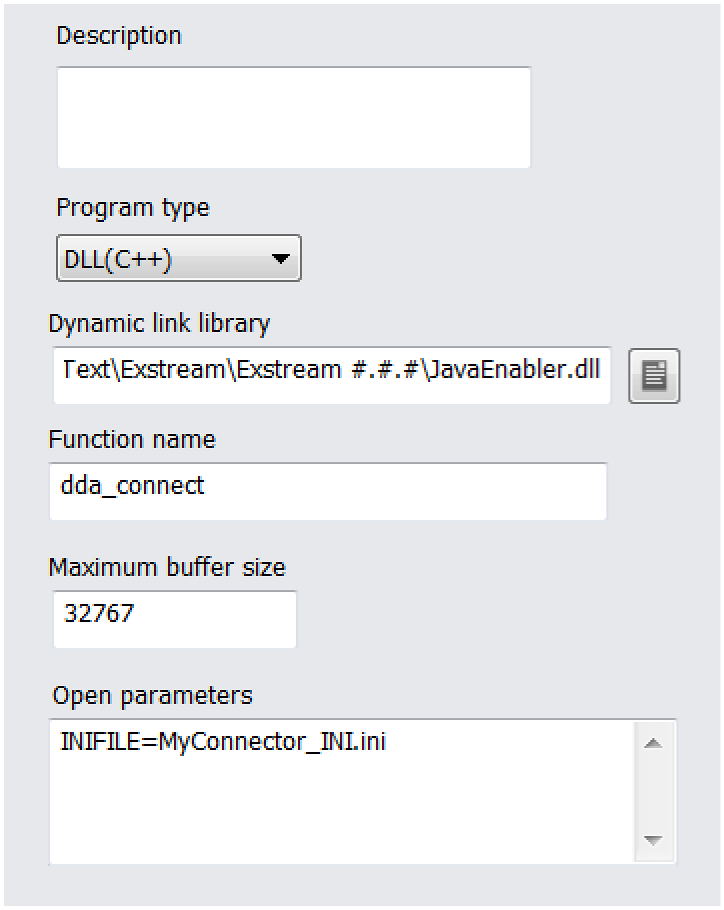
8.2.1 Configuring the WSMQ Connector in Design Manager

To configure the WSMQ Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to WSMQ DLL, or click 📄 to go to the file: • For Windows—Enter the path to `ExMQConnect.dll`. • For UNIX/Linux—Enter the path to `libExMQConnect.so`. • For HP-UX—Enter the path to `libExMQConnect.sl`. • For z/OS—Enter the path to `EXMQCONN`.
Specify the type of function name	In the **Function name** box, enter `processRec`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, then the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on the next page

Example of the completed connector properties for the WSMQ Connector


Description

Program type
DLL(C++)

Dynamic link library
Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name
dda_connect

Maximum buffer size
32767

Open parameters
INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Dynamic Data Access" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning the WSMQ Connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to an output queue.

   For more information about assigning the WSMQ Connector to an output, see "Assigning a Connector Object to an Output Queue" on page 28.

8.2.2 Setting Up Your Initialization File

After you configure the WSMQ Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager as opposed to listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

To set up your initialization file for the WSMQ connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For more information about parameters for the WSMQ Connector, see “WSMQ Connector Parameters” on the next page.

Specifying the Type of Initialization File to Use

Some of the parameters available for a connector support the use of double-byte characters. The use of double-byte characters in a connector parameter gives the user more flexibility in defining connector behavior. If the connector you configure allows for parameters that support the use double-byte characters, you must first specify the type of initialization file you will be using to define the parameters. For example, if your initialization file for the WSMQ Connector lists the `MESSAGEFILE` parameter with a message file name value that uses double-byte characters, you must first specify a Unicode initialization file by defining the `UNICODEINIFILE` parameter.

To specify the type of initialization file to use, you must specify one of the following parameters.

Parameters for specifying an initialization file

Parameter	Description
`INIFILE`	The `INIFILE` parameter specifies an initialization file containing parameters that support only the use of single-byte characters. When specifying connector behavior with an initialization file, the `INIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. **Example:** `INIFILE=MyINIFile.ini` There is no default value for this parameter.

Parameters for specifying an initialization file, continued

Parameter	Description
`UNICODEINIFILE`	The `UNICODEINIFILE` parameter specifies an initialization file that contains one or more supported parameters with Unicode characters. When specifying connector behavior with an initialization file that contains Unicode characters in parameters, the `UNICODEINIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. When the engine does not detect a byte-order marker, you must use the `UNICODEINIFILE` parameter so that the engine can interpret Unicode characters correctly. **Example:** `UNICODEINIFILE=MyUnicodeINIFile.ini` There is no default value for this parameter.

8.2.3 WSMQ Connector Parameters

After you have specified the type of initialization file that will contain the WSMQ Connector parameters, you must also include in the initialization file (at a minimum) the parameters that identify the queue manager, input queue, and output queue. For more advanced configurations, you can add optional parameters that control how the input queue receives print requests, which control messages that the engine uses to regulate the message delivery process, and/or how reference files are used to search for messages.

This section discusses the following topics:

- “Required Initialization Parameters for the WSMQ Connector” below
- “Optional Parameters for the WSMQ Connector” on the next page
- “Secure Channel Parameters for the WSMQ Connector” on page 147

Required Initialization Parameters for the WSMQ Connector

To configure the initialization parameters for the WSMQ Connector, you must include at least one of the following required parameters.

Required initialization parameters for the WSMQ Connector

Parameter	Description
`INPUTQUEUE`	The `INPUTQUEUE` parameter specifies the queue name on which the WSMQ Connector receives messages from the WebSphere MQ server. **Example:** `INPUTQUEUE=MyInputQueue` There is no default value for this parameter.

Required initialization parameters for the WSMQ Connector, continued

Parameter	Description
`OUTPUTQUEUE`	The `OUTPUTQUEUE` parameter specifies the queue name on which the WSMQ Connector sends return messages to the WebSphere MQ server. **Example:** `OUTPUTQUEUE=MyOutputQueue` There is no default value for this parameter.
`QUEUEMGR`	The `QUEUEMGR` parameter specifies the name of the queue manager that the WSMQ Connector uses. **Example:** `QUEUEMGR=MyQueueMgr` There is no default value for this parameter.

Optional Parameters for the WSMQ Connector

After you have specified the required parameters for the WSMQ Connector, you can define optional parameters that are available for the WSMQ Connector. The optional parameters let you define specialized functions of the connector, such as correlation mode or the size of the buffer for the message queue.

Optional initialization parameters for the WSMQ Connector

Parameter	Description
`CONTROLONLY`	The `CONTROLONLY` parameter specifies that the WSMQ Connector deliver only control messages to the engine when in transaction mode. **Example:** `CONTROLONLY` There is no default value for this parameter.
`CONVERT`	The `CONVERT` parameter specifies whether to convert messages into the format required by the receiving system before transmission. Select from one of the following values: • Y—The `Y` value specifies that the WSMQ Connector converts messages. • N—The `N` value specifies that the WSMQ Connector does not convert messages. **Example:** `CONVERT=Y` The default value for this parameter is `Y`.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`CONVERTCCSID`	If you are using the `CONVERT` parameter to convert messages that contain accented characters into the format required by the receiving system before transmission, the `CONVERTCCSID` parameter specifies the coded character set identification number that the messaging queue manager requires to convert the messages and read accented characters correctly. **Note:** If you are using the `CONVERTCCSID` switch, you must also specify `Y` for the `CONVERT` parameter in your WSMQ Connector initialization file. If you set the `CONVERT` parameter to `N`, you receive a warning message and the `CONVERTCCSID` parameter is ignored. The value for this parameter is the number that represents the coded character set identification number. For example, if you want to convert UTF-8 characters, you must specify `1208` as the argument. For a complete listing of argument values, see the IBM MQ documentation on the IBM website. **Example:** `-CONVERTCCSID=1208` The default value for this parameter is `MQCCSI_DEFAULT`.
`CORRELATE`	The `CORRELATE` parameter specifies whether the WSMQ Connector correlates input and output messages based on IDs. The `CORRELATE` parameter, used with correlation mode, can use single or multiple output files. Select from one of the following values: • `MSG`—The `MSG` value passes the input `MSGID` value to the `CORRELID` field of the output message. • `CID`—The `CID` value passes the input `CORRELID` value to the `CORRELID` field of the output message. **Example:** `CORRELATION=CID` There is no default value for this parameter. For more information about using correlation mode with the WSMQ Connector, see "Enabling Correlation Mode for the WSMQ Connector and Multiple Engines" on page 150.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`CORRELATIONREPLY`	The `CORRELATIONREPLY` parameter sends the return message first to `REPLYTOQUEUE` and `REPLYTOQMGR` correlation parameters specified in the input message descriptor. The `CORRELATIONREPLY` parameter is valid only when creating one output file per input record. The `CORRELATIONREPLY` parameter, used with correlation mode, determines the result if the first connect attempt fails. Select from one of the following values: • `FROMINPUTMSGONLY`—The `FROMINPUTMSGONLY` value specifies that the application exits and logs an error. • `FROMINPUTMSGFIRST`—The `FROMINPUTMSGFIRST` value specifies that the application uses the global queue manager and the specified output queue. • `QUEUE_FROM_INPUT_MSG_ONLY`—The `QUEUE_FROM_INPUT_MSG_ONLY` value specifies the same action as the `FROMINPUTMSGONLY` value, but the application replies to the default queue manager instead of replying to the `ReplyToQMgr` correlation parameter. • `QUEUE_FROM_INPUT_MSG_FIRST`—The `QUEUE_FROM_INPUT_MSG_FIRST` value specifies the same action as the `FROMINPUTMSGFIRST` value, but the application replies to the default queue manager instead of replying to the `ReplyToQMgr` correlation parameter. **Example:** `CORRELATIONREPLY=FROMINPUTMSGONLY` The default value for this parameter is `FROMINPUTMSGFIRST`. For more information about using correlation mode with the WSMQ Connector, see "Enabling Correlation Mode for the WSMQ Connector and Multiple Engines" on page 150.
`CORRELATIONREPLYMETHOD`	The `CORRELATIONREPLYMETHOD` parameter specifies the location to send the reply message by using the incoming queue message. Select from one of the following values: • `DIRECTLYCONNECT`—The `DIRECTLYCONNECT` value specifies that the WSMQ Connector send the reply message to the queue manager and queue specified in the initialization file. • `MQODONLY`—The `MQODONLY` value uses the queue manager specified in the initialization file as a relay manager by extracting the values for the `ReplyToQMgr` and `ReplyToQueue` correlation parameters and placing those values on the Message Queue Object Descriptor (MQOD) of the outgoing message. Placing the values of the `ReplyToQMgr` and `ReplyToQueue` correlation parameters on the MQOD specifies the location that the queue manager and queue uses to forward the message. **Example:** `CORRELATIONREPLYMETHOD=DIRECTLYCONNECT` The default value for this parameter is `DIRECTLYCONNECT`. For more information about the `ReplyToQueue` and `ReplyToQMgr` correlation parameters, see "Enabling Correlation Mode for the WSMQ Connector and Multiple Engines" on page 150.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`DATAONLY`	The `DATAONLY` parameter specifies that the WSMQ Connector send only driver and reference file messages when in transaction mode. **Example:** `DATAONLY` There is no default value for this parameter.
`DDAPOSTPROCESSOR=ProgramName, FunctionName, ProgramType [,MaxBufferSize, OpenParameters, AccessType]`	The `DDAPOSTPROCESSOR` parameter specifies the processor program to call after receiving an input message from the WSMQ Connector input queue before sending it to the engine. Select one of the following access type values: • `0`—Read. This is the default access type. • `9`—Transform. Use this access type when you want to call `DDAPOSTPROCESSOR` as an input record transform. For more information about access types, see "Parameters for DDA Routine Functions" on page 35. **Example:** `DDAPOSTPROCESSOR=C:\Program Files\Exstream\JavaEnabler.dll, dda_connect, dll, 32767, INIFILE=javaenabler.ini, 0` If you do not require open parameters as a part of your DDAPOSTPROCESSOR routine, but you require an access type, you can enter a dummy string of open parameters. WSMQ Connector passes the dummy parameters on the Open call and the DDAPOSTPROCESSOR routine ignores these parameters.
`DDAPREPROCESSOR=ProgramName, FunctionName,ProgramType [,MaxBufferSize, OpenParameters, AccessType]`	The `DDAPREPROCESSOR` parameter specifies the processor program to call after receiving an output message from the engine. Processing occurs before sending an MQ report message to the queue named in the `OUTPUTQUEUE` parameter. Select one of the following access type values: • `1`—Write. This is the default access type. • `9`—Transform. Use this access type when you want to call `DDAPREPROCESSOR` as an output record transform. For more information about access types, see "Parameters for DDA Routine Functions" on page 35. **Example:** `DDAPREPROCESSOR=C:\Program Files\Exstream\JavaEnabler.dll, dda_connect, dll, 32767, INIFILE=javaenabler.ini, 1` If you do not require open parameters as a part of your DDAPREPROCESSOR routine, but you require an access type, you can enter a dummy string of open parameters. WSMQ Connector passes the dummy parameters on the Open call and the DDAPOSTPROCESSOR routine ignores these parameters.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`EOF_CYCLE_AFTER_MSG`	The `EOF_CYCLE_AFTER_MSG` parameter requires that the WSMQ Connector return an `EOF` call after every message. Use the `EOF_CYCLE_AFTER_MSG` parameter when your input messages and output messages are one-to-one. **Example:** `EOF_CYCLE_AFTER_MSG` There is no default value for this parameter.
`EXITON`	The `EXITON` parameter specifies how the WSMQ Connector handles reason codes returned by calls, such as `MQGET`. Select from one of the following values: • `#[,#,#,...]`—The `#[,#,#,...]` value lets you specify the WSMQ Connector reason codes returned from `MQGET` calls that cause the engine to exit and shut down. Because the WSMQ Connector dynamically allocates memory for incoming data, the WSMQ Connector bypasses a `2080` return code by increasing the input buffer size. Therefore, `EXITON=2080` does not force the engine to exit. • `WRITEERROR`—The `WRITEERROR` value lets you specify the reason code for an `MQPUT` call. If the `MQPUT` call returns the failure reason code, then the WSMQ Connector shuts down the engine. **Example:** `EXITON=WRITEERROR` The default value for this parameter is `WRITEERROR`. For more information about setting engine commands for the WSMQ Connector, see "Setting Engine Commands for WSMQ Connector Output Queues" on page 149.
`EXPIRYOUTPUT`	The `EXPIRYOUTPUT` parameter specifies the number of seconds an output message remains active in the queue before the queue manger deletes the message. Select from one of the following values: • `[number of seconds]`—The `[number of seconds]` value specifies the number of seconds that the WSMQ Connector waits before deleting a message from the queue manager. • `MQEI_UNLIMITED`—The `MQEI_UNLIMITED` value specifies that the WSMQ Connector does not delete messages from the queue manager. **Example:** `EXPIRYOUTPUT=MQEI_UNLIMITED` The default value for this parameter is `MQEI_UNLIMITED`.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`EXPIRYSHUTDOWN`	The `EXPIRYSHUTDOWN` parameter specifies the number of seconds the `DDA SHUTDOWN` message stays in queue. When WSMQ Connector receives a shutdown message (an MQ message with a priority of `9` and a body of `SHUTDOWN`), the DDA posts a copy of that shutdown message back onto the queue so other engines reading from the same queue also receive the shutdown request. This parameter indicates the expiry property of the copied shutdown message. If you are running only one engine, set the value to `0`. This parameter reduces wait time for the copy that was placed in the queue to expire before you can restart the engine. **Example:** `EXPIRYSHUTDOWN=60` The default value for this parameter is `60`.
`FILETYPE`	The `FILETYPE` parameter specifies the input file type. Select from one of the following values: • `LOOKUP`—The `LOOKUP` parameter specifies DDA reference files. • `DRIVER`—The `DRIVER` value specifies driver files. **Example:** `FILETYPE=DRIVER` The default value for this parameter is `DRIVER`. For more information about using the `FILETYPE=LOOKUP` parameter with reference file lookups, see "Supporting Reference File Lookups in the Messaging Service" on page 152.
`INPUTBUFFERSIZE`	The `INPUTBUFFERSIZE` parameter specifies the size of the input buffer for WSMQ Connector messages. You must specify the buffer sizes in bytes. **Example:** `INPUTBUFFERSIZE=512K` The default value for this parameter is `512K`.
`LOOKUP`	The `LOOKUP` parameter specifies the lookup options for retrieving data. Select from one of the following values: • `CONFIRM`—The `CONFIRM` value specifies that the WSMQ Connector waits for data while the queue confirms the sent message. • `CORREL`—The `CORREL` value specifies that the WSMQ Connector waits for data to be returned from the specified `REPLYTOQUEUE` correlation parameter, or waits for data from the same input queue. **Example:** `LOOKUP=CORREL` The default value for this parameter is `CONFIRM`. For more information about using the `LOOKUP` parameter with reference file lookups, see "Supporting Reference File Lookups in the Messaging Service" on page 152.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`LOOKUPWAITINTERVAL`	The `LOOKUPWAITINTERVAL` parameter specifies the number of seconds that the WSMQ Connector waits for a lookup message to appear in the `REPLYTOQUEUE` field. **Example:** `LOOKUPINTERVAL=10` The default value for this parameter is `10`. For more information about using the `LOOKUPWAITINTERVAL` parameter with reference file lookups, see "Supporting Reference File Lookups in the Messaging Service" on page 152.
`MAXINACTIVITYCOUNT`	The `MAXINACTIVITYCOUNT` parameter specifies the number of get message sequential failures allowed before the transaction engine shuts down. **Example:** `MAXACTIVITYCOUNT=0` The default value for this parameter is `0`.
`MESSAGEFILE`	The `MESSAGEFILE` parameter specifies the name of the message file for DDA messages. Output is limited to every hundredth message after 100 consecutive queue messages. When the message queue message file exceeds 100,000,000 bytes, the message queue restarts. **Example:** `MESSAGEFILE=exmqsmsg.txt` The default value for this parameter is `exmqsmsg.txt`. The `MESSAGEFILE` parameter supports the use of DBCS characters.
`MSGFORMAT`	The `MSGFORMAT` parameter applies an IBM-reserved or user-defined format name to the message descriptor format field. The `MSGFORMAT` parameter notifies the receiver of the type of data in the message. The `MSGFORMAT` parameter affects all messages that the WSMQ Connector sends. You can use up to eight alphanumeric characters known to the queue manager's character set. Enter `NONE` as the value that sets the data type as undefined. Refer to the WebSphere MQ Application Programming Reference Manual from IBM for a list of the queue manager's built-in format names. **Example:** `MSGFORMAT=MQSTR` The default value for this parameter is `MQSTR`.
`PERSISTENCE`	The `PERSISTENCE` parameter lets you specify whether a message persists when the WSMQ Connector receives a put command. Select from one of the following values: • Y—The `Y` value specifies that messages persist. • N—The `N` value specifies that messages do not persist. **Example:** `PERSISTANCE=Y` The default value for this parameter is `Y`.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`PROCESS_CONTROL_MESSAGES`	The `PROCESS_CONTROL_MESSAGES` parameter lets the WSMQ Connector handle control messages that determine when the message queue and engine perform an action, such as `SHUTDOWN` or `SLEEP`. For example, when you place the `PROCESS_CONTROL_MESSAGES` parameter in your initialization file and the `STOP` message is sent to the message queue, the WSMQ Connector stops operation and sends a `SHUTDOWN` message to the engine so the engine stops processing. Additionally, when running multiple instances of the engine, you must place the `PROCESS_CONTROL_MESSAGES` parameter in your initialization file so that all instances of the engine receive and process the action specified in the control message. **Example:** `PROCESS_CONTROL_MESSAGES` There is no default value for this parameter. For more information about setting engine commands for the WSMQ Connector, see "Setting Engine Commands for WSMQ Connector Output Queues" on page 149.
`RECONNECT`	The `RECONNECT` parameter defines how the system responds to broken connections. Select from one of the following values: • `n`—The `n` value specifies the number of `MQRC_CONNECTION_BROKEN` errors the system tolerates in a session before taking action. The default is `n=0`. • `m`—The `m` value specifies the number of times the system attempts to reconnect. The system stops after the specified number. The default is `m=0`. • `i`—The `i` value specifies the number of seconds between each reconnection attempt. The default is `i=10`. If you use the `RECONNECT` parameter, you must define each value. **Example:** `RECONNECT=n=0,m=0,i=10`
`RECORD`	The `RECORD` parameter specifies that the WSMQ Connector buffers input messages per the record format. The `RECORD` parameter is required when sending multiple records per message by way of the input queue. Select from one of the following values: • `FIXED, #`—The `FIXED, #` value specifies buffer fixed records of specified length in # bytes. • `TERMINATED, HEX`—The `TERMINATED, HEX` value specifies the buffer terminated records according to the specified hex termination string. For example, `ODOA` is the hex code for `CR|LF` for native ASCII platforms. **Example:** `RECORD=TERMINATED, HEX` There is no default value for this parameter.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`REPLYTOQUEUE`	The `REPLYTOQUEUE` correlation parameter specifies the MQ Series `REPLYTOQUEUE` correlation parameter for lookup-type messages. **Example:** `REPLYTOQUEUE` There is no default value for this parameter. **Note:** Valid only when the `LOOKUP` parameter is set to `CORREL`. For more information about using the `REPLYTOQUEUE` correlation parameter with reference file lookups, see "Supporting Reference File Lookups in the Messaging Service" on page 152.
`SINGLEPACKAGE`	The `SINGLEPACKAGE` parameter lets you create one output file per input message. Without this option, an `EOF` call is sent only if there are no messages in the queue. **Example:** `SINGLEPACKAGE` There is no default value for this parameter.
`SYNCPOINT`	The `SYNCPOINT` parameter specifies whether to get messages with or without syncpoint control. Select from one of the following values: • Y—The Y value specifies to get messages with syncpoint control. • N—The N value specifies to get messages without syncpoint control. **Example:** `SYNCPOINT=Y` The default value for this parameter is `Y`.
`THROTTLEPROC`	The `THROTTLEPROC` parameter determines whether the engine entered sleep mode. The return value is `0` to continue or a number of seconds to sleep. The `THROTTLEPROC` parameter is not necessary unless you are using an external program to pace the engine. **Example:** `THROTTLEPROC=0` There is no default value for this parameter.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`TIMESTAMP_LOG_MESSAGES`	The `TIMESTAMP_LOG_MESSAGES` parameter specifies a time stamp for each action that the WSMQ Connector performs with the IBM WebSphere MQ server. For example, each time the WSMQ Connector receives a `read` call, a time stamp appears in the log message file identifying the date and time of the call. When the initialization file contains the `TIMESTAMP_LOG_MESSAGES` parameter, a time stamp appears for the following actions: • Open • Close • Read • Write • Seek **Example:** `TIMESTAMP_LOG_MESSAGES` There is no default value for this parameter.
`TRACE`	The `TRACE` parameter specifies that the output trace messages are written to the debug file. The `L` value creates a log file of the trace (`MQConn_Log.dat`). Select from one of the following values: • Y—The `Y` value specifies that the WSMQ Connector writes output trace messages to the debug log file. • N—The `N` value specifies that WSMQ Connector does not write messages to the debug log file. • L—The `L` value creates a log file named `MQConn_Log.dat` and specifies that the WSMQ Connector writes messages to the `MQConn_Log.dat` log file. **Example:** `TRACE=Y` The default value for this parameter is `N`.
`TRACEFILE`	The `TRACEFILE` parameter specifies the trace log file name and is valid only when the `TRACE` parameter is set to the `L` value. **Example:** `TRACEFILE=MQConn_Trace.dat` The default value for this parameter is `MQConn_Trace.dat`. The `TRACEFILE` parameter supports the use of DBCS characters in the file name.

Optional initialization parameters for the WSMQ Connector, continued

Parameter	Description
`TRANSMANAGE`	The `TRANSMANAGE` parameter specifies that the engine perform transaction management. Select from one of the following values: • `Y`—The `Y` value specifies that the engine checks the customer side for any connector calling errors. If there are none, then the WSMQ Connector commits the unit. If there is an error, then the WSMQ Connector backs out of the unit and the original set of customer information remains at the data source. • `N`—The `N` value specifies that the engine does not perform the check. • `SYNCPOINT`—The `SYNCPOINT` value specifies that the engine performs the check based on the commit time specified in the `TRANSCOMMITTIME` parameter. **Example:** `TRANSMANAGE=Y` The default value for this parameter is `N`.
`TRANSCOMMITTIME`	The `TRANSCOMMITTIME` parameter specifies when the engine sends a commit call to the WSMQ Connector. Use one of the following values: • `CUSTOMERANDTRANSACTION`—The engine commits the transaction at the end of the customer and at the end of the engine transaction. • `CUSTOMERONLY`—The engine commits the transaction at the end of the customer. • `TRANSACTIONONLY`—The engine commits the transaction at the end of the engine transaction. • `NEVER`—The engine will never commits the transaction. **Example:** `TRANSCOMMITTIME=NEVER` The default value for this parameter is `CUSTOMERANDTRANSACTION`.
`WAITINTERVAL`	The `WAITINTERVAL` parameter specifies the number of seconds that the WSMQ Connector waits for a message to be available on the input queue. The value indicates the polling interval for the `MQGET` call. Each time the WSMQ Connector performs a read, it calls `MQGET` with the number of seconds specified in the `WAITINTERVAL` parameter. The `MQGET` call waits for a message to arrive for up to the specifies number of seconds in the `WAITINTERVAL` parameter before returning a failure. **Example:** `WAITINTERVAL=10` The default value for this parameter is `10`.

Secure Channel Parameters for the WSMQ Connector

If you want to use a secure channel to transmit encrypted messages, you can set additional initialization parameters. To specify secure channel parameters for the WSMQ Connector using a Secure Sockets Layer (SSL) protocol, you must include the following parameters.

SSL initialization parameters, the WSMQ Connector

Parameter	Description
`CHANNELNAME`	The `CHANNELNAME` parameter specifies the name of the channel used in the queue manager settings. **Example:** `CHANNELNAME=CHANNAME` There is no default value for this parameter.
`CONNECTIONNAME`	The `CONNECTIONNAME` parameter specifies the name of the connection used for the MQ server. The value is dependent on the value specified in the `TRANSPORTTYPE` parameter. **Example:** `CONNECTIONNAME=CONNMAE` There is no default value for this parameter.
`SSLCIPHER`	The `SSLCIPHER` parameter specifies the type of SSL cipher used by the application. The value must match the cipher used for the MQ server. You can use any SSL cipher supported by WebSphere MQ. **Example:** `SSLCIPHER=NULL_MD5` There is no default value for this parameter. For information on the SSL ciphers supported by WebSphere MQ, see the IBM website: http://www-306.ibm.com/software/integration/wmq/index.html.
`SSLKEYREPOSITORY`	The `SSLKEYREPOSITORY` parameter specifies the path name to the local directory that contains the queue manager key. **Example:** `SSLKEYREPOSITORY=C:\Sample\SSL` There is no default value for this parameter. The `SSLKEYREPOSITORY` parameter supports the use of DBCS characters.
`TRANSPORTTYPE`	The `TRANSPORTTYPE` parameter specifies the type of protocol used to connect to the MQ server. Select from one of the following values: • `LU62`—The `LU62` value specifies the `LU62` protocol. • `NETBIOS`—The `NETBIOS` value specifies the `NETBIOS` protocol. • `SPX`—The `SPX` value specifies the `SPX` protocol. • `TCP`—The `TCP` value specifies the `TCP` protocol. **Example:** `TRANSPORTTYPE=NETBIOS` There is no default value fro this parameter.

8.2.4 Setting Engine Commands for WSMQ Connector Output Queues

In addition to your basic configuration of the WSMQ Connector, you can also set engine commands for the output queues that are assigned to the WSMQ Connector. The engine commands that you set for the output queues communicate with the DDA routines that you write for the WSMQ Connector. Controlling the workflow process helps you establish an efficient configuration when running multiple engines simultaneously, using correlation mode, and for dynamic memory allocation.

To control how the engine commands work with the WSMQ Connector, you must specify the `PROCESS_CONTROL_MESSAGES` parameter in your initialization file.

For more information about the `PROCESS_CONTROL_MESSAGES` parameter, see "Optional Parameters for the WSMQ Connector" on page 137.

You can use the following engine control messages with the output queue:

- `STOP`—The `STOP` engine command instructs the DDA routine to shut down. The DDA routine places the message back on the queue, so that other active DDA routines are able to receive the `STOP` message. The DDA routine adds the message to the queue with an expiration of 60 seconds. The `STOP` operator clears the message before restarting the engine(s) if the engine (s) must be restarted within 60 seconds of the last engine shutting down.
- `SLEEP #ss`—The `SLEEP #ss` engine command instructs the engine to sleep for a specified number of seconds. The `SLEEP #ss` engine command requires that the DDA routine place a `SLEEP` control message back on the queue, so other currently running engines receive the same `SLEEP` message. The message ID appends to the `SLEEP` control message before the WSMQ Connector places the message back on the queue. The WSMQ Connector extracts the expiry time from the original sleep message. If the expiry time is non-zero, the WSMQ Connector reduces the expiry time by a second and uses the expiry time for the message added to the queue. If the expiry time is zero, the WSMQ Connector uses the number of seconds in the `SLEEP` control message as the expiry time. The WSMQ Connector places all control messages back in queue and the queue times out, so that all engine(s) reading from the queue can receive the control messages.
- `SHUTDOWN`—The `SHUTDOWN` engine command instructs the DDA routine to perform a delayed shutdown. The shutdown does not process until the WSMQ Connector clears all other messages from the input queue. The shutdown occurs when the call to `MQGET` returns a `2033` code, meaning there are no messages available on the queue. As with a `STOP`, the DDA routine is responsible for placing the `SHUTDOWN` message back on the queue. Expiry time is queue-specific. A message expires after a default time of 60 seconds, so that `SHUTDOWN` messages do not persist after the engine restarts.

WebSphere messages are given priorities of 0–9. Control messages have a priority of 9, while all print messages have a priority of 0–8.

8.2.5 Enabling Correlation Mode for the WSMQ Connector and Multiple Engines

You can enable correlation mode for the WSMQ Connector to coordinate incoming data from one input for distribution to multiple outputs. Correlation mode is a specialized mode of operation involving several instances of the Exstream engine.

To enable correlation mode for the WSMQ Connector, you must specify the `CORRELATE` or `CORRELATIONREPLY` parameters in your initialization file. When you set the `CORRELATE` or `CORRELATIONREPLY` parameters for the WSMQ Connector to run in correlation mode, each instance of the engine follows a specific sequence.

For more information about the parameters required to enable correlation mode for the WSMQ Connector, see "Optional Parameters for the WSMQ Connector" on page 137.

The engine follows the following synchronization sequence while using the WSMQ Connector in this mode:

1. The engine requests a `read` from the WSMQ Connector.
2. If the engine posts `read`, the WSMQ Connector reads from the input queue and retrieves the `MSGID|CORRELID`, and the `REPLYTOQ/REPLYTOQMGR`.
3. If the engine posts a second `read`, the WSMQ Connector returns `EOF, continue Engine Processing`. The exception to this operation is when the message is extremely long (over 32K in length), and the subsequent reads are posted to obtain the remaining message data.
4. If the engine posts a third `read`, the WSMQ Connector writes a `NULL` message to the output queue, clears `MSGID|CORRELID` and or `REPLYTOQ|REPLYTOQMGR`, and returns to step `1`. The `NULL` message to the output queue notifies the requestor that no output was produced.
5. If the engine posts `write`, the WSMQ Connector writes to the output queue or the specified `REPLYTOQ`, clears the `MSGID/CORRELID`, and/or `REPLYTOQ/REPLYTOQMGR`, and returns to step 3.

Note: Correlation mode works with both single and multiple output file operation.

DDA Functions for Correlation Mode

When you create the custom DDA routine to handle correlation mode, you must specify a certain DDA functions to handle the process of sending output to multiple queues. Each function provides a specific method for handling aspects of the correlation mode process.

For more information about exporting DDA functions, see "DDA Routine Functions" on page 39.

This section discusses the following topics:

- “The correlIDCallback Function” below
- “The correlIDUpdate Function” below
- “The setCorrelID Function” below

The correlIDCallback Function

If you correlate input messages with output or log messages, then you can export the `correlIDCallback` function so that the engine can pass a pointer to the `setCorrelID` function of the DDA routine.

The only parameter that the engine passes to `correlIDCallback` is a function pointer to the `setCorrelID` function. The engine can call the `setCorrelID` function any time that the correlation ID requires updating.

When you export the `correlIDCallback` function, the engine calls the `correlIDCallback` function during the initialization of the DDA routine before calling `getRec`.

Syntax:

```
correlIDCallback(void (__stdcall *pSetCorrelID)(BYTE*, unsigned long))
```

The correlIDUpdate Function

If you correlate input messages with output or log messages, you must export the `correlIDUpdate` function so that the engine can pass the correlation ID to the DDA routine.

The two parameters that the engine passes to the `correlIDUpdate` function are the correlation ID (`pCorrelID`) and its size (`iSize`, in bytes).

When you export the `correlIDUpdate` function, the engine calls the `correlIDUpdate` function before calling `getRec`. The engine calls the `correlIDUpdate` function each time the engine generates output or passes a correlation ID.

Syntax:

```
correlIDUpdate(BYTE *pCorrelID, unsigned long iSize)
```

The setCorrelID Function

The DDA routine can call the `setCorrelID` function at any time to set the correlation ID. When the engine calls the `setCorrelID` function to set the correlation ID, the correlation ID changes.

The parameters for `setCorrelID` are the same as those for the `correlIDUpdate` function.

Syntax:

```
setCorrelID(BYTE *pCorrelID, unsigned long iSize)
```

Note: The engine passes the DDA routine a pointer to the `SetCorrelID` function through the `correlIDCallback` function.

8.2.6 Supporting Reference File Lookups in the Messaging Service

You can create DDA routines to utilize the WebSphere MQ Queue Manager and support reference file lookups using key data. Using a reference file DDA routine to run the WSMQ Connector lets you specify search criteria so that the WSMQ Connector can search for specific sets of incoming data from the message queue to distribute to the output queue.

In addition to creating a custom reference file DDA routine, you must also specify the following parameters in your initialization file:

- `INPUTQUEUE`
- `REPLYTOQUEUE`
- `LOOKUPWAITINTERVAL`
- `LOOKUP`

For more information about the parameters that you must use for reference file lookups, see "WSMQ Connector Parameters" on page 136

Keep in mind the following order of operations when you use the WSMQ Connector as a reference file:

1. The DDA routine receives an initial open call. The open parameters are parsed and the options are set by the initialization file.
2. The input queue opens for `get|put` access.
3. For each lookup, the DDA routine is called with an access type of `SEEK` and passed the lookup key.

   Keep in mind the processes for the following methods when you use the WSMQ Connector for reference file lookups:

   - **Default Method**—The default method places the key in the input queue using the delivery confirmation option that you specify. The confirmation message instructs the MQ Series Queue Manager to create a report message and place it on the `REPLYTOQUEUE` correlation parameter. The message is assigned the message ID copied into the correlation ID field so the WSMQ Connector input (flag `MQRO_COPY_MSG_ID_TO_CORREL_ID`) can recognize it. The input module waits for a confirmation message from the `REPLYTOQUEUE`. This confirmation message is available when the key is read by another application. After delivery is confirmed by the report message, the input module then grabs the returned data message off the queue. The input module returns the data length to the engine and an `EOF` status is returned.

- **Correlation ID Method**—The key is placed in the input queue as a `REQUEST` message, with the option set to `MQRO_COPY_MSG_ID_TO_CORREL_ID`. The `REPLYTOQUEUE` is set to either the specified `REPLYTOQUEUE` or the `INPUTQUEUE`. After putting the key in the queue, the input module attempts to get the return message from the known correlation ID with a wait interval of 10 seconds (or the number of seconds specified with `LOOKUPWAITINTERVAL=#`). The input module returns the data length to the engine and an `EOF` status returns. You specify this method by using the `LOOKUP=CORREL` parameter and value in the initialization file, or in the **Open parameters** box in Design Manager when you set the WSMQ Connector properties.

4. The DDA routine is called repeatedly with an access type of `READ`. The lookup data is returned in the buffer and the length in the buffer size field, both of which send a return code `0` (indicating returned data) to the engine.
5. After all data has been returned, the DDA routine returns a code on an `EOF` call to the engine to signify that the lookup data is complete.
6. After all lookups have completed, the DDA routine then passes the final close call, and the input queue closes.

8.3 Testing the WSMQ Connector

For best results, before using the WSMQ Connector in production, you should test your WSMQ Connector configuration. Testing the configuration of the WSMQ Connector helps to prevent errors that can occur during production runs.

To test the WSMQ Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - All data source files to be used with the connector
   - Initialization file with specified parameters
2. Verify that you have assigned all relevant data files to the connector object.
3. Verify that you have assigned an output to the connector object.
4. Package your application and run the engine with the ONDEMAND switch in your control file.
5. Verify that your output is correct based on the configuration of the WSMQ Connector.

Chapter 9: The Java Database Connectivity Connector

The Java Database Connectivity (JDBC) Connector lets the engine communicate and interact with databases that use Java Database Connectivity. Java Database Connectivity is java-based data access technology. The JDBC Connector communicates with a JDBC-compliant database, such as DB2, or from an Oracle stored procedure to retrieve results which are then passed to the engine for document creation and fulfillment. You can initiate the JDBC Connector connected to the engine as a DDA, or in a local standalone mode for testing purposes.

The JDBC Connector lets you do the following:

- Return or store data in XML or delimited format.
- Retrieve or store binary content from a database.
- Retrieve results from basic SQL statement, such as SELECT, INSERT, and DELETE, including advanced modifiers such as HAVING, ORDER BY, and CUBE.
- Support a JNDI connection that can be used for application servers such as IBM WebSphere Application Server.

For example, suppose you want to integrate Exstream Design and Production with your existing JDBC-compliant database to take advantage of the on-demand capabilities of Exstream. You employ a solution where developers producing applications using Exstream Design and Production can retrieve customer information from your SQL database on a per request basis. You customize and configure the JDBC Connector to retrieve data from your JDBC-compliant database to create documents from applications in real time.

This chapter discusses the following topics:

- "Preparing the JDBC Connector" below
- "Configuring the JDBC Connector" on the next page
- "Testing the JDBC Connector" on page 185

9.1 Preparing the JDBC Connector

To prepare the JDBC Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the `JdbcConnector_<Version_#>_<Release>.zip` file.
2. Extract the contents of the `JdbcConnector_<Version_#>_<Release>.zip` file to the

directory that contains the Exstream engine.

3. If you are using a JDBC driver to connect your database and the JDBC Connector, refer to your database vendor documentation for the latest information regarding JDBC drivers and JAR files required to run the drivers and ensure a connection. JDBC drivers can be downloaded from the following websites:

    - IBM DB2 JDBC drivers are available at http://ww.ibm.com.
    - Microsoft SQL Server JDBC drivers are available at http://www.microsoft.com.
    - Oracle JDBC drivers are available at http://www.oracle.com.

    For more information about configuring a JDBC driver for use with the JDBC Connector, see "File Type Parameters" on page 168.

9.2 Configuring the JDBC Connector

You can configure the JDBC Connector in standalone mode, or in real-time mode. As a real-time solution, the JDBC Connector can request multiple sets of information from different databases, returning results in multiple formats. When you configure and implement the JDBC Connector in standalone mode, you can test your set up and configuration. In both the standalone configuration and the real-time configuration, you must first use a call format that defines the tasks that the JDBC Connector performs.

To configure the JDBC Connector, you must complete the following tasks:

1. "Configure the JDBC Connector in Design Manager" on the next page
2. "Developing a Call Format for the JDBC Connector to Call a Database" on page 158
3. "Configuring the JDBC Connector to Run in Standalone Mode" on page 161
4. "Setting Up Your Log File" on page 163
5. "Setting Up Your Initialization File" on page 164
6. "JDBC Connector Parameters" on page 165

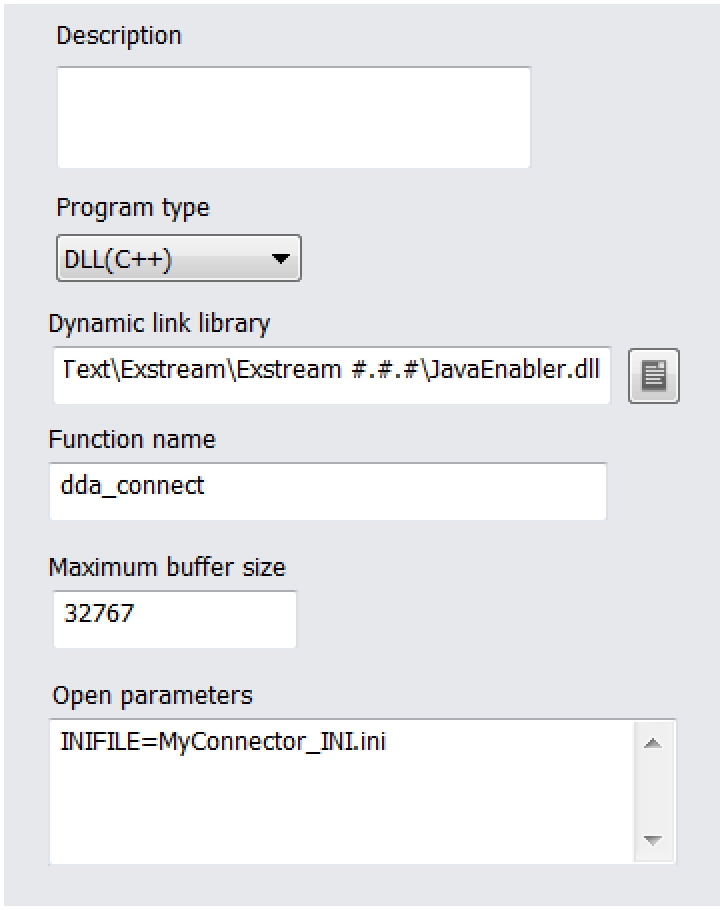
9.2.1 Configure the JDBC Connector in Design Manager

To configure the JDBC Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the Java Connector DLL, or click to go to the file: For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on page 164.

Example of the completed connector object properties for the JDBC Connector


Description

Program type
DLL(C++)

Dynamic link library
Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name
dda_connect

Maximum buffer size
32767

Open parameters
INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Connectors in Design Manager" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning the connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to a placeholder variable.

   For more information about assigning the connector to a placeholder variable, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

4. Assign the connector to an output.

   For more information about assigning the connector to an output, see "Assigning a Connector Object to an Output" on page 27.

5. Assign the connector to an output queue.

For more information about assigning the connector to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

9.2.2 Developing a Call Format for the JDBC Connector to Call a Database

The JDBC Connector searches for data in a database and returns these database results based on a set of parameters that you define. Because you can configure the JDBC Connector to call multiple sets of data from a database and return this data in a number of formats, you must develop a call format that you can use to pass information to the JDBC Connector at run time. The call format for the JDBC Connector is a string, referenced as a seek key. The seek key specifies the type of database calls that the engine makes using the JDBC Connector.

This section discusses the following topics:

- "Formatting the Seek Key for the JDBC Connector" below
- "Formatting the Seek Key for the JDBC Connector File Types" on page 160

Formatting the Seek Key for the JDBC Connector

The seek key indicates which SQL statement to execute. Additionally, the seek key contains the SQL parameters required for a SQL execution. The production goals that you want to achieve with the JDBC Connector determines how you pass the seek key to the JDBC Connector at run time.

The type of functionality a seek key provides is no different than any other connector, however the JDBC Connector requires additional specialized configurations depending on your initialization file set up. Design Manager does not create a seek key for you, therefore you must build a formula variable. In the formula variable that you create for the seek key, make sure you include the name of the SQL statement, a delimiter to indicate separate result sets, and the SQL parameters required for a SQL execution.

You can format the seek key using a prepared SQL format, or a generic SQL format.

For more information about how to build a formula variable, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

This section discusses the following topics:

- "Formatting a Prepared SQL Seek Key" on the next page
- "Formatting a Generic SQL Seek Key" on the next page
- "JDBC Connector Data Type Format for the Seek Key" on the next page

Formatting a Prepared SQL Seek Key

A prepared SQL format allows you to configure the seek key for multiple SQL statements by way of specifying parameters whose values are determined based on initialization parameters. Prepared SQL uses the following format:

```
<sqlKey><delimiter><param1><delimiter><param2>
```

Each value corresponds to a specified parameter in the initialization file. The following table describes each parameter required for a prepared SQL format:

Prepared SQL parameters for the JDBC Connector seek key

Parameter	Description
`<sqlKey>`	The `<sqlKey>` value must match the value defined in the initialization file for the `SQL.#.KEY` parameter. For more information about the `SQL.#.KEY` parameter, see "SQL Definition Parameters" on page 180.
`<delimiter>`	The `<delimiter>` value must match the value defined in the initialization file for the `KEY.DELIMITER` parameter. For more information about the `KEY.DELIMITER` parameter, see "Delimited Result Parameters" on page 178.
`<param#>`	The `<param#>` value must match the value defined in the initialization file for the `SQL.#.IN.#.PARAM` parameter. The first instance of the `<param#>` attribute in the seek key indicates the lowest defined input parameter specified in the initialization file for the SQL statement. For example, if the parameters defined in your initialization file are for parameters `3`, `4`, and `5`, then the first `<param#>` value in the seek key would be `3`, the second `<param#>` value in the seek key would be `4`, and the third `<param#>` value in the seek key would be `5`. For more information about the `SQL.#.IN.#.PARAM` parameter, see "SQL Definition Parameters" on page 180.

Formatting a Generic SQL Seek Key

A generic SQL format lets you configure a seek key without corresponding initialization parameters. When configuring a seek key in a generic SQL format, you must make sure that the SQL contains the entire SQL statement with parameters included. Generic SQL uses the following format:

```
generic<delimiter><sqlStatement>
```

The following table describes each attribute required for a generic SQL format:

Generic SQL parameters for the JDBC Connector seek key

Attribute	Description
`generic`	The "generic" string is a hard-coded attribute that indicates the seek key is in generic SQL format.
`<delimiter>`	The `<delimiter>` value is the delimiter defined in the initialization file.
`<sqlStatement>`	The `<sqlStatement>` value is the full SQL statement to execute.

JDBC Connector Data Type Format for the Seek Key

The JDBC specific format for data types used in the seek key string value are as follows:

- **String**—No special formatting required
- **Integer**—The data type format for integer must be a valid integer only
- **Float**—The data type format for float must be a valid number only
- **Date**—The data type format for date must be `yyyy-mm-dd`
- **Time**—The data type format for time must be `hh:mm:ss`
- **Timestamp**—The data type format for timestamp must be `yyyy-mm-dd hh:mm:ss:ffffffff`

  Fractions of seconds (`fffffff`) as noted in the timestamp format are not required for use with the seek key.

Formatting the Seek Key for the JDBC Connector File Types

Because you can configure the JDBC Connector to act as a file type, such as a placeholder or reference file, some additional configurations might be necessary to ensure delivery of the seek key at run time. To pass the seek key to the JDBC Connector at run time, you must first format the string value and create a formula variable in Design Manager. A formula variable is sufficient for most actions that the JDBC Connector makes, however some file types require that you deliver the seek key in different formats.

Each file type that you configure to accept and subsequently pass the seek key to the JDBC Connector requires that you set initialization parameters that correspond to the configurations and values of the seek key.

For more information about initialization files that correspond with the seek key, see "File Type Parameters" on page 168.

To pass the seek key to the JDBC Connector as a specific file type, you must format the seek key according to the file type you choose:

To	Do this
Format the seek key for JDBC Connector as a reference file	Set the seek key as a user variable, then create a formula variable that sets the user variable to be the seek key for the JDBC Connector to be a reference file. For a DDA, you must set the seek key as a single variable. For more information about reference files and reference key layouts, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.
Format the seek key for JDBC Connector as a placeholder	Set the value of the placeholder variable to be the string value that makes up the seek key. For more information about placeholder variables, see *Importing External Content* in the Exstream Design and Production documentation.
Format the seek key for JDBC Connector as a report file	Set the format of the report file to match the format and value of the seek key. For more information about report files, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

To	Do this
Format the seek key for JDBC Connector as an output queue (DDA output)	Set the value of the 'SYS_DDAOutputUserData' variable to be the string value that makes up the seek key. When passing the seek key to JDBC Connector as an output queue, you must make sure you enable DDA output headers. You must also make sure that the output is set to use a DDA (in this case, JDBC Connector), which uses an output queue connector (Design Manager connector object) or uses the DDAOUTPUT engine switch. For more information about DDA output headers and the 'SYS_DDAOutputUserData' system variable, see <WILL BE XREF TO DDA CHAPTER For more information about the DDAOUTPUT engine switch, see "Configuring DDA Routines Using a Control File" on page 28.

9.2.3 Configuring the JDBC Connector to Run in Standalone Mode

You can use the JDBC Connector in standalone mode to run the connector without the engine. In standalone mode, you can test your connector configurations, such as database connection configurations and your initialization file parameter configurations. By testing your connector configurations, you can significantly reduce production down scenarios that result from faulty configurations. Running the JDBC Connector in standalone mode also provides a best practice for ensuring that you receive the correct results from the database you are calling.

To configure the JDBC Connector to run in standalone mode, you must complete the following tasks:

1. "Initiating the JDBC Connector in Standalone Mode" below
2. "Running the JDBC Connector in Standalone Mode" on the next page

Initiating the JDBC Connector in Standalone Mode

To initiate the JDBC Connector in standalone mode, you must perform the following steps:

1. Set your CLASSPATH to include the following files:
   - `JdbcConnector.jar`
   - `log4j.properties`
   - `JaveEnabler.jar`
   - Database JDBC JAR files

   For more information about log4j files, see "Setting Up Your Log File" on page 163.
2. Run the following command in the console window to initiate the JDBC Connector in standalone mode:

   ```
   java com.exstream.dda.jdbc.JdbcConnector iniFile [-encrypt]]
   ```

3. If your connector parameter initialization files contain double-byte characters, you must add the `encoding` configuration switch to the command structure you enter in the console window.

   The following example shows all encoding possibilities:

   ```
   java com.exstream.dda.jdbc.JdbcConnector iniFile [-encoding [ISO-8859-1 |
   UTF–8 | UTF–16BE | UTF–16LE]] [[resultFile]] | [-encrypt]]
   ```

   The `encoding` switch specifies the character encoding used in the connector parameter initialization file. The default argument for this switch is `ISO–8859–1`.

Running the JDBC Connector in Standalone Mode

You can run the JDBC Connector in standalone mode to do the following:

- **Encrypt JDBC User Name and Password**—Encrypting the JDBC user name and password allows the initialization file to contain an encrypted user name and password.
- **Retrieve Results for Local Storage**—Retrieving results for local storage allows a SQL statement to be executed and the results stored in a local file, such as a text file.
- **Testing**—Testing allows SQL statements to be executed to validate the SQL and the results.

To run the JDBC Connector in standalone mode to complete specific tasks, you can do the following:

To	Do this
Encrypt the user name and password in an initialization file	Enter the following command: `java com.exstream.dda.jdbc.JdbcConnector iniFile -encrypt` The encryption command checks the initialization file to verify that the `ENCRYPT` parameter equals `true` and that the `ENCRYPTED` parameter equals `false`. If `ENCRYPT=true` and `ENCRYPTED=false`, the user name and password becomes encrypted. For more information about the `ENCRYPT` and `ENCRYPTED` parameters, see "Encryption Parameters" on page 167.

To	Do this
Run the JDBC Connector to store results to a local file	Enter the following command: `java com.exstream.dda.jdbc.JdbcConnector iniFile -seek seekKey [resultFile]` The JDBC Connector runs based on the initialization file using the seek key. If you want to write output to a specific location, you must specify a file location in the output file parameter. If you do not specify a file path in the output file parameter, the output displays on the console. For more information about the seek key, see "Formatting the Seek Key for the JDBC Connector" on page 158.
Test the JDBC Connector the initialization file or against SQL statements	Enter the following command: `java com.exstream.dda.jdbc.JdbcConnector iniFile -seek seekKey [resultFile]` The command is the same command used for retrieving results and storing them in a local file. However, for testing purposes, this command tests the CLASSPATH, database connection, parameters, and the configuration of your initialization files. For best results, use the testing features before you begin producing applications to prevent unnecessary downtime and/or errors that prevent successful production runs.

9.2.4 Setting Up Your Log File

Logging lets you configure a record log that you can use to monitor the behavior of applications produced by the engine and the JDBC Connector. You can set up logging to prevent errors in your output. To use logging, configure the standard `log4j` configuration file provided by the Apache community. All parameter definitions follow the standard `log4j.properties` file specifications. You must extract, modify, and then compress the `log4j.properties` file back to the JAR file.

To set up your log file, you must configure the standard `log4j.properties` file located inside the JAR executable archive:

1. Extract the contacts of the `log4j.properties` file.

   The `log4j.properties` file is located in the `JdbcConnector.jar` file of the `JdbcConnector.zip` file.

2. Modify the following `log4j.properties` parameters according as necessary:

   log4j.properties configuration file parameters for the JDBC Connector

Parameter	Description
`log4j.rootLogger`	Specifies which loggers to use in the logging process and the minimal threshold of the loggers (specified as a first parameter)
`log4j.appender.stdout.threshold`	Specifies the threshold for the console (`stdout`) logger and which messages to show on the console
`log4j.appender.debugFile.File`	Specifies the file name of the debug-logger file that contains all the messages from the module)
`log4j.appender.debugFile.threshold`	Specifies the threshold of the debug-logger file
`log4j.appender.errorFile.File`	Specifies the name of the file in which to log the error and warning messages. This file must already exist.
`log4j.appender.errorFile.threshold`	Specifies the threshold of the error-logger file

3. Compress the `log4j.properties` file back to the `JdbcConnector.jar` file.
4. Back up your `log4j.properties` file before you upgrade the JDBC Connector module, so you can reuse it with the later version.

9.2.5 Setting Up Your Initialization File

After you configure the JDBC Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager as opposed to listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

To create an initialization file for the JDBC Connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization

file in the **Open parameters** box.

5. Save the connector object.

For more information about parameters for the JDBC Connector, see “JDBC Connector Parameters” below.

9.2.6 JDBC Connector Parameters

The JDBC Connector contains a number of different parameters that you can utilize to meet your production goals, business requirements, and other objectives. For example, you can specify the method by which you connect to a database, in which format you receive database result sets, and what credentials are required to connect to a databases.

This section discusses the following topics:

- “JDBC Connection Parameters” below
- “Encryption Parameters” on page 167
- “File Type Parameters” on page 168
- “XML Result Parameters” on page 172
- “Delimited Result Parameters” on page 178
- “SQL Definition Parameters” on page 180
- “Key Parameters” on page 184

JDBC Connection Parameters

In order for the JDBC Connector to establish a connection to a JDBC-compliant database, you must specify the appropriate method that the JDBC Connector uses to connect to your database. For example, if you are using a JNDI connection, you would specify a value for the `JNDI_CONNECTION` parameter. If you are using a JDBC driver to connect to a database, you would specify values for other parameters that correspond with the type of JDBC driver that you are using in your infrastructure.

To configure the JDBC connection, specify values for the JDBC connection initialization parameters.

JDBC Connection initialization parameters

Parameter	Description
`JNDI_CONNECTION`	The `JNDI_CONNECTION` parameter specifies the JNDI name used to retrieve a JDBC data source. Example: `JNDI_CONNECTION=jndi/data_source_name` There is no default value for this parameter. Use the `JNDI_CONNECTION` parameter as a required parameter when using a JNDI connection to a data source. If you use a connection service other than a JNDI connection, you must remove the parameter from your initialization file.
`DRIVER_CLASS`	The `DRIVER_CLASS` parameter specifies the full Java class name when the JDBC Connector manages a JDBC connection. Example: `DRIVER_CLASS=COM.ibm.db2.jdbc.app.DB2Driver` There is no default value for this parameter. For the full JDBC driver Java class name, you must refer to your database vendor's documentation. The `DRIVER_CLASS` parameter is required only when the JDBC Connector manages the JDBC connection. This parameter is not required when using a JNDI connection.
`CONNECTION_URL`	The `CONNECTION_URL` parameter specifies the connection URL when the JDBC Connector manages the JDBC connection. The connection URL is the value and format required by your database provider's JDBC driver. Example: `CONNECTION_URL=jdbc:db2://www.example.com:50000/DB` There is no default value for this parameter. For the full JDBC driver connection URL value and format, you must refer to your database vendor's documentation. The `CONNECTION_URL` parameter is required only when the JDBC Connector manages the JDBC connection.

JDBC Connection initialization parameters, continued

Parameter	Description
`USERNAME`	The `USERNAME` parameter specifies the user name for the JDBC connection. The user name value is your database user name. Example: `USERNAME=admin` There is no default value for this parameter.. The `USERNAME` parameter is required only when the JDBC Connector manages the JDBC connection. The value for this parameter can be encrypted. For more information about encrypting the user name, see "Configuring the JDBC Connector to Run in Standalone Mode" on page 161.
`PASSWORD`	The `PASSWORD` parameter specifies the password for the JDBC connection. The parameter value is your database password. Example: `PASSWORD=admin` There is no default value for this parameter. The `PASSWORD` parameter is required only when the JDBC Connector manages the JDBC connection. For more information about encrypting the user name, see "Configuring the JDBC Connector to Run in Standalone Mode" on page 161.
`KEEP_CONNECTION=TRUE|FALSE`	The `KEEP_CONNECTION` parameter specifies whether to maintain a database connection between database calls. Maintaining a database connection improves the performance of the JDBC Connector. Example: `KEEP_CONNECTION=TRUE` The default value for this parameter is `FALSE`. For best results, if you are using a JNDI connection, specify `FALSE` for the `KEEP_CONNECTION` parameter because JNDI typically handles connection pooling internally. **Note:** If you specify `TRUE` for the `KEEP_CONNECTION` parameter and you experience a network timeout or a loss of network connectivity, the database connection can become orphaned on the server, which prevents the JDBC Connector from closing the connection.

Encryption Parameters

The JDBC Connector allows you to set encryption parameters for the user name and password. You must define the encryption parameters in your initialization file and run the JDBC Connector in standalone mode.

For information about using the JDBC Connector in standalone mode, see "Configuring the JDBC Connector to Run in Standalone Mode" on page 161.

To configure standalone mode, specify values for the encryption parameters.

Encryption parameters for the JDBC Connector

Parameter	Description
`ENCRYPT=true|false|Y|N|YES|NO`	Use the `ENCRYPT` parameter to specify whether to encrypt user name and password information for the JDBC Connector. Use the `ENCRYPT` parameter only when running the JDBC Connector in standalone mode. After you have encrypted your credentials, you must set the value back to `false`, N, or NO to avoid errors. **Example:** `ENCRYPT=true` The default value for the this parameter is `false`.
`ENCRYPTED=true|false|Y|N|YES|NO`	The JDBC Connector specifies the value of the `ENCRYPTED` parameter based on the value specified in the `ENCRYPT` parameter. **Example:** `ENCRYPTED=true` There is no default value for this parameter. If you specify `true`, Y, or YES as the value for the `ENCRYPT` parameter and run the JDBC Connector in standalone mode for encryption, then the JDBC Connector automatically sets the first parameter in the initialization file.

In the following example, a user name and password are encrypted based on the value set in the ENCRYPT parameter:

```
ENCRYPT=true
ENCRYPTED=false
USERNAME=myusername
PASSWORD=mypassword
ENCRYPT=true
ENCRYPTED=true
USERNAME=3NRo7sjXSq8rpdsH51x0IQ
PASSWORD=3NRo7sjXSq8rpdsH51x0IQ
```

File Type Parameters

You can configure the JDBC Connector to communicate with types of routines that enable the connector to act as a file type. For example, if you want the JDBC Connector to act as a placeholder variable, you assign the connector to a placeholder variable as part of your connector configuration. For certain file types, you must specify a parameter value.

The following sections discuss the parameters required to set up the JDBC Connector as a file type:

- "Driver File Mode Parameter" below
- "Placeholder Mode Parameter" below
- "Output Queue Mode Parameters" below

Driver File Mode Parameter

To configure the driver file mode, specify a value for the driver file mode parameter.

Optional driver file parameter for the JDBC Connector

Parameter	Description
`DRIVER=true|false`	The `DRIVER` parameter specifies the JDBC Connector as a driver file. If you want to use the JDBC Connector as a driver file, specify `true`. **Example:** `DRIVER=true` The default value for this parameter is `false`.

For more information about using connectors as driver files, see "Assigning a Connector Object to a Data File for Data Input" on page 24

Placeholder Mode Parameter

To configure the placeholder mode, specify a value for the placeholder mode parameter.

Optional placeholder file parameter for the JDBC Connector

Parameter	Description
`PLACEHOLDER=true|false`	The `PLACEHOLDER` parameter specifies the JDBC Connector as a placeholder file. If you want to use the JDBC Connector as a placeholder file, specify `true`. **Example:** `PLACEHOLDER=true` The default value for this parameter is `false`.

For more information about using the JDBC Connector as a placeholder file, see "Configure the JDBC Connector in Design Manager" on page 156.

Output Queue Mode Parameters

You can use output queue parameters to specify how to store output using the JDBC Connector. For output integration, you must pass the seek key for the JDBC Connector in the Exstream system variable 'SYS_DDAOutputUserData. You must make sure that you enter the parameter values within the <> symbols for all output queue parameters, except the `OUTPUT` parameter.

For more information about seek key configurations, see "Formatting the Seek Key for the JDBC Connector File Types" on page 160

For more information about DDA Output, see "DDA Routine Output Headers" on page 40.

To configure the output queue mode, specify values for the output queue mode parameters:

Optional output queue file parameters for the JDBC Connector

Parameter	Description
`OUTPUT=true|false`	The `OUTPUT` parameter specifies whether the JDBC Connector writes output to the database. If the value of the `OUTPUT` parameter is set to `false`, the engine writes output in report file mode. The value of the `OUTPUT` parameter must be set to `true` for the engine to write output to the database. **Example:** `OUTPUT=true` The default value for this parameter is `false`. The `OUTPUT` parameter is not required.
`OUTPUT.INDICATOR.OUTPUT`	The `OUTPUT.INDICATOR.OUTPUT` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the DDA output. **Example:** `OUTPUT.INDICATOR.OUTPUT=<output>` The default value for this parameter is `<output>`. This `OUTPUT.INDICATOR.OUTPUT` parameter is not required.
`OUTPUT.INDICATOR.MESSAGES`	The `OUTPUT.INDICATOR.MESSAGES` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the engine messages. **Example:** `OUTPUT.INDICATOR.MESSAGES=<messages>` The default value for this parameter is `<messages>`. This `OUTPUT.INDICATOR.MESSAGES` parameter is not required.
`OUTPUT.INDICATOR.USER.DATA`	The `OUTPUT.INDICATOR.USER.DATA` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the DDA output user data. **Example:** `OUTPUT.INDICATOR.USER.DATA=<user data>` The default value for this parameter is `<user.data>`. This `OUTPUT.INDICATOR.USER.DATA` parameter is not required.

Optional output queue file parameters for the JDBC Connector, continued

Parameter	Description
`OUTPUT.INDICATOR.HEADER.SIZE`	The `OUTPUT.INDICATOR.HEADER.SIZE` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the DDA output header size. **Example:** `OUTPUT.INDICATOR.HEADER.SIZE=<header size>` The default value for this parameter is `<header.size>`. This `OUTPUT.INDICATOR.HEADER.SIZE` parameter is not required.
`OUTPUT.INDICATOR.HEADER.VERSION`	The `OUTPUT.INDICATOR.HEADER.VERSION` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the DDA output header version. **Example:** `OUTPUT.INDICATOR.HEADER.VERSION=<header.version>` The default value for this parameter is `<header.version>`. This `OUTPUT.INDICATOR.HEADER.VERSION` parameter is not required.
`OUTPUT.INDICATOR.OUTPUT.DRIVER`	The `OUTPUT.INDICATOR.OUTPUT.DRIVER` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the output driver. **Example:** `OUTPUT.INDICATOR.OUTPUT.DRIVER=<output.driver>` The default value for this parameter is `<output.driver>`. This `OUTPUT.INDICATOR.OUTPUT.DRIVER` parameter is not required.
`OUTPUT.INDICATOR.OUTPUT.SIZE`	The `OUTPUT.INDICATOR.OUTPUT.SIZE` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the output size. **Example:** `OUTPUT.INDICATOR.OUTPUT.SIZE=<output.size>` The default value for this parameter is `<output.size>`. This `OUTPUT.INDICATOR.OUTPUT.SIZE` parameter is not required.
`OUTPUT.INDICATOR.MESSAGE.SIZE`	The `OUTPUT.INDICATOR.MESSAGE.SIZE` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the message size. **Example:** `OUTPUT.INDICATOR.MESSAGE.SIZE=<message.size>` The default value for this parameter is `<message.size>`. This `OUTPUT.INDICATOR.MESSAGE.SIZE` parameter is not required.

Optional output queue file parameters for the JDBC Connector, continued

Parameter	Description
`OUTPUT.INDICATOR.USER.DATA.SIZE`	The `OUTPUT.INDICATOR.USER.DATA.SIZE` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains the DDA output use data size. **Example:** `OUTPUT.INDICATOR.USER.DATA.SIZE=<user.data.size>` The default value for this parameter is `<user.data.size>`. This `OUTPUT.INDICATOR.USER.DATA.SIZE` parameter is not required.
`OUTPUT.INDICATOR.PAGE.COUNT`	The `OUTPUT.INDICATOR.PAGE.COUNT` parameter specifies the string value that the JDBC Connector uses to determine which parameter in the SQL contains the page count. **Example:** `OUTPUT.INDICATOR.PAGE.COUNT=<page.count>` The default value for this parameter is `<page.count>`. This `OUTPUT.INDICATOR.PAGE.COUNT` parameter is not required.
`OUTPUT.INDICATOR.RETURN.CODE`	The `OUTPUT.INDICATOR.RETURN.CODE` parameter specifies the string value for the placeholder that the JDBC Connector uses to determine which parameter in the SQL contains return codes. **Example:** `OUTPUT.INDICATOR.RETURN.CODE=<return code>` The default value for this parameter is `<return.code>`. This `OUTPUT.INDICATOR.RETURN.CODE` parameter is not required.

For more information about using the JDBC Connector as an output queue, see "Configure the JDBC Connector in Design Manager" on page 156.

XML Result Parameters

You can use a XML format with the JDBC Connector to return database results.

Note: If you want to use XML result parameters, you must make sure the `DELIMITED` parameter is set to `false`.

To configure how the XML is formatted to return results, specify values for the XML result parameters.

Optional XML result parameters for the JDBC Connector

Parameter	Description
`XML.ATTR=true|false`	The `XML.ATTR` parameter specifies whether to include attributes in the XML tags. If the `XML.ATTR` parameter is set to `true`, then attributes are enabled and the values given for corresponding XML tags replace the tag name. **Example:** `XML.ATTR=true` The default value for this parameter is `false`. The `XML.ATTR` parameter is not required.
`XML.ATTR.NAME`	If XML attributes are enabled, the `XML.ATTR.NAME` parameter specifies the attribute name for all attributes. **Example:** `XML.ATTR.NAME=type` The default value for this parameter is `type`. The `XML.ATTR.NAME` parameter is not required.
`XML.ENCODING`	The `XML.ENCODING` parameter specifies the output encoding for the XML. **Example:** `XML.ENCODING=ISO-8859-1` The default value for this parameter is `ISO-8859-1` The `XML.ENCODING` parameter is not required.
`XML.CDATA=true|false`	The `XML.CDATA` parameter specifies whether to wrap data in CDATA tags. **Example:** `XML.CDATA=true` If the value of the `XML.CDATA` equals `true`, then CDATA tags contain data returned in the XML. The default value for this parameter is `false`. The `XML.CDATA` parameter is not required.
`XML.ATTR.COLUMN=true|false`	If XML attributes are enabled, the `XML.ATTR.COLUMN` parameter specifies whether the JDBC Connector places attributes in the XML tags for columns. If `XML.ATTR` is set to `true`, the value given for `XML.ATTR.COLUMN` is ignored. **Example:** `XML.ATTR.COLUMN=true` The default value for this parameter is `false` The `XML.ATTR.COLUMN` parameter is not required.

Optional XML result parameters for the JDBC Connector, continued

Parameter	Description
`XML.RESULT`	The `XML.RESULT` parameter specifies the tag name given for the `result` tag. The `result` is the root tag. **Example:** `XML.RESULT=result` The default value for this parameter is `result`. The `XML.RESULT` parameter is not required.
`XML.KEY`	The `XML.KEY` parameter specifies the tag name given for the `key` tag. The value for the `key` tag contains the seek key. **Example:** `XML.KEY=key` The default value for this parameter is `key`. The `XML.KEY` parameter is not required.
`XML.OUTPUT`	The `XML.OUTPUT` parameter specifies the tag name given for the `output` tag. The `output` tag contains the output parameters from a stored procedure call. **Example:** `XML.OUTPUT=output` The default value for this parameter is `output`. The `XML.OUTPUT` parameter is not required.
`XML.GENERATED_KEYS`	The `XML.GENERATED_KEYS` parameter specifies the tag name given for the `generatedKeys` tag. The `generatedKeys` tag contains any generated keys from a SQL statement. **Example:** `XML.GENERATED_KEYS=generateKeys` The default value for this parameter is `generatedKeys`. The `XML.GENERATED_KEYS` parameter is not required.
`XML.RESULT_SET`	The `XML.RESULT_SET` parameter specifies the tag name given for the `resultSet` tag. The `resultSet` tag wraps the tags and tag values of a result set. For example, the `resultSet` tag contains the `row`, `updateCount`, and `column` tags with their corresponding values. **Example:** `XML.RESULT_SET=resultSet` If you are calling a stored procedure on DB2, you must make sure that the stored procedure specifies the proper number of result sets that you expect to be returned. The default value for this parameter is `resultSet`. The `XML.RESULT_SET` parameter is not required.

Optional XML result parameters for the JDBC Connector, continued

Parameter	Description
`XML.UPDATE_COUNT`	The `XML.UPDATE_COUNT` parameter specifies the tag name given for the `updateCount` tag. The `updateCount` tag contains the update count number from a SQL statement. **Example:** `XML.UPDATE_COUNT=updateCount` The default value for this parameter is `updateCount`. The `XML.UPDATE_COUNT` parameter is not required.
`XML.ROW`	The `XML.ROW` parameter specifies the tag name given for the `row` tag. The `<row>` tag contains a row of data. **Example:** `XML.ROW=row` The default value for this parameter is `row`. The `XML.ROW` parameter is not required.
`XML.COLUMN`	The `XML.COLUMN` parameter specifies the tag name for the `column` tag only if XML attributes are enabled. If XML attributes are enabled, the tag name given to the `column` is the name of the column in the database. **Example:** `XML.COLUMN=column` The default value for this parameter is `column`. The `XML.COLUMN` parameter is not required.
`XML.ATTR.RESULT`	If XML attributes are enabled, the `XML.ATTR.RESULT` parameter specifies the attribute value for the `result` tag. **Example:** `XML.ATTR.RESULT=result` The default value for this parameter is `result`. The `XML.ATTR.RESULT` parameter is not required.
`XML.ATTR.KEY`	If XML attributes are enabled, the `XML.ATTR.KEY` parameter specifies the attribute value for the `key` tag. **Example:** `XML.ATTR.KEY=key` The default value for this parameter is `key`. The `XML.ATTR.KEY` parameter is not required.

Optional XML result parameters for the JDBC Connector, continued

Parameter	Description
`XML.ATTR.OUTPUT`	If XML attributes are enabled, the `XML.ATTR.OUTPUT` parameter specifies the attribute value for the `output` tag. **Example:** `XML.ATTR.OUTPUT=output` The default value for this parameter is `output`. The `XML.ATTR.OUTPUT` parameter is not required.
`XML.ATTR.GENERATED_KEYS`	If XML attributes are enabled, the `XML.ATTR.GENERATED_KEYS` parameter specifies the attribute value for the `generatedKeys` tag. **Example:** `XML.ATTR.GENERATED_KEYS=generatedKeys` The default value for this parameter is `generatedKeys`. The `XML.ATTR.GENERATED_KEYS` parameter is not required.
`XML.ATTR.RESULT_SET`	If XML attributes are enabled, the `XML.ATTR.RESULT_SET` parameter specifies the attribute value for the `resultSet` tag. **Example:** `XML.ATTR.RESULT_SET=resultSet` The default value for this parameter is `resultSet`. The `XML.ATTR.RESULT_SET` parameter is not required.
`XML.ATTR.UPDATE_COUNT`	If XML attributes are enabled, the `XML.ATTR.UPDATE_COUNT` parameter specifies the attribute value for the `updateCount` tag. **Example:** `XML.ATTR.UPDATE_COUNT=updateCount` The default value for this parameter is `updateCount`. The `XML.ATTR.UPDATE_COUNT` parameter is not required.
`XML.ATTR.ROW`	If XML attributes are enabled, the `XML.ATTR.ROW` parameter specifies the attribute value for the `row` tag. **Example:** `XML.ATTR.ROW=row` The default value for this parameter is `row`. The `XML.ATTR.ROW` parameter is not required.

The following XML string shows the XML format for results returned by the JDBC Connector using the default values:

```
<result>
   <key>key</key>
   <output1>value</output1>
   ...
   <outputx>value</outputx>
   <generatedKeys>
      <row>
         <keyName>value</keyName>
         ...
         <keyName>value</keyName>
      </row>
         ...
      <row>
         <keyName>value</keyName>
         ...
         <keyName>value</keyName>
      </row>
   </generatedKeys>
   <resultSet1>
      <updateCount>value</updateCount>
      <row>
         <columnName>value</columnName>
         ...
         <columnName>value</columnName>
      </row>
      ...
      <row>
         <columnName>value</columnName>
         ...
         <columnName>value</columnName>
      </row>
   </resultSet1>
   ...
   <resultSetx>
      <updateCount>value</updateCount>
      <row>
         <columnName>value</columnName>
         ...
         <columnName>value</columnName>
      </row>
      ...
      <row>
         <columnName>value</columnName>
         ...
         <columnName>value</columnName>
      </row>
   </resultSetx>
</result>
```

In the previous XML format of results, the name of the table column replaces the `<columnName>` tag.

Delimited Result Parameters

You can use a delimited format with the JDBC Connector to return database results. The database returns data to Exstream in the form of configurable record indicators.

To configure the delimited format, specify values for the delimited result parameters.

Optional delimited result parameters for the JDBC Connector

Parameter	Description
`DELIMITED=true|false`	The `DELIMITED` parameter specifies the use of delimited results. When the `DELIMITED` parameter is set to `true`, it overrides the default XML result format. **Example:** `DELIMITED=true` The default value for this parameter is `false`. The `DELIMITED` parameter is not required.
`DELIMITED.ENCODING`	The `DELIMITED.ENCODING` parameter specifies the output encoding for the delimited file. **Example:** `DELIMITED.ENCODING=ISO-8859-1` The default value for this parameter is `ISO-8859-1`. The `DELIMITED.ENCODING` parameter is not required.
`NEW_LINE`	The `NEW_LINE` parameter specifies the command that indicates a new line at the end of a delimited record. **Example:** `NEW_LINE=\r\n` The default value for this parameter is `\r\n`. The `NEW_LINE` parameter is not required.
`DELIMITED.DELIMITER`	The `DELIMITED.DELIMITER` parameter specifies the delimiter to be used to separate data returned in a delimited record. **Example:** `DELIMITED.DELIMITER=,` The default value for this parameter is (`,`). The `DELIMITED.DELIMITER` parameter is not required.

Optional delimited result parameters for the JDBC Connector, continued

Parameter	Description
`DELIMITED.QUOTE`	The `DELIMITED.QUOTE` parameter specifies a quotation mark that wraps a delimited field if the field contains a delimiter. **Example:** `DELIMITED.QUOTE="DELIMITED.QUOTE="` The default value for this parameter is (`"`). The `DELIMITED.QUOTE` parameter is not required.
`DELIMITED.KEY`	The `DELIMITED.KEY` parameter specifies the record indicator for the seek key. **Example:** `DELIMITED.KEY=01` The default value for this parameter is `01`. The `DELIMITED.KEY` parameter is not required.
`DELIMITED.OUTPUT`	The `DELIMITED.OUTPUT` parameter specifies the record indicator for output parameters returned from a stored procedure. **Example:** `DELIMITED.OUTPUT=10` The default value for this parameter is `10`. The `DELIMITED.OUTPUT` parameter is not required.
`DELIMITED.GENERATED_KEYS`	The `DELIMITED.GENERATED_KEYS` parameter specifies the record indicator for the generated keys returned from a SQL statement. **Example:** `DELIMITED.GENERATED_KEYS=20` The default value for this parameter is `20`. The `DELIMITED.GENERATED_KEYS` parameter is not required.
`DELIMITED.RESULT_SET`	The `DELIMITED.RESULT_SET` parameter specifies the record indicator for a result set. **Example:** `DELIMITED.RESULT_SET=30` The default value for this parameter is `30`. The `DELIMITED.RESULT_SET` parameter is not required.

Optional delimited result parameters for the JDBC Connector, continued

Parameter	Description
`DELIMITED.UPDATE_COUNT`	The `DELIMITED.UPDATE_COUNT` parameter specifies the record indicator for an update count returned from a SQL statement. **Example:** `DELIMITED.UPDATE_COUNT=31` The default value for this parameter is `31`. The `DELIMITED.UPDATE_COUNT` parameter is not required.
`DELIMITED.ROW`	The `DELIMITED.ROW` parameter specifies the record indicator from a row of data in a result set. **Example:** `DELIMITED.ROW=32` The default value for this parameter is `32`. The `DELIMITED.ROW` parameter is not required.

The following record indicators show the delimited format for results returned by the JDBC Connector using the default values:

```
01,key
10,outputValue1
...(one or more output values)
10,outputValueX
20,keyValue
...(one or more generated keys)
20,keyValue
30,1
31,updateCount
32,column1Value,column2Value,...,columnXValue
...(one or more rows of data)
32,column1Value,column2Value,...,columnXValue
...(one or more result sets)
30,X
31,updateCount
32,column1Value,column2Value,...,columnXValue
...(one or more rows of data)
32,column1Value, column2Value,...,colmnXValue
```

SQL Definition Parameters

To receive data from a SQL database, you must configure the SQL definition parameters for the JDBC Connector. The SQL parameters that you define as a part of a seek key configuration are directly related to the SQL initialization parameters that you will define.

For more information about the seek key and seek key SQL formatting, see "Formatting the Seek Key for the JDBC Connector" on page 158.

To configure how the JDBC Connector requests and returns SQL statements, specify values for the SQL definition parameters.

Optional SQL definition parameters for the JDBC Connector

Parameter	Description
`SQL.#.KEY`	The `SQL.#.KEY` parameter specifies the unique name of the SQL statement. The `SQL.#.KEY` parameter is a required parameter. **Example:** `SQL.1.KEY=getAddressInfo` To define the unique name for this parameter, you must replace # in `SQL.#.KEY` with the unique number given for the SQL statement. There is no default value for this parameter. The value for # groups all of the SQL properties for a given statement.
`SQL.#.STMT`	The `SQL.#.STMT` parameter specifies the value of the actual SQL statement with `?` to represent input and output variables. The `SQL.#.STMT` parameter is a required parameter. To specify the SQL statement, replace the # with the value given for the SQL statement. **Example:** `SQL.1.STMT=select name from profile where state = ? and city = ?` There is no default value for this parameter.
`SQL.#.TYPE=stored|sql`	The `SQL.#.TYPE` parameter specifies the type of SQL statement for the given group. The type is either a stored procedure or a regular SQL statement. The `SQL.#.TYPE` parameter is a required parameter. **Example:** `SQL.1.TYPE=stored` There is no default value for this parameter.
`SQL.#.OUT.#.PARAM`	The `SQL.#.OUT.#.PARAM` parameter specifies an output parameter for the SQL stored procedure using the value as the parameter number within the SQL statement. In the `SQL.#.OUT.#.PARAM` parameter, the first # represents the group number. The second # represents a unique output parameter number given for the group. **Example:** `SQL.1.OUT.1.PARAM=3` There is no default value for this parameter. The `SQL.#.OUT.#.PARAM` is not required.

Optional SQL definition parameters for the JDBC Connector, continued

Parameter	Description
`SQL.#.OUT.#.TYPE= string|integer|boolean|date|double|float|long|short|time|timestamp|bigdecimal|cursor|blob`	The `SQL.#.OUT.#.TYPE` parameter specifies the output type for the `SQL.#.OUT.#.TYPE` parameter. In the `SQL.#.OUT.#.TYPE`, the first # represents the specified group. The second # represents the specified output parameter. **Example:** `SQL.1.OUT.1.TYPE=integer` There is no default value for this parameter. The `SQL.#.OUT.#.TYPE` is not required.
`SQL.#.IN.#.PARAM`	The `SQL.#.IN.#.PARAM` parameter specifies an input parameter for the SQL stored procedure using the value as the parameter number within the SQL statement. In the `SQL.#.IN.#.PARAM` parameter, the first # represents the group number. The second # represents a unique input parameter number given for the group. **Example:** `SQL.1.IN.1.PARAM=1` There is no default value for this parameter. The `SQL.#.IN.#.PARAM` is not required.
`SQL.#.IN.#.TYPE= string|integer|boolean|date|double|float|long|short|time|timestamp|bigdecimal|blob`	The `SQL.#.OUT.#.TYPE` parameter specifies the input type for the `SQL.#.IN.#.PARAM` parameter. In the `SQL.#.IN.#.PARAM` parameter, the first # represents the group number. The second # represents a unique output parameter number given for the group. **Example:** `SQL.1.IN.1.TYPE=date` There is no default value for this parameter. The `SQL.#.IN.#.TYPE` is not required.
`SQL.#.GK.#.PARAM`	The `SQL.#.GK.#.PARAM` parameter specifies column name to return from the SQL statement for generated keys value as the column name. In the `SQL.#.GK.#.PARAM` parameter, the first # represents the group number. The second # represents the column name to be returned by the generated keys functionality. **Example:** `SQL.1.GK.1.PARAM=column1` There is no default value for this parameter. The `SQL.#.GK.#.PARAM` is not required.

Optional SQL definition parameters for the JDBC Connector, continued

Parameter	Description
`SQL.#.GK.#.TYPE= string|integer|boolean|date| double|float|long|short|time| timestamp|bigdecimal|blob`	The `SQL.#.GK.#.PARAM` parameter specifies the input type for the `SQL.#.GK.#.PARAM` parameter. In the `SQL.#.GK.#.PARAM` parameter, the first # represents the group number. The second # represents the generated key given for the group. **Example:** `SQL.1.GK.1.TYPE=string` There is no default value for this parameter. The `SQL.#.GK.#.TYPE` is not required.
`MAX_STATEMENTS`	The `MAX_STATEMENTS` parameter specifies the maximum number of statements to search for within the initialization file. The `MAX_STATEMENTS` parameter can be used to increase the performance of the initialization file. If more than 100 SQL statements are defined, you must set the value of this parameter accordingly. **Example:** `MAX_STATEMENTS=100` The default value for this parameter is `100`. The `MAX_STATEMENTS` parameter is not required.
`MAX_PARAMETERS`	The `MAX_PARAMETERS` parameter specifies the maximum number of parameters to search for within each SQL statement. The `MAX_PARAMETERS` parameter can be used to improve the performance of the initialization file. If more than 100 parameters are defined, you must set the value of this parameter accordingly. **Example:** `MAX_PARAMETERS=100` The default value for this parameter is `100`. The `MAX_PARAMETERS` parameter is not required.
`SQL_CURSOR_TYPE_VALUE`	If you use cursors as a value for the output type parameters, then use the `SQL_CURSOR_TYPE_VALUE` parameter to specify the JDBC Type integer value for cursors. **Example:** `SQL_CURSOR_TYPE_VALUE=-10` The default value for this parameter is `-10`. The `SQL_CURSOR_TYPE_VALUE` parameter is not required. **Note:** Your database vendor distributes the value that you must specify for the `SQL_CURSOR_TYPE_VALUE` parameter. The default value of `-10` is a defaulted Oracle setting and does not need to be changed.

Optional SQL definition parameters for the JDBC Connector, continued

Parameter	Description
`ALLOW_GENERIC=true|false`	The `ALLOW_GENERIC` parameter specifies the field that determines if "generic" SQL statements are allowed by way of the key format. **Example:** `ALLOW_GENERIC=true` The default setting for this parameter is `true`. The `ALLOW_GENERIC` parameter is not required.

Key Parameters

The JDBC Connector parameters ensure that the appropriate database call information is passed to the connector so that you can successfully retrieve database results. Since you must pass a seek key string to the JDBC Connector, you have the option to add more flexibility to key behavior by configuring specific key parameters.

For more information about the seek key and which keys to pass in your initialization file, see

To configure key behavior, specify values for the key parameters.

Optional key parameters for the JDBC connector

Parameter	Description
`SQL_NULL=null|string value`	The `SQL_NULL` parameter specifies the string that represents the null value used in SQL statement inputs. The `SQL_NULL` parameter is not required. **Example:** `SQL_NULL=null` The default value for this parameter is `null`.
`KEY.DELIMITER=:::|delimiter character`	The `KEY.DELIMITER` parameter specifies the delimiter for the key passed to the JDBC Connector. The key is parsed using regular expression characters. For regular expression special characters, you must use escape characters. For example, you would specify `|` as `\\\|` in your initialization file. Due to the behavior Java regular expression engine, you must use `\\` when specifying special characters as Java automatically removes one `\` when parsing the key. **Example:** `KEY.DELIMITER=:::` The default value for this parameter is `:::`. The `KEY.DELIMITER` parameter is not required.

9.3 Testing the JDBC Connector

For best results, before using the JDBC Connector in production, you should test your configuration. Testing the configuration of the JDBC Connector helps to prevent errors that can occur during production runs.

To test the JDBC Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - All data source files to be used with the connector
   - Initialization file with specified parameters
2. Verify that you have assigned the connector object to all relevant data files in the application.
3. Verify that you have assigned the connector object to an output queue.
4. Package your application and run the JDBC Connector in standalone mode.

   For more information about testing the JDBC Connector configuration in standalone mode, see “Configuring the JDBC Connector to Run in Standalone Mode” on page 161.
5. Verify that your output is correct based on the configuration of the JDBC Connector.

Chapter 10: The Java Messaging Service Connector

The Java Messaging Service (JMS) Connector lets the engine communicate with JMS-compliant enterprise messaging software, such as Sun Java System MQ software and IBM WebSphere MQ software, using user-written routines. The JMS Connector is part of the open J2EE platform, and is useful with Web applications, particularly transaction processing. The JMS Connector has the following capabilities:

- Multiple correlation IDs (one input to multiple outputs)
- Sending messages to remote queues
- Synchronous and asynchronous messaging

For example, suppose your organization wants to give field representatives the opportunity to request a letter that introduces a customer to a new insurance policy upgrade. Insurance agents in field offices place a request for the letter on the IBM WebSphere Message Queue. When the message queue service receives the letter request, the JMS Connector, configured to communicate with IBM WebSphere Message Queue and the Exstream engine, sends the request to the engine for document generation. After the engine processes the data and produces the document, the JMS Connector passes a PDF of the letter back to the message queue. The representative at the field office can then print or email the letter to the customer.

This chapter discusses the following topics:

- “Preparing the JMS Connector” below
- “Configuring the JMS Connector” on page 189
- “Testing the JMS Connector” on page 213

10.1 Preparing the JMS Connector

The JMS Connector is available on multiple platforms. Depending on whether you are installing the JMS Connector on a mainframe or non-mainframe system, you must download the appropriate ZIP file from My Support.

- `JMS_Connector_[Version]_[Release].zip`—The ZIP file contains an executable Java Archive (JAR) file. After you extract the JAR file, you should keep the file in the application directory.
- `JMS_Connector_Auth_[Version]_[Release].zip`—The ZIP file contains authentication files and a series of Java documents to assist in the configuration of JMS Connector.

To prepare the JMS Connector for configuration, testing, and use in production, you must complete one of the following tasks:

- "Installing the JMS Connector on a Non-Mainframe Platform" below
- "Installing the JMS Connector on a Mainframe Platform" on the next page

10.1.1 Installing the JMS Connector on a Non-Mainframe Platform

To install the JMS Connector on a non-mainframe platform, you must complete the following tasks:

1. Extract the JAR file for the JMS Connector to a directory on your machine.

   The JAR file for the JMS Connector uses the following syntax:

   ```
   JMSConnector_[Version].jar
   ```

2. Configure the command environment for your platform:

   - **Windows**—If you are operating on the Windows platform, you must configure the `Path` environment variable to reference the `jvm.dll` directory.
   - **UNIX**—If you are operating on a UNIX platform, you must configure a command environment script file. The command environment file allows the operating system to locate all of the necessary files to run the JMS Connector and the engine. Configuring a command environment script sets up the Java Virtual Machine (JVM) on your UNIX platform.

3. If you operate the JMS Connector on an HP-UX platform, set up the following environment variables:

To	Do this
Reference the `WebSphere MQ lib` directory and `java/lib` directory	Set up the `SHLIB_PATH` environment variable.
Reference the HP-UX JDK version from Sun	Set up `JAVA_HOME` environment variable.
Reference the `libJavaEnabler.sl` file directory	Set up the `SHLIB_PATH` environment variable.

10.1.2 Installing the JMS Connector on a Mainframe Platform

To install the JMS Connector on a mainframe platform, you must complete the following tasks:

- "Copying and Expanding the JMS Archive" below
- "Copying and Renaming Exstream Sample Files" below
- "Setting Up the Environment File" on the next page

Copying and Expanding the JMS Archive

To copy and expand the JMS archive, you must complete the following steps:

1. Download the connector TAR file into the root directory of your local machine.
2. Open an FTP session to the UNIX services side of your z/OS system.
3. Create the following directory on the z/OS system: `/usr/local/exstream`.
4. Change the directory to `/usr/local/exstream`.
5. Use the `bin` command to change the mode to binary.
6. Enter `C:\JMS_Connector.tar` to transfer the TAR file.
7. Close the FTP session.
8. Open a telnet session.
9. Change the directory to `/usr/local/exstream/`.
10. Extract the file: `tar –xvf jms_connector_<version>.tar`.

Copying and Renaming Exstream Sample Files

To copy and rename the sample files, you must complete the following steps:

1. While still logged in to the previous telnet session that you opened in the previous task, change the directory to the `samples` directory (`/usr/local/exstream/samples`).
2. Enter `cp verjms "//'<job dataset name>'"`.

   For example, `cp verjms "//'"HLQ.JENABLER.JOB'"`.
3. Enter `cp loaddadv "//'<controldataset name>'"`.

   For example, `cp verjms "//'"HLQ.JENABLER.CONTROL'"`.
4. Enter `cp EXJC_P_1_1_1_.INI '//'<your queue name>'"`.

For example, `cp EXJC_P_1_1_1_.INI '//'MYQUEUE.INI'"`

5. Copy the following to the `/usr/local/exstream/` directory:
   - The job file
   - The new control file
   - `ENV`
   - `EXJC_P_1_1_1_.INI`

Note: When you enter the file names, make sure you enter the names in capital letters.

Setting Up the Environment File

To set up the environment file, you must complete the following steps:

1. Using a text editor, open `ENV` from `/usr/local/exstream`.
2. Configure the appropriate settings for the environment file.

10.2 Configuring the JMS Connector

The JMS Connector lets the engine communicate with the multiple message queue systems, such as IBM WebSphere Message Queue and Sun Java Message Queue, to generate on-demand documents. Documents are generated when a message is sent to the message queue and a request is sent to the engine for fulfillment. During fulfillment, the resulting output is then sent back to the message queue to fulfill the request. The JMS Connector is responsible for managing the flow of data to the engine and the delivery of the resulting output back to the message queue. In order to successfully manage incoming data and, subsequently, the delivery of the resulting output, you must configure the JMS Connector so that it can be identified by both the message queue for which it is configured and the engine and identify the data to be processed. Additionally, you must make sure that configuration includes a set of parameters that define the way in which the JMS Connector handles specific processes.

In addition to the basic configuration requirements for the JMS Connector, you can also use the JMS Connector to use multiple engines simultaneously (correlation mode), automatically reconnect, or enable security features. To use the JMS Connector with these additional configurations, you must assign a set of optional parameters in your initialization file.

This section discusses the following topics:

- “Configuring the JMS Connector in Design Manager” on the next page
- “Setting Up Your Initialization File” on page 191

- “JMS Connector Parameters” on page 192
- “Configuring Security Features for the JMS Connector” on page 202
- “Configuring QueueClientFactory Class Load Property Methods” on page 203
- “Configuring Client API Connection Classes for the JMS Connector” on page 204
- “Extending a Java Program as a Client Document Server for the JMS Connector” on page 210
- “Configuring the ExstreamResult Class of the ExstreamEngine Function” on page 212

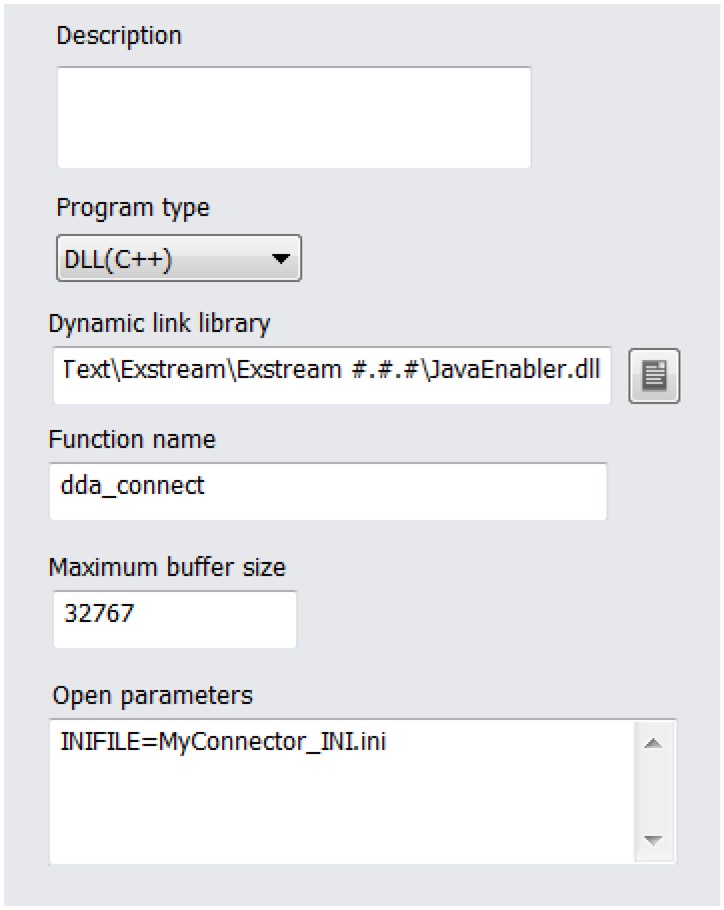
10.2.1 Configuring the JMS Connector in Design Manager

To configure the JMS Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name for the Java Enabler library for your environment, or click to go to the file: For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to 32767, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see “Setting Up Your Initialization File” on the next page.

Example of the completed connector object properties for the JMS Connector



Description

Program type: DLL(C++)

Dynamic link library: Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name: dda_connect

Maximum buffer size: 32767

Open parameters: INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Connectors in Design Manager" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning a connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to an output.

   For more information about assigning a connector to an output, see "Assigning a Connector Object to an Output" on page 27.

10.2.2 Setting Up Your Initialization File

After you configure the JMS Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager as opposed to listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

For more information about referencing a connector initialization file in Design Manager, see "Configuring the JMS Connector in Design Manager" on page 190

To set up your initialization file for the JMS connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For more information about parameters for the JMS Connector, see "JMS Connector Parameters" below

10.2.3 JMS Connector Parameters

After you set up your initialization file, you can begin adding control parameters to the initialization file. The JMS Connector contains a number of different parameters that specify the type of queue you are using in your infrastructure, the optional parameters that you can use with the specified queue, and other parameters that you can define to control the behavior of the connector.

This section discusses the following topics:

- "Message Queue Service Initialization Parameters" below
- "Setting Automated Reconnection Parameters" on page 200
- "Configuring Correlation Parameters for Multiple Outputs" on page 201

Message Queue Service Initialization Parameters

The JMS Connector is compatible with several messaging queue services. Each messaging queue service requires the same parameters, but values can vary depending on the messaging queue service that you configure for the JMS Connector. The JMS Connector is supported on the following messaging queue services:

- IBM WebSphere MQ
- Sun Java MQ
- Java Naming and Directory Interface Messaging Service (JNDI)

To configure a messaging queue service for the JMS Connector, specify values for the following parameters.

IBM WebSphere MQ Messaging Service initialization parameters

Parameter	Description

`CLASS`	The `CLASS` parameter specifies the name of the JMS Connector class used by the Java Enabler.
	You must set the `CLASS` parameter to the following path for all of the supported messaging queues:
	`com/exstream/connector/JMSConnector`
	Example:
	`CLASS=com/exstream/connector/JMSConnector`
	There is no default value for this parameter.

`CLASSPATH`	The `CLASSPATH` parameter specifies the system paths to the class libraries that the JMS Connector requires depending on the messaging queue.
	WebSphere MQ example:
	`CLASSPATH=<path>\jms.jar;<path>\jta.jar<path>\com.ibm.mqjms.jar;<path>\connector.jar;<path>\JMSConnector.jar`
	Sun Java MQ example:
	`CLASSPATH=<path>\imq.jar;<path>\jms.jar;<path>\JMSConnector.jar`
	Sun Java MQ example:
	`CLASSPATH=<path>\imq.jar;<path>\jms.jar;<path>\JMSConnector.jar`
	BEA WebLogic 9.0 (JNDI) example:
	`CLASSPATH=<path>\JMSConnector_4_5_1.jar;<path>weblogic.jar`
	BEA WebLogic 10.0 (JNDI) example:
	`CLASSPATH=<path>JMSConnector_4_6_0004.jar;<path>weblogic.jar;<path>wljmsclient.jar`
	There is no default value for this parameter.
	Separate each file by the correct CLASSPATH separator for the platform (a colon or semicolon).

IBM WebSphere MQ Messaging Service initialization parameters, continued

Parameter	Description
`SERVERTYPE`	The `SERVERTYPE` parameter specifies the server type for the messaging queue. Select from one of the following values depending on the message queue that you use: • WebSphere MQ— `WebSphere MQ` • Sun Java— `SunONE` • JNDI— `JNDI` **Example:** `SERVERTYPE=JNDI` There is no default value for this parameter.
`HOSTNAME`	The `HOSTNAME` parameter specifies the host name for the messaging queue. **Example:** `HOSTNAME=localhost` There is no default value for this parameter.
`CHANNEL`	The `CHANNEL` parameter specifies the channel to use with the messaging queue. **Example:** `CHANNEL=Z` There is no default value for this parameter.
`PORT`	The `PORT` parameter specifies the port where the message server waits for connections. Select from one of the following values depending on the message queue that you use: • WebSphere MQ— `1414` • Sun Java— `7676` • JNDI— `20080` **Example:** `PORT=1414` There is no default value for this parameter.
`USERNAME`	The `USERNAME` parameter specifies the user name to use when connecting to the message queue. **Example:** `USERNAME=admin` There is no default value for this parameter.
`PASSWORD`	The `PASSWORD` parameter specifies the password to use when connecting to the message queue. **Example:** `PASSWORD=admin` There is no default value for this parameter.

IBM WebSphere MQ Messaging Service initialization parameters, continued

Parameter	Description
`QUEUENAME`	The `QUEUENAME` parameter specifies the queue for the message queue to write and read messages for a specific application. **Example:** `QUEUENAME=OTEX_JMS/JMSConnector!OTEX_ONLQ` There is no default value for this parameter.
`TIMEOUT`	The `TIMEOUT` parameter specifies the amount of time (in seconds) to wait for a message to arrive in the queue on each request to the message queue. **Example:** `TIMEOUT=10` There is no default value for this parameter.
`TRANSPORT`	The `TRANSPORT` parameter specifies the transport type for the JMS Connector. Select from one of the following values: • `bindings` • `tcpip` **Example:** `TRANSPORT=tcpip` There is no default value for this parameter. You must set to `tcpip` for the real-time logging channel.
`MODE`	The `MODE` parameter specifies the action you want the connector and routine to take. For example, if you want the connector and routine to read the file in the message queue, select `R`. Select from one of the following values: • `R`—Read • `W`—Write • `RW`—Read and Write • `S`—Seek **Example:** `MODE=R` There is no default value for this parameter. Use `MODE=RW` only once for each engine instance.

IBM WebSphere MQ Messaging Service initialization parameters, continued

Parameter	Description
`TRACE`	The `TRACE=` parameter specifies whether the connector writes trace messages to a debug file. Select from one of the following options: • `Y` • `N` **Example:** `TRACE=Y` The default value for this parameter is `N`.
`INSTANCE`	The `INSTANCE` parameter specifies whether a separate connector is created for each connector mapping. Select from one of the following values: • `UNIQUE` • `SHARED` **Example:** `INSTANCE=UNIQUE` The default value for this parameter is `SHARED`. If you set the value of the `MODE` parameter to either `R` or `W`, then you must set the value of the `INSTANCE` parameter to `UNIQUE`. If you set the value of the `MODE` parameter to `RW`, then you must set the value of the `INSTANCE` parameter to `SHARED`.
`QUEUEMANAGER`	The `QUEUEMANAGER` parameter specifies the name of the queue manager. The `QUEUEMANAGER` parameter is required only on WebSphere MQ. **Example:** `QUEUEMANAGER=webverse.queue.manager` There is no default value for this parameter.
`MODELQUEUE`	The `MODELQUEUE` parameter specifies the model queue to use when creating temporary queues. The `MODELQUEUE` parameter is requierd only on WebSphere MQ. **Example:** `MODELQUEUE=SYSTEM.DEFAULT.MODEL.QUEUE` There is no default value for this parameter.
`QUEUEFACTORY`	The `QUEUEFACTORY` parameter specifies the lookup name for the initial connection factory on the JMS server. The `QUEUEFACTORY` parameter is requierd only on JNDI. **Example:** `QUEUEFACTORY=jms/OTEX_ONLQ_DEVCF` There is no default value for this parameter.

IBM WebSphere MQ Messaging Service initialization parameters, continued

Parameter	Description
`CONTEXTFACTORY`	The `CONTEXTFACTORY` parameter specifies the context factory class name. The `CONTEXTFACTORY` parameter is required only on JNDI. **Example:** `CONTEXTFACTORY=weblogic.jndi.WLInitialContextFactory` There is no default value for this parameter.
`PROVIDERURL`	The `PROVIDERURL` parameter specifies the JMS broker URL. The `PROVIDERURL` parameter is requierd only on JNDI. **Example:** `PROVIDERURL=t3://localhost:20080` There is no default value for this parameter.
`PRINCIPAL`	The `PRINCIPAL` parameter specifies the User ID required to connect to a JMS provider. The `PRINCIPAL` parameter is requierd only on JNDI. **Example:** `PRINCIPAL=weblogic` There is no default value for this parameter.
`CREDENTIALS`	The `CREDENTIALS` parameter specifies the password required to connect to a JMS provider. The `CREDENTIALS` parameter is required only on JNDI. **Example:** `CREDENTIALS=weblogic` There is no default value for this parameter.

IBM WebSphere MQ Messaging Service initialization parameters, continued

Parameter	Description
`EOM`	The `EOM` parameter specifies the indicator used by the reference file server to signal when the last message is sent for a single seek request. Select from one the following values: • `Single`—The `Single` indicates only one reply-to message per seek request is expected. Subsequent replies are ignored. When a reference file server sends multiple reply-to messages per seek request, an end-of-message indicator is required. • `Header`—The `Header` value indicates that a header-only message marks the end-of-message condition. The header-only message is the last message sent in the response to a seek request; therefore, it must be sent even if only one reply-to messages is sent. A header-only message is an instance of the JMS class message. • `Property Name`—The `Property Name` value is a user supplied name and indicates that a message header property marks end-of-message. The header property name is the value of property name. The header property is required only for the last message and optional on preceding messages. You must make sure that the value of the property does not indicate end-of-message before the last message is sent. When `EOM` is set to property name, the `EOMValue` parameter must be defined with the value notifying the end-of-message indication. The `EOMValue` value is not case-sensitive. An exception is thrown if the `EOMValue` is not defined. If `EOM` is not set to the property name, `EOMValue` is ignored. The `EOM` parameter is not required and defaults to single. **Example:** `EOM=Header` There is no default value for this parameter.
`EOMValue`	The `EOMValue` parameter specifies the property value indicating end-of-message. The `EOMValue` is required only when the value for the `EOM` parameter is set to `Property Name`. **Example:** `EOMValue=property value` There is no default value for this parameter.

Example Initialization File for IBM WebSphere MQ Messaging Service

The following example shows an initialization file for the JMS Connector on IBM WebSphere MQ:

```
CLASS=com/exstream/connector/JMSConnector
CLASSPATH=<path>\jms.jar;<path>\jta.jar<path>\com.ibm.mqjms.jar;
   <path>\connector.jar;<path>\JMSConnector.jar
TRACE=Y
SERVERTYPE=WebSphereMQ
MODE=RW
HOSTNAME=localhost
PORT=1414
USERNAME=
PASSWORD=
QUEUENAME=P1_1_1
MODELQUEUE=SYSTEM.DEFAULT.MODEL.QUEUE
TIMEOUT=10
QUEUEMANAGER=webverse.queue.manager
CHANNEL=Z
TRANSPORT=BINDINGS
```

Example Initialization File for Sun Java MQ Messaging Service

The following example shows an initialization file for the JMS Connector on Sun Java:

```
CLASS=com/exstream/connector/JMSConnector
CLASSPATH=<path>\imq.jar;<path>\jms.jar;<path>\JMSConnector.jar
TRACE=Y
SERVERTYPE=SunONE
MODE=RW
USERNAME=admin
PASSWORD=admin
HOSTNAME=localhost
PORT=7676
QUEUENAME=P7_2_1
TIMEOUT=10
```

Example Initialization File for JNDI Messaging Service

The following example shows an initialization file for the JMS Connector on JNDI:

```
CLASS=com/exstream/connector/JMSConnector
CLASSPATH=/opt/exstream/Tar/wllib/JMSConnector_4_6_0004.jar:
   /opt/bea/wlserver_10.3/server/lib/weblogic.jar:
   /opt/bea/wlserver_10.3/server/lib/wljmsclient.jar:
   /opt/bea/jdk160_05/jre/bin/client
SERVERTYPE=JNDI
HOSTNAME=localhost
PORT=20080
USERNAME=weblogic
PASSWORD=weblogic
QUEUENAME=OTEX_JMS/JMSConnector!OTEX_ONLQ
TIMEOUT=1
MODE=RW
QUEUEFACTORY=jms/OTEX_ONLQ_DEVCF
CONTEXTFACTORY=weblogic.jndi.WLInitialContextFactory
PROVIDERURL=t3://localhost:20080
AUTHENTICATE=N
PRINCIPAL=weblogic
CREDENTIALS=weblogic
TRACE=Y
INSTANCE=SHARED
```

Setting Automated Reconnection Parameters

The JMS Connector lets you specify parameters that control how the JMS Connector behaves in the event of a loss of network connectivity. To set automated reconnection parameters, you must specify a value for all of the following parameters.

Parameters for automatic reconnection

Parameters	Description
`MaxNumberFailures`	The `MaxNumberFailures` parameter specifies the number of times the JMS Connector attempts to connect to the server after a connection is lost. **Example:** `MaxNumberFailures=0` The default value for this parameter is `0`.
`MaxNumberRetries`	The `MaxNumberRetries` parameter specifies the number of times the JMS Connector attempts to complete an operation when a connection is lost. **Example:** `MaxNumberRetries=0` The default value for this parameter is `0`.

Parameters for automatic reconnection, continued

Parameters	Description
`RetryWaitSecs`	The `RetryWaitSecs` parameter specifies the amount of time (in seconds) JMS Connector waits before attempting to reconnect. **Example:** `RetryWaitSecs=0` The default value for this parameter is `0`.

Configuring Correlation Parameters for Multiple Outputs

After you have included the parameters for your message queue service and specified reconnection parameters, you can set up the JMS Connector to correlate one input to multiple outputs. In addition to including the required parameter in your initialization file, you must also associate the initialization file to for each object in your application.

Before you can set up correlation, you must include the following parameter in your initialization file.

Parameter for multiple output support

Parameter	Description
`CorrelationClass`	This parameter specifies a string that lets JMS Connector and the engine determine objects that must be correlated to multiple outputs. You determine what you want the string to be. The string must be the same for each object you want to correlate. **Example:** `CorrelationClass=[string]` There is no default value for this parameter.

To set up correlation classes for multiple outputs, you must complete the following steps:

1. Set up the initialization file for each object needed for your engine run (customer driver file, output, report file, log file, and so on).
2. Specify the same string in the `CorrelationClass` parameter in the initialization file for each object you want to correlate.
3. Run the engine.

The JMS Connector monitors the correlation information for each input file by way of the `CorrelationClass` parameter. The engine sets the correlation information to the output of each object that contains the same correlation class string.

10.2.4 Configuring Security Features for the JMS Connector

The JMS Connector offers additional features for security purposes.

The following security features are available with the JMS Connector:

- **Authentication Notes**—The authentication feature verifies the identity of the request before proceeding with a job.
- **Encryption Notes**—The encryption feature secures the initialization file so it cannot be modified.

The JMS installer ZIP file contains a series of documents that explain these security features.

Authentication Notes for JMS Connector Security

You can set up an authentication system within your JMS solution by using methods within the `cas.jar` authentication archive.

JMS Connector authentication is a framework that lets an installation require that each message placed on the queue be accompanied by a security token. After placing the security token with the message, the framework intercepts the message and redirects it to a special authorization queue. This queue makes the message available to the installation service that performs the authorization.

Each installation must provide a token generation, an authorization service, and an agent for sending the tokens to the authorization service. Agent and service modules can be separate or combined.

The framework provides two interfaces:

- A class used by the client for submitting a real-time request with a security token
- A class used by an installation's authorization agent to receive security tokens and determine whether the authorization request succeeds or fails

The factory class `QueueClientFactory` is used to create either interface. The factory class is first initialized with the initialization file.

For submitting messages, the `QueueClientFactory` has two proxy interfaces:

- `ServiceRequestor`—The `ServiceRequestor` submits data file requests.
- `CommandRequest`—The `CommandRequest` submits engine commands.

For information about how to use the `ServiceRequestor` and `CommandRequest` to submit messages, refer to the documents located in the JMS installer ZIP file.

Additionally, the `QueueClientFactory` creates the Authentication Listener for monitoring the authorization queue. The listener makes messages and tokens available to an installation security agent. The security agent creates a listener and makes a call to get the message. If no message exists, a timeout starts and the agent can recall to a message. This loop can continue until shutdown.

10.2.5 Configuring QueueClientFactory Class Load Property Methods

The `QueueClientFactory` class creates the queue connection factory that lets the JMS Connector communicate with configured messaging queues. You must configure the `QueueClientFactory` class load properties file to pass customer information to the authentication queue for validation.

The `loadproperties` method takes a string or a file path to an initialization file that contains the information needed to set up a connection factory.

For example:

```
factory.loadProperties(new String("jmsconfigini.ini"));
```

To configure the `QueueClientFactory` class methods, you must complete the following tasks:

To	Do this
Create a new instance of a document service requestor	Enter the following in your string or initialization file path for the connection factory: `public ServiceRequestor createServiceRequestor()throws JMSServiceException` Creating the `ServiceRequestor` method lets you submit document generation requests to the on-demand engine. Clients that must have each document request authenticated use this instance of the `ServiceRequestor` method. Clients must provide an authentication token and the driver file for the engine. This method returns an instance of `ServiceRequestor` and creates an exception object called `JMSServiceException`. This method does not require any parameters.
Create a new instance of an authentication service requestor	Enter the following in your string or initialization file path for the connection factory: `public AuthenticationListenercreateAuthenticationRequestor ()throws JMSServiceException` The `AuthenticationRequestor` method receives requests from an authentication queue. Each authentication response returns to the authentication queue. The `AuthenticationRequestor` method returns a single instance of the engine and creates an exception object called `JMSServiceException`. This method does not require any parameters.

To	Do this
Create a new instance of a connector command requestor	Enter the following in your string or initialization file path for the connection factory: `public CommandRequest createACommandRequestor()throws JMSServiceException` The `CommandRequest` method submits a connector command to the real-time engine. Clients that must have each command authenticated use an instance of the `CommandRequest` method. Clients must provide an authentication token when using the `CommandRequest` method. The `CommandRequest` method returns an instance of the engine and creates an exception object called `JMSServiceException`. This method does not require any parameters.

10.2.6 Configuring Client API Connection Classes for the JMS Connector

After you have configured security features and the QueueClientFactory properties for the JMS Connector, you must configure client API connection classes to use with the JMS Connector and the messaging queue that you specified in the JMS Connector initialization parameters.

To configure client API connection classes for the JMS Connector, you must configure the properties and functions of the following connection classes in an initialization file:

- **Factory Builder Class**—The Factory Builder Class establishes a connection with the JMS Server
- **Preview Engine Class**—The Preview Engine Class sets the connection limits for the JMS Preview Engine
- **Preview Engine Connection Class**—The Preview Engine Connection Class sets the JMS server expiration time limits

To configure the client API connection classes for the JMS Connector, you must complete the following tasks depending on the messaging service that you use:

- “Configuring Factory Builder Classes With WebSphere MQ Messaging Service” on the next page
- “Configuring Factory Builder Classes for Sun Java System MQ Messaging Service” on page 206
- “Configuring Factory Builder Classes with JNDI Messaging Services” on page 207
- “Setting JMS Connector Connection Limits for the Preview Engine Class” on page 208
- “Setting JMS Connector Expiration Time Limits for the Preview Engine Connection Class” on page 209

Configuring Factory Builder Classes With WebSphere MQ Messaging Service

IBM Websphere MQ messaging service uses specific properties and functions to create a connection class that can be used with the JMS Connector. You must use the following syntax to configure these settings.

```
com.exstream.jms[propertysetting].[function]
```

If you are using the IBM WebSphere MQ Messaging Service with the JMS Connector, you must configure the following properties and functions.

Property settings and functions for JMS on WebSphere MQ Messaging Service

Property settings and functions	Description
`QueueConnectionFactoryBuilder.hostname`	The `QueueConnectionFactoryBuilder.hostname` property setting and function specifies the host name of the WebSphere MQ servers.
`QueueConnectionFactory.port`	The `QueueConnectionFactory.port` property setting and function specifies the port numbers for WebSphere MQ. For WebSphere MQ, the port number is usually `1414`.
`QueueConnectionFactory.username`	The `QueueConnectionFactory.username` property setting and function specifies the user name for connecting to the WebSphere MQ server.
`QueueConnectionFactory.password`	The `QueueConnectionFactory.password` property setting and function specifies the password for the user name.
`QueueConnectionFactory.queueContext`	The `QueueConnectionFactory.queueContext` property setting and function specifies the prefix expected for queue names. This property must always be empty for use with WebSphere MQ.
`QueueConnectionFactoryBuilder=` `com.exstream.jms.`	The `QueueConnectionFactoryBuilder` property setting and function specifies the name of the `Queue Connection Factory Builder` class. This name you use is message server-specific. For WebSphere MQ, enter `WebSphereMQQueueConnectionFactoryBuilder`.
`QueueConnectionFactoryBuilder.` `queueManager`	The `QueueConnectionFactoryBuilder.queueManager` property setting and function specifies the queue manager you use with the WebSphere MQ server.
`QueueConnectionFactoryBuilder.channel`	The `QueueConnectionFactoryBuilder.channel` property setting and function specifies the channel you use with the WebSphere MQ server.
`QueueConnectionFactoryBuilder.` `temporaryQueueModel`	The `QueueConnectionFactoryBuilder.temporaryQueueModel` property setting and function specifies the model you use when you create temporary queues for use with the WebSphere MQ server.

The following example shows the configured factory builder classes in the client initialization file for WebSphere MQ messaging services:

```
com.exstream.jms.QueueConnectionFactoryBuilder.hostname=localhost
com.exstream.jms.QueueConnectionFactoryBuilder.port=1414
com.exstream.jms.QueueConnectionFactoryBuilder.username=
com.exstream.jms.QueueConnectionFactoryBuilder.password=
com.exstream.jms.QueueConnectionFactoryBuilder.queueContext=
com.exstream.jms.QueueConnectionFactoryBuilder=
   com.exstream.jms.WebSphereMQQueueConnectionFactoryBuilder
com.exstream.jms.QueueConnectionFactoryBuilder.queueManager=realtime.demo
com.exstream.jms.QueueConnectionFactoryBuilder.channel=Z
com.exstream.jms.QueueConnectionFactoryBuilder.temporaryQueueModel=
   SYSTEM.DEFAULT.MODEL.QUEUE
```

Configuring Factory Builder Classes for Sun Java System MQ Messaging Service

Sun Java MQ messaging service uses specific properties and functions to create a connection class that can be used with the JMS Connector. You must use the following syntax to configure these settings.

`com.exstream.jms[propertysetting].[function]`

If you are using the Sun Java MQ Messaging Service with the JMS Connector, you must configure the following properties and functions.

Property settings and functions for JMS on Sun Java System MQ Messaging Service

Property settings and functions	Use
`QueueConnectionFactoryBuilder.hostname`	The property `QueueConnectionFactoryBuilder.hostname` setting and function specifies the host name of the Sun Java MQ server.
`QueueConnectionFactory.port`	The `QueueConnectionFactory.port` property setting and function specifies the port number for Sun Java MQ. For Sun Java MQ, the port number is usually `7676`.
`QueueConnectionFactory.username`	The `QueueConnectionFactory.username` property setting and function specifies the user name for connecting to the Sun Java MQ server.
`QueueConnectionFactory.password`	The `QueueConnectionFactory.password` property setting and function specifies the password for the user name.
`QueueConnectionFactory.queueContext`	The `QueueConnectionFactory.queueContext` property setting and function specifies the prefix expected for queue names. This property must always be empty for use with Sun Java MQ.

Property settings and functions for JMS on Sun Java System MQ Messaging Service, continued

Property settings and functions	Use
`QueueConnectionFactoryBuilder=com.exstream.jms`	The `QueueConnectionFactoryBuilder` property setting and function specifies the name of the `Queue Connection Factory Builder` class. The class name that you use is message server-specific. For Sun Java MQ, enter `SunONEQueueConnectionFactoryBuilder`.

The following example shows the configured factory builder classes in the client initialization file for Sun Java messaging services:

```
com.exstream.jms.QueueConnectionFactoryBuilder.hostname=localhost
com.exstream.jms.QueueConnectionFactoryBuilder.port=7676
com.exstream.jms.QueueConnectionFactoryBuilder.username=admin
com.exstream.jms.QueueConnectionFactoryBuilder.password=admin
com.exstream.jms.QueueConnectionFactoryBuilder.queueContext=
com.exstream.jms.QueueConnectionFactoryBuilder=
   com.exstream.jms.SunONEQueueConnectionFactoryBuilder
```

Configuring Factory Builder Classes with JNDI Messaging Services

The JNDI messaging service uses specific properties and functions to create a connection class that can be used with the JMS Connector. If you are using the JNDI Messaging Service with the JMS Connector, you must configure the following properties and functions.

Property settings and functions for JNDI messaging services

Property settings and functions	Use
`java.naming.security.principal`	The `java.naming.security.principal` property setting and function specifies the name of the user to authenticate when you establish a connection to the JMS server. The JNDI principal must be set to a user who can only perform the JNDI lookup, and has no privileges for any JMS destinations.
`java.naming.security.credential`	The `java.naming.security.credential` property setting and function specifies the credentials set for the user who can only perform the JNDI lookup. The user must have no privileges for any JMS destinations. You must enter the credentials, such as a password, that authenticates the security principal to the JMS provider.
`java.naming.factory.initial`	The `java.naming.factory.initial` property setting and function specifies the `javax.naming.spi. InitialContextFactory`, which your client uses to obtain initial naming context.

Property settings and functions for JNDI messaging services, continued

Property settings and functions	Use
`java.naming.provider.url`	The `java.naming.provider.url` property setting and function specifies the host name and port of the DNS server. Enter the provider URL in `<host>[:<port>]` format.
`com.exstream.jms.QueueConnectionFactoryBuilder`	The `com.exstream.jms.QueueConnectionFactoryBuilder` property setting and function specifies the name of the message server-specific `QueueConnectionFactory` builder class to use.
`com.exstream.jms. QueueConnectionFactoryBuilder. QueueConnectionFactory`	The `com.exstream.jms.QueueConnectionFactoryBuilder. QueueConnectionFactory` property setting and function specifies the JNDI for the registered `QueueConnectionFactory`.

The following example shows the configured factory builder classes in the client initialization file for JNDI messaging services:

```
com.exstream.jms.QueueConnectionFactoryBuilder.hostname=md-doej
com.exstream.jms.QueueConnectionFactoryBuilder.port=7001
com.exstream.jms.QueueConnectionFactory.username=weblogic
com.exstream.jms.QueueConnectionFactory.password=weblogic
com.exstream.jms.QueueConnectionFactoryBuilder=com.exstream.jms.
   JNDIQueueConnectionFactoryBuilder
com.exstream.jms.QueueConnectionFactory.queueContext=weblogic.jndi.
   WLInitialContextFactory
com.exstream.jms.QueueConnectionFactoryBuilder.queueConnectionFactory=
   javax.jms.QueueConnectionFactory
```

Setting JMS Connector Connection Limits for the Preview Engine Class

The optimum number of JMS connections can vary from system to system due to many factors, including third-party equipment, facilities/resources, average document size, and user expectations. The goal is to provide sufficient JMS connections to support the wait-time expectations of users without over-allocating resources so other applications and services do not compete for available resources.

You can configure the client API of the JMS Connector to set connection limits for the preview engine to provide sufficient JMS connections. To begin the process of determining the appropriate connection limits, you can configure the following settings:

- Small system: minimum `5`, maximum `15`.
- Large system: minimum `10`, maximum `30`.

After you begin with the recommended settings, you can then perform load testing to determine whether these levels are acceptable to both system resources and user expectations. If you need to adjust the settings, the preview engine class lets you set the connection limits for the JMS Connector preview engine by configuring the following property settings and functions:

When you configure the property settings and functions, you must use the following syntax:

```
com.exstream.engine.preview.PreviewEngine.[property setting].[function]
```

To configure the connection limits for the preview engine, you must configure the following properties and functions.

Property settings and functions for JMS preview engine class

Property settings and functions	Use
`com.exstream.engine.preview.PreviewEngine`	The `com.exstream.engine.preview.PreviewEngine` property setting and function lets the system load the class for the preview engine.
`minimumConnections`	The `minimumConnections` property setting and function sets the minimum number of JMS connections the connection pool maintains for the client. The system maintains no fewer than the number of connections you define.
`maximumConnections`	The `maximumConnections` property setting and function sets the maximum number of JMS connections the connection pool maintains for the client. When all connections defined in the `minimumConnections` function are in use, the Preview Engine Class creates temporary connections based on the number you define in the `maximumConnections` function. When the temporary connections become idle, they are disconnected and no longer used.

The following example shows the configured JMS preview engine class for WebSphere MQ messaging service:

```
com.exstream.engine.preview.PreviewEngine=com.exstream.engine.preview.
   PreviewEngine
com.exstream.engine.preview.PreviewEngine.minimumConnections=1
com.exstream.engine.preview.PreviewEngine.maximumConnections=5
```

Setting JMS Connector Expiration Time Limits for the Preview Engine Connection Class

The preview engine connection class lets you set JMS expiration time limits. These time limits let you control how long the JMS server holds reply messages or how long the Preview Engine waits for a reply.

When you configure the property settings and functions, you must use the following syntax:

```
com.exstream.jms[propertysetting].[function]
```

To configure the expiration time limits for the preview engine connection class, you must configure the following properties and functions.

Property settings and functions for JMS expiration time limits

Property settings and functions	Use
`PreviewEngineConnection.replyMessageTTL`	The `PreviewEngineConnection.replyMessageTTL` property setting and function specifies the time, in milliseconds, the `replyMessage` stays on the JMS server. After time expires, the `replyMessage` is no longer read. The recommended default setting is `25000`.
`PreviewEngineConnection.receiveTimeout`	The `PreviewEngineConnection.receiveTimeout` property setting and function specifies the time, in milliseconds, the Preview Engine waits for a reply. The recommended default setting is `30000`.

The following example shows the configured JMS Connector expiration time limits in the preview engine class for WebSphere MQ messaging service:

```
com.exstream.engine.preview.PreviewEngineConnection.replyMessageTTL=25000
com.exstream.engine.preview.PreviewEngineConnection.receiveTimeout=1000000
```

10.2.7 Extending a Java Program as a Client Document Server for the JMS Connector

To extend a Java program to act as a client of the document server, you can implement an instance of the ExstreamEngine class. This class serves as a wrapper for all document generation and takes the customer data as input and returns the document as output.

To create a client document server for the JMS Connector using instances of the ExstreamEngine class, you must complete the following tasks:

To	Do this
Create one instance of the engine	Enter the following method in your Java program: `public static ExstreamEngine getInstance()` The `public static ExstreamEngine getInstance()` method returns a single instance of the engine and creates an exception object called `EngineException`. This method does not require any parameters.
Initialize ExstreamEngine object properties	Enter the following method in your Java program: `public void initialize(Properties properties)` The `public void initialize(Properties properties)` method has no return value and creates an exception object called `Exception`. This method requires the `Properties` parameter.

To	Do this
Generate a document based on customer data by placing a request on the named queue	Enter the following method in your Java program: `public byte[] generate(String queue, byte[] bytes)` The `public byte[] generate(String queue, byte[] bytes)` method returns output as an array of bytes and creates an exception object called `EngineException`. This method requires the following parameters: • `Queue`—The name of the current engine queue • `Bytes`—The customer data for the engine
Generate a document based on customer data by placing a request on the named queue, but not return the generated document	Enter the following method in your Java program: `public void place(String queue, byte[] bytes)` The `public void place(String queue, byte[] bytes)` method does not return the generated document and creates an exception object called `EngineException`. This method requires the following parameters: • `Queue`—The name of the current engine queue • `Bytes`—The customer data for the engine
Generate a document based on customer data by placing a request on the named queue, but return a class that provides the header information	Enter the following method in your Java program: `public ExstreamResult generateResult(String queue, byte[] bytes)` The `public ExstreamResult generateResult(String queue, byte[] bytes)` method returns an instance of the `ExstreamResult` class and creates an exception object called `EngineException`. This method requires the following parameters: • `Queue`—The name of the current engine queue • `Bytes`—The customer data for the engine
Close the JMS session with the Preview Engine and close the messaging resources used by the client	Enter the following method in your Java program: `public void close( )` The `public void close( )` method has no return and creates an exception object called `EngineException`. This method does not require any parameters.

To	Do this
Send a JMS TextMessage containing a `SHUTDOWN` command to the indicated queue	Enter the following method in your Java program: `public void shutdown(String queueName)` The `public void shutdown(String queueName)` method instructs the engine to proceed normally until the queue has been emptied. The engine broadcasts a `SHUTDOWN` message to other engines working from the same queue. This method has no return and creates an exception object called `EngineException`. This method requires the `Queue` parameter only. The `Queue` parameter specifies the name of the current engine queue.
Send a JMS TextMessage containing a `STOP` command to the indicated queue	Enter the following method in your Java program: `public void stop(String queueName)` The `public void stop(String queueName)` method instructs the engine to stop execution immediately. The engine broadcasts a `STOP` message to stop execution of other engines working from the same queue. This method has no return and creates an exception object called `EngineException`. This method requires the `Queue` parameter only. The `Queue` parameter specifies the name of the current engine queue.

Caution: The command messages generated by the call persist up to 30 seconds. If you command a single engine to `SHUTDOWN`, you must wait at least 30 seconds after it ceases execution before attempting to start a new engine. The same is true for the `STOP` engine command. If you tell a single engine to `STOP`, you must wait at least 30 seconds after it ceases execution before attempting to start a new engine.

10.2.8 Configuring the ExstreamResult Class of the ExstreamEngine Function

You can configure the JMS Connector client API to generate results using the `generateResult` method of the the ExstreamEngine function. The `generateResult` method gives information about the document also returned with that method call.

To configure the ExstreamResult class of the ExstreamEngine function, you must complete the following tasks:

To	Do this
Return the engine code	Enter the following command in your Java program: `public int getReturnCode( )` The `public int getReturnCode( )` command requires no parameters.
Return the number of pages in a document	Enter the following command in your Java program: `public int getPageCount( )` The `public int getPageCount( )` command requires no parameters.
Return the engine version number	Enter the following command in your Java program: `public int getVersion( )` The `public int getVersion( )` command requires no parameters.
Return the code indicating the output format	Enter the following command in your Java program: `public int getPDL( )` The `public int getPDL( )` command requires no parameters.
Return the document itself	Enter the following command in your Java program: `public byte[ ] getContent( )` The `public byte[ ] getContent( )` command requires no parameters.

10.3 Testing the JMS Connector

For best results, before using the JMS Connector in production, you should test your configuration. Testing the configuration of the JMS Connector helps to prevent errors that can occur during production runs.

To test the JMS Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - All data source files to be used with the connector
   - The connector is assigned to all relevant data files in the application
   - The connector is assigned to an output in Design Manager
   - Initialization file with specified parameters

2. Verify that you have assigned the connector object to all relevant data files in your application.

3. Verify that you have assigned the connector object to an output.
4. Make sure that the JMS Connector is connected to the appropriate messaging queue software.
5. Package your application and run the engine with the ONDEMAND switch in your control file.
6. Verify that your output is correct based on the configuration of the JMS Connector.

Chapter 11: The Microsoft MQ Connector

The Microsoft MQ (MSMQ) Connector lets the engine communicate with the Microsoft MQ messaging service using user-written applications.

For example, suppose you work in the travel industry and want to increase your customer base by providing easy access for customers to plan vacations through your company. Your organization places documents for itineraries, package tours, and basic customer information on the Microsoft Message Queue. As customers request information from your organization, the MSMQ Connector passes the request to the Exstream engine where the request is processed and the appropriate document is generated. The MSMQ Connector then places the document back on the message queue. The document is then sent to the customer by way of the specified channel, such as a PDF.

This chapter discusses the following topics:

- “Preparing the MSMQ Connector” below
- “Configuring the MSMQ Connector” on the next page
- “Testing the MSMQ Connector” on page 225

11.1 Preparing the MSMQ Connector

To prepare the MSMQ Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the `MSMQ_Connector_[Version]_[Release].zip` file.
2. Extract the contents of the `MSMQ_Connector_[Version]_[Release].zip` file to the directory that contains the Exstream engine.
3. Verify that the following files are available with the Exstream engine:
   - `CLRConnector.dll`—The dynamic linked library (DLL) file that allows access to the connector
   - `CLREnabler.dll`—The DLL file that contains the CLR Enabler

   The CLR Enabler and CLR Connector DLLs are packaged and distributed with the Exstream engine. To run the MSMQ Connector, you must have the CLR Enabler and CLR Connector DLLs.

4. Register the `CLRConnector.dll` and `MSMQConnector.dll` files with the .NET framework.

   For example:

   ```
   C:\Windows\Microsoft.NET\Framework\v1.1.4322\RegAsm.exe CLRConnector.dll
   ```

   ```
   C:\Windows\Microsoft.NET\Framework\v1.1.4322\RegAsm.exe MSMQConnector.dll
   ```

5. For best results, make sure that any applications with test files package and run in batch mode with no errors.

11.2 Configuring the MSMQ Connector

The MSMQ Connector lets the engine communicate with the Microsoft Message Queue system to generate documents. Documents are generated when a message is sent to the message queue and a request is sent to the engine for fulfillment. Output is then sent to the message queue to fulfill the request.

This section discusses the following topics:

- “Configuring the MSMQ Connector in Design Manager” on the next page
- “Setting Up Your Initialization File” on page 218
- “MSMQ Connector Parameters” on page 220
- “Client Methods for the MSMQ Connector” on page 222
- “Preview Classes for the MSMQ Connector” on page 224

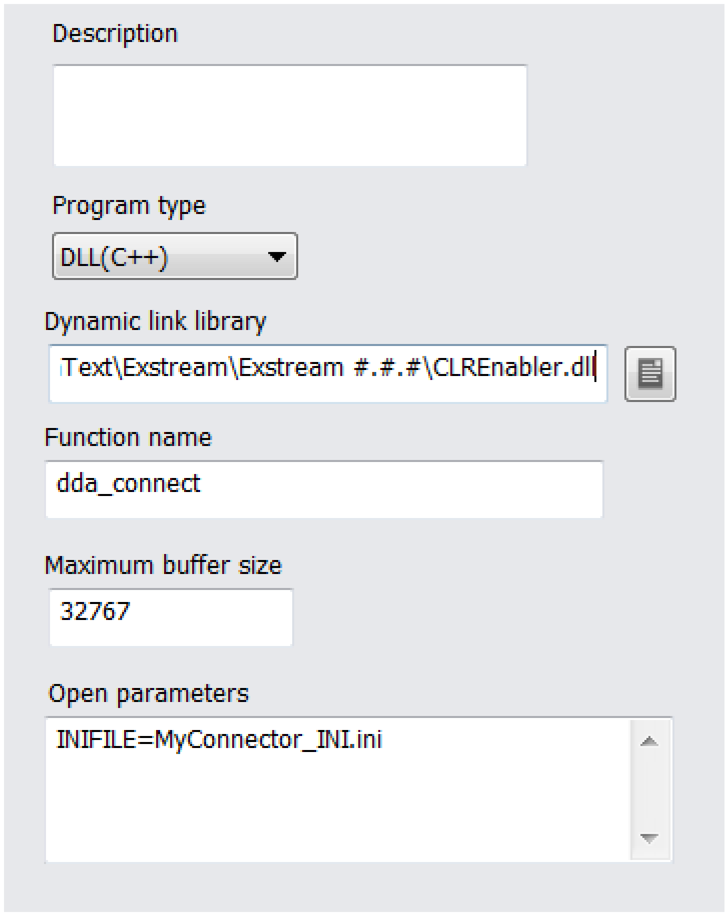
11.2.1 Configuring the MSMQ Connector in Design Manager

To configure the MSMQ Connector, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	Enter the path and file name to the CLR Enabler. For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\CLREnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to 32767, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on the next page.

Example of the completed connector object properties for the MSMQ Connector



Description

Program type
DLL(C++)

Dynamic link library
Text\Exstream\Exstream #.#.#\CLREnabler.dll

Function name
dda_connect

Maximum buffer size
32767

Open parameters
INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Dynamic Data Access" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning a connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to an output queue.

   For more information about assigning a connector to an output queue, see "Assigning a Connector Object to an Output Queue" on page 28.

11.2.2 Setting Up Your Initialization File

After you configure the MSMQ Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager instead of listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

To create an initialization file for the MSMQ connector parameters, you must complete the following steps:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For information about MSMQ parameters, see “MSMQ Connector Parameters” on the next page.

Specifying the Type of Initialization File to Use

Some of the parameters available for a connector support the use of double-byte characters. The use of double-byte characters in a connector parameter gives the user more flexibility in defining connector behavior. If the connector that you configure allows for parameters that support the use double-byte characters, you must first specify the type of initialization file that you will be using to define the parameters. For example, if your initialization file for the MSMQ Connector lists the MSMQ parameter with a value that uses DBCS characters, you must first specify a Unicode initialization file by defining the UNICODEINIFILE parameter.

To configure the parameters for the MSMQ Connector, you must specify a value for on the following parameters.

Parameters for specifying an initialization file

Parameter	Description
`INIFILE`	The `INIFILE` parameter specifies the fully-qualified file name for the initialization file. When specifying connector behavior with an initialization file, the `INIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. **Example:** `INIFILE=MyINI.ini` There is no default value for this parameter.

Parameters for specifying an initialization file, continued

Parameter	Description
`UNICODEINIFILE`	The `UNICODEINIFILE` parameter specifies an initialization file that contains one or more supported parameters with Unicode characters. When specifying connector behavior with an initialization file that contains Unicode characters in parameters, the `UNICODEINIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. The `UNICODEINIFILE` parameter is necessary for the engine to interpret Unicode characters correctly when the engine does not detect a byte order marker. **Example:** `MyUnicodeINI.ini` There is no default value for this parameter.

11.2.3 MSMQ Connector Parameters

The following information details the parameters you can use with the MSMQ Connector. In addition to operational parameters, you can also use methods that let you construct client applications to send and request messages from the Microsoft Message Queue system.

This section discusses the following topics:

- "Required Initialization Parameters for the MSMQ Connector" below
- "Optional Parameters for the MSMQ Connector" on the next page

Required Initialization Parameters for the MSMQ Connector

To configure MSMQ Connector, you must include at least one of the following parameters.

Required parameters for the MSMQ Connector

Parameter	Description

`CLASS`	The CLASS parameter specifies the class name of the MSMQ Connector and the CLR Connector.
	
	The path that you specify for the CLASS parameter must be the following:
	
	`com.exstream.CLRConnector.MSMQConnector`
	
	Example:
	
	`CLASS=com.exstream.CLRConnector.MSMQConnector`
	
	There is no default value for this parameter.

`DLL`	The DLL parameter specifies the file path location of the MSMQ DLL file. **Example:** `C:\Program Files\OpenText\Exstream\Exstream 16.2.0\MSMQConnector.dll` There is no default value for this parameter.

Optional Parameters for the MSMQ Connector

To configure additional parameters for the MSMQ Connector, you must specify values for the following parameters.

Optional parameters for the MSMQ Connector

Parameter	Description
`HOSTNAME`	The `HOSTNAME` parameter specifies the host name of the MSMQ server. **Example:** `HOSTNAME=http://MyServer1.exstream.com` There is no default value for this parameter.
`QUEUENAME`	The `QUEUENAME` parameter specifies the name of the message queue used by the application. The `QUEUENAME` parameter supports the use of DBCS characters. **Example:** `QUEUENAME=MyQueue1` There is no default value for this parameter.
`TIMEOUT`	The `TIMEOUT` parameter specifies the number of seconds before the connector times out. If a message is not received within this time, the request fails. **Example:** `TIMEOUT=10` The default value for this parameter is `10`.
`TTL`	The `TTL` parameter specifies the Time To Live (duration). The `TTL` parameter sets up the time-out for preview configuration. **Example:** `TTL=20` The default value for this parameter is `30`.

11.2.4 Client Methods for the MSMQ Connector

After you have configured the connector in Design Manager, assigned the connector to the necessary objects in Design Manager, and specified values for the parameters that you want to use, you can configure client methods for the MSMQ Connector. You configure client methods for the MSMQ Connector to create client-side applications and send commands through the connector, which lets you control more of the connector, message queue, and engine relationship. For example, you can specify methods that send the driver file to the requested queue to wait for generated documents from the engine. You define the parameters for the following methods in the client API that you create for the MSMQ Connector.

- “Specifying the Generate Method” below
- “Specifying the Place Method” on the next page
- “Specifying the Send Method” on the next page
- “Initiating the Send Method” on page 224

Specifying the Generate Method

The Generate method sends the customer driver file to the request queue, and waits for the generated document on the response queue. This method is derived from the PreviewEngine class.

You must use the following command structure:

```
public virtual PreviewResult Generate(requestMessageQueue,
responseMessageQueue,bytes);
```

To specify the Generate method, you must configure the following parameters.

Parameters for the MSMQ Connector Generate Method

Parameter	Description
`requestMessageQueue`	The `requestMessageQueue` parameter specifies the name of the requesting Message Queue. **Example:** `requestMessageQueue=ABCMessageQueue` There is no default value for this parameter.
`responseMessageQueue`	The `responseMessageQueue` parameter specifies the name of the response message queue. **Example:** `responseMessageQueue=XYZMessageQueue` There is no default value for this parameter.

Parameters for the MSMQ Connector Generate Method, continued

Parameter	Description
`bytes`	The `bytes` parameter specifies the customer driver file in bytes (as a byte array). **Example:** `bytes=byte[]b1` There is no default value for this parameter.

Specifying the Place Method

The Place method inserts the customer driver file into the given queue. This method is derived from the OrderEngine class.

You must use the following command structure:

```
public virtual void Place(requestMessageQueue, bytes);
```

To specify the Place method, you must configure the following parameters.

Parameters for the MSMQ Connector Place Method

Parameter	Description
`requestMessageQueue`	The `requestMessageQueue` parameter specifies the name of the message queue. **Example:** `requestMessageQueue=ABCMessageQueue` There is no default value for this parameter.
`bytes`	The `bytes` parameter specifies the customer driver file in bytes (as a byte array). **Example:** `bytes=byte[]b1` There is no default value for this parameter.

Specifying the Send Method

The Send method sends the given control message to the engine, per the request queue name. This method is derived from the EngineController class.

You must use the following command structure:

```
public virtual void Send(requestMessageQueue, controlMessage);
```

To specify the Send method, you must configure the following parameters.

Parameters for the MSMQ Connector Send Method

Parameter	Description
`requestMessageQueue`	The `requestMessageQueue` parameter specifies the name of the message queue. **Example:** `requestMessageQueue=ABCMessageQueue` There is no default value for this parameter.
`controlMessage`	The `controlMessage` parameter specifies the name of the control message. **Example:** `controlMessage=SLEEP` There is no default value for this parameter.

Initiating the Send Method

To initiate the Send method, you must use the following engine command messages.

Engine command messages for the MSMQ Connector Send Method

Engine command	Description
`SHUTDOWN`	The `SHUTDOWN` engine command instructs the DDA routine to perform a delayed shutdown. The shutdown does not process until all other messages are cleared from the input queue.
`STOP`	The `STOP` engine command instructs the DDA routine to shutdown. The message is added to the queue with an expiration of 60 seconds. The `STOP` engine command clears the message before restarting the engine(s), if the engines need to be restarted within 60 seconds of the last engine shutting down.

11.2.5 Preview Classes for the MSMQ Connector

After configuring client-side applications for the MSMQ Connector, you can use information derived from the message header to understand the types of requests you are receiving and how they are being processed in the engine. The information that the preview class returns from the current message contains information derived from the message header and can be used to gather data regarding the types of messages being sent from your message queue.

To interpret the data returned from the message header, you can use the following preview class property information.

Preview properties for the MSMQ Connector Preview Result Class

Property	Description	Data Type
`Message`	The `Message` property contains the message in current operation.	byte

Preview properties for the MSMQ Connector Preview Result Class, continued

Property	Description	Data Type
`PageCount`	The `PageCount` property contains the number of pages in the current document.	int
`PDL`	The `PDL` property contains the identification of the Printer Definition Language (PDL).	int
`ReturnCode`	The `ReturnCode` property contains the return code for the current document.	int
`TempFileName`	The `TempFileName` property contains the temporary file used to store the message data.	string
`UserData`	The `UserData` property contains the current user data.	byte
`Version`	The `Version` property contains the MSMQ Connector version.	int

11.3 Testing the MSMQ Connector

For best results, before using the MSMQ Connector in production, you should test your MSMQ Connector configuration. Testing the configuration of the MSMQ Connector helps to prevent errors that can occur during production runs.

To test the MSMQ Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - All data source files to be used with the connector
   - Initialization file with specified parameters
2. Verify that you have assigned all of the relevant data files to the connector object.
3. Verify that you have assigned an output to the connector object.
4. Package your application and run the engine.
5. Verify that your output is correct based on the configuration of the MSMQ Connector.

Chapter 12: The SOAP Connector

The SOAP Connector lets you use Web services as a source for customer data, so that the Web service can generate documents. Although neither are required, you can use the SOAP Connector with the Exstream Engine as a Web Service (EWS) or with the Web Services Interface (WSI) modules. The SOAP Connector uses an auxiliary layout file to map the responses received from the Web service so that the engine can populate placeholder variables defined in an application. The auxiliary layout file contains XML that makes up the SOAP payload (the information contained within the XML <body> tags). To deliver the SOAP payload to a Web service and create a link between the engine and the Web service, you must configure the auxiliary layout file and assign the auxiliary layout file to a report file or reference file.

For example, suppose you use a Web service to utilize multiple communication protocols, such as HTTP and SOAP. Representatives in the field gather customer information in order to generate quotes for insurance policies. After the representative gathers the necessary information, the representative submits a request to the main offices. The communication protocol used by your organization is SOAP and HTTP. Your organization also uses Exstream Design and Production as their document generation software and the SOAP Connector as the interface between Exstream and the Web service is used to receive requests and submit responses. Using an application as a template for insurance quotes, a representative submits a request for policy prices. The SOAP Connector receives the requests and communicates with the engine. The engine generates the customer document and the SOAP Connector communicates with the Web service to send a PDF to the representative in the field.

This chapter discusses the following topics:

- “Preparing the SOAP Connector” below
- “Configuring the SOAP Connector” on the next page
- “Testing the SOAP Connector” on page 238

12.1 Preparing the SOAP Connector

To prepare the SOAP Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the `SOAP_Connector_[Version]_[Release].zip` file.

   The SOAP Connector module also contains ZIP files for the Exstream Engine as a Web Service (EWS). Although Exstream distributes both the SOAP Connector and EWS as the same module, each product provides different functionality that you must configure separately.

   For more information about EWS, see “Exstream Engine as a Web Service” on page 69.

2. Extract the contents of the `SOAPConnector.jar` executable JAR archive file to a directory on your machine.
3. For best results, make sure any applications that include test files package and run in batch mode with no errors.

12.2 Configuring the SOAP Connector

The SOAP Connector uses web services as data sources and makes the engine accessible to the web services. The web service is then used to distribute generated documents from the engine. To configure the SOAP Connector, you must create data files that contain the information the web services and engine requires in order to receive requests for documents, generate the documents, and finally, to distribute the documents.

To configure the SOAP Connector, you must complete the following tasks:

1. “Configuring the SOAP Connector in Design Manager” on the next page
2. “Configuring an Auxiliary Layout File” on page 230
3. “Configuring a Reference File” on page 231
4. “Configuring a Report File” on page 231
5. “Configuring a Call Type” on page 232
6. “Setting Up Your Initialization File” on page 235
7. “SOAP Connector Parameters” on page 237

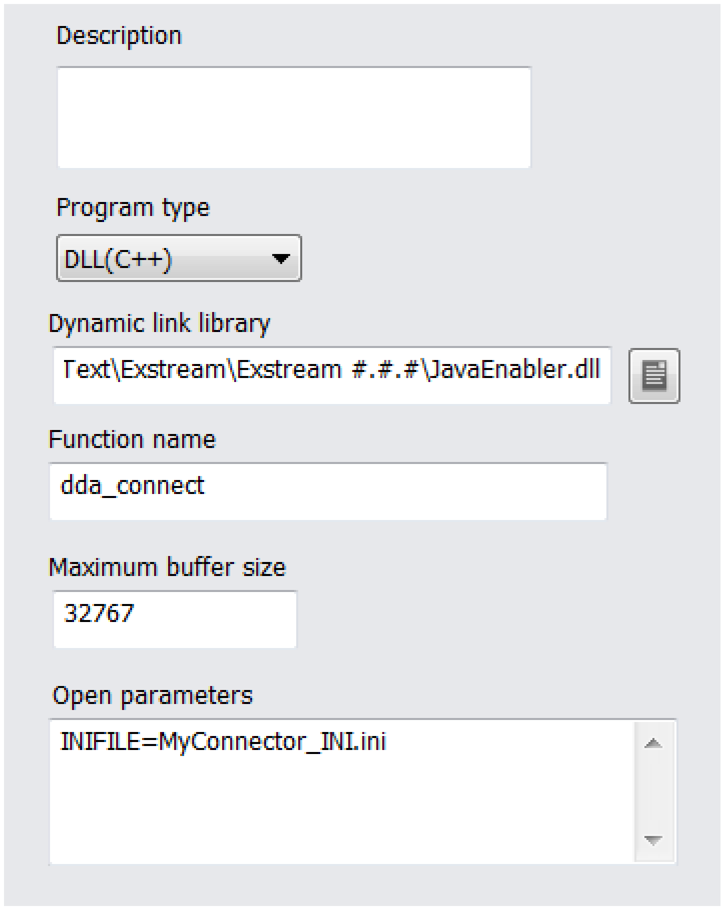
12.2.1 Configuring the SOAP Connector in Design Manager

To configure the SOAP Connector in Design Manager, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the **Program Type** drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the Java Enabler DLL, or click to go to the file: For example: `C:\Program Files\OpenText\Exstream\Exstream <version>\JavaEnabler.dll`
Specify the type of function name	In the **Function name** box, enter `dda_connect`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on page 235.

Example of the completed connector object properties for the SOAP Connector



Description

Program type: DLL(C++)

Dynamic link library: Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name: dda_connect

Maximum buffer size: 32767

Open parameters: INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Connectors in Design Manager" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning the connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Assign the connector to a placeholder variable.

   For more information about assigning the connector to a data file, see "Assigning a Connector Object to a Placeholder Variable" on page 26.

4. Assign the connector to an output.

   For more information about assigning the connector to a data file, see "Assigning a Connector Object to an Output" on page 27.

5. Assign the connector to an output queue.

For more information about assigning the connector to a data file, see "Assigning a Connector Object to an Output Queue" on page 28.

12.2.2 Configuring an Auxiliary Layout File

Before you define the reference or report file, you must first create an auxiliary layout file that has at least one sample record. This auxiliary layout file maps the request that the SOAP Connector sends to the web service. The purpose of the auxiliary layout file is to map requested information to variables in the application. After you have configured the auxiliary layout file, you must then assign the auxiliary layout file to a reference or report file. The information contained in the auxiliary layout file becomes the SOAP payload (the information that appears between the <body> tags), therefore the layout file contents must be in well-formed XML format.

In order for an application to use the SOAP Connector as a reference file, you must make sure that you create the auxiliary layout file according to the specifications of the call type that you will use to handle web service calls.

For layout information and examples of call types, see "Configuring a Call Type" on page 232.

To create an auxiliary layout file for the SOAP Connector, you must complete the following steps:

1. In Design Manager, in the Library, go to **Data Files** heading.
2. In the Library, right-click the type of data file that you want to create. For example, select **SBCS Data File** for a new SBCS data file.

   The **New Data File** dialog box opens.
3. In the **Name** box, enter a name.
4. In the **Description** box, enter a description (optional).
5. From the **File type** drop-down list, select **Auxiliary layout**.
6. From the **File format** drop-down list, select **XML data file**.
7. Click **Finish**.

   The auxiliary layout file opens in the Property Panel.
8. In the **Sample layout** box, click 📄 to select the required layout file.
9. Save the file.
10. In the Edit Panel, map the auxiliary layout file.

The variables that you map to the auxiliary layout file must also be mapped in the customer driver file. The values for these variables become the customer key(s) sent to the web service.

12.2.3 Configuring a Reference File

If you want to use the customer driver data file to point the engine to a standard data source, such as a flat file or an ODBC database, you must use a reference file set up as a DDA routine for the SOAP Connector. The customer data that you map to the auxiliary layout file becomes the customer key that is sent to the web service. The reference file DDA routine sends the customer key as a SOAP payload to the web service.

To configure a reference file for the SOAP Connector, you must complete the following steps:

1. In Design Manager, in the Library, go to **Data Files** heading.
2. In the Library, right-click the type of data file that you want to create. For example, select **SBCS Data File** for a new SBCS data file.

   The **New Data File** dialog box opens.
3. In the **Name** box, enter a name.
4. In the **Description** box, enter a description (optional).
5. From the **File type** drop-down list, select **Reference file**.
6. From the **File format** drop-down list, select **XML data file**.
7. Click **Finish**.

   The reference file opens in the Property Panel.
8. On the **Basic** tab, in the **Reference key layout** drop-down list, select your auxiliary layout file.
9. To identify the method to store data passed with this reference file, select **User routine** in the **Access** drop-down list.
10. Since the web service receives and sends data in XML format, leave the **File format** as **XML data file**.
11. Click the **Advanced** tab.
12. From the **IO time** drop-down list, select the option appropriate for your application. For example, if you want the engine to read data records after the engine reads the initialization file, select the **After initialization files read** option.

    For details about **IO time**, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

12.2.4 Configuring a Report File

If you want to archive customer data, you must configure a report file for the SOAP Connector set up as a DDA routine for the SOAP Connector. The report file uses the information in the auxiliary layout file that you created to send information to the web service for archiving. You can use the SOAP Connector report file DDA routine to archive information such as customer account numbers and customer names. This information can be used by your organization for many purposes, including internal audits of customer applications produced in a period of time.

To configure a report file for SOAP Connector, you must complete the following steps:

1. In Design Manager, in the Library, go to **Data Files** heading.
2. In the Library, right-click the type of data file that you want to create. For example, select **SBCS Data File** for a new SBCS data file.

   The **New Data File** dialog box opens.
3. In the **Name** box, enter a name.
4. In the **Description** box, enter a description (optional).
5. From the **File type** drop-down list, select **Report file**.
6. From the **File format** drop-down list, select **XML data file**.
7. Click **Finish**.

   The report file opens in the Property Panel.
8. In the **Data mapping source** box, select your auxiliary layout file.
9. Click the **Advanced** tab.
10. From the **IO time** drop-down list, select **Completion of all customers**.

    For more information about options in the **IO time** drop-down list, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

12.2.5 Configuring a Call Type

You must define the call type that the web service uses to handle calls. The web service uses the call type to determine the method to use to process data. For example, if the data being sent to the web service is encoded, then a Remote Procedure call type is necessary to identify information such as input parameters and return values. If the data being sent to the web service simply defines a payload that contains a document, then a Document call type is necessary so that the web service can interpret the data accordingly.

You specify the call type that you want to use in the initialization file that you create for the SOAP Connector parameters.

You must select one of the following call types:

- **RPC (Remote Procedure Call)**—Takes an encoding approach to data. This call type identifies a method name, input parameters, and return values in a "send procedure/receive result" technique to messaging. To use the call type of Remote Procedure Call, set the style parameter as `STYLE=rpc`.

- **Document**—Takes a literal approach to data. This call type defines a payload that contains a document. To use the call type of Document, set the style parameter as `STYLE=Document`.

For more information about the `STYLE` parameter, see "SOAP Connector Parameters" on page 237.

This section discusses the following topics:

- "Layout of an RPC Call Type" below
- "Layout of a Document Call Type" on the next page

Layout of an RPC Call Type

The format that you use for an RPC call type requires typed arguments. The SOAP Connector uses the 5.1 encoding standards (section 5.1 in the SOAP 1.1 specification) to promote compatibility across multiple platforms. The SOAP Connector does not support arrays or structs.

In the first tag you specify the named operation to be called on the web service. Some web services require a namespace target object in this tag. Subsequent tags in the layout of the RPC call type form the customer key(s) to the web service (one tag per argument). You map variables in the application to these tags. The values for these variable(s) are included in the customer driver file and are passed through the reference file to the SOAP Connector to the web service.

The following example defines the makeup of an RPC call type layout configured specifically for the SOAP Connector. The sample configuration shows an RPC call type that is configured to request customer data from a web service. The `<getCustomerData>` tag identifies the operation. The engine substitutes actual customer numbers from the customer driver file at run time

For example:

```
<soapenv:Envelope xmlns:soapenv="http:www.w3.org/2003/05/soap-envelope">
<soapenv:Body>
     <exst:getCustomerData xmlns:exst="urn:exstream-customer-service">
        <customer xsi:type="xsd:int">986572</customer>
     </exst:getCustomerData>
   </soapenv:Body>
</soapenv:Envelope>
```

The web service receives the RPC call type layout and processes the data. The following example shows a simplified Java-based string that the web service uses to process the data in the RPC call:

```
public class CustomerLookupImpl implements CustomerLookup
{
       String customerData = null;
public CustomerLookupImpl ()
{
       super ();
}
public String getCustomerData (int customerId)
{
       customerData = lookupCustomer (customerId);
       return customerData;
}
```

Layout of a Document Call Type

The format that you use for a Document call type is well-formed XML that contains customer data, such as the customer account number and customer name. In the following example, when the SOAP Connector requests customer data from a web service using the Document call type, the web service expects an XML structure containing the customer identification number and the customer account number as data input.

The `<CustomerData>` tag is the document root tag that passes to the web service. The web service requires the information in the `<customer>` and `<account>` tags.

When you map the auxiliary layout file, map the arguments (`1234567` and `0987654` in this example), to the corresponding variables in the customer driver file. The engine substitutes customer numbers from the customer driver file at run time.

For example:

```
<exst:CustomerData xmlns:exst="urn:exstream-customer-service">
   <customer>1234567</customer>
   <account>0987654</account>
</exst:getCustomerData>


<soapenv:Envelope>
   <soapenv:Body>
      <exst:CustomerData xmlns:exst="urn:exstream-customer-service">
         <customer>986572</customer>
         <account>876541</account>
      </exst:CustomerData>
   </soapenv:Body>
</soapenv:Envelope>
```

The web service receives the Document call type layout and processes the data. The following example shows a simplified Java-based string that the web service uses to process the data in the Document call:

```
public class CustomerLookupImpl implements CustomerLookup
{
   Document customerData = null;
public CustomerLookupImpl ()
{
    super ();
}
public Document getCustomerData (Document customerRequest)
{
   customerData = lookupCustomer (customerRequest);
   return customerData;
}
```

When you use the Document call type, you have the option to include a `<Header>` element as the first child in the `<SOAPEnvelope>` tag. This element can contain any required tags, but is typically used to pass security tokens. The SOAP Connector uses the `<header>` tag to include required tags in the `<Header>` of the SOAP message.

To include tags in the SOAP `<Header>` tag, you must place the tags under a `<header>` tag in the file. When you include the `<Header>` tag, you must meet the following requirements:

- The reference file requires a root tag set to include the <header> tag set.
- The `<header>` tag must be the first child of the root tag.
- The root tags are placed in the `<SOAPBody>` tag.
- Child elements of the `<header>` tag must be namespaced.

For example:

```
<exst:CustomerData xmlns:exst="urn:exstream-customer-service">
   <header>
      <exst:username=juser>
      <exst:password=XXXXX>
   </header>
   <customer>986572</customer>
   <account>876541</account>
<exst:CustomerData>
```

12.2.6 Setting Up Your Initialization File

After you configure the SOAP Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager instead of listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications requires different connector behavior.

To create an initialization file for the SOAP Connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For more information about SOAP Parameters, see "SOAP Connector Parameters" on the next page.

Specifying the Type of Initialization File to Use

Some of the parameters available for a connector support the use of double-byte characters. The use of double-byte characters in a connector parameter gives the user more flexibility in defining connector behavior. If the connector you configure allows for parameters that support the use double-byte characters, you must first specify the type of initialization file you will be using to define the parameters. For example, if your initialization file for the SOAP Connector lists the `MESSAGEFILE` parameter with a message file name value that uses double-byte characters, you must first specify a Unicode initialization file by defining the `UNICODEINIFILE` parameter.

To specify the type of initialization file to use, you must specify a value for one of the following parameters.

Parameters for specifying an initialization file

Parameter	Description
`INIFILE`	The `INIFILE` parameter specifies an initialization file containing parameters that support only the use of single-byte characters. When specifying connector behavior with an initialization file, the `INIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. **Example:** `INIFILE=MyINI.ini` There is no default value for this parameter.

Parameters for specifying an initialization file, continued

Parameter	Description
`UNICODEINIFILE`	The `UNICODEINIFILE` parameter specifies an initialization file that contains one or more supported parameters with Unicode characters. When specifying connector behavior with an initialization file that contains Unicode characters in parameters, the `UNICODEINIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. When the engine does not detect a byte-order marker, you must use the `UNICODEINIFILE` parameter so that the engine can interpret Unicode characters correctly. **Example:** `UNICODEINIFILE=MyUnicodeINI.ini` There is no default value for this parameter.

12.2.7 SOAP Connector Parameters

The following information details the parameters you can use with the SOAP Connector. Parameters for the SOAP Connector define how the SOAP Connector communicates with Web services. You must include the following parameters for the SOAP connector to communicate with Web services and the engine.

Required SOAP Connector parameters

Parameter	Description
`MODE`	The `MODE` parameter indicates the operating mode. Select from one of the following values: • `reference`—The `reference` value specifies that the SOAP Connector operates in reference mode. • `report`—The `report` value specifies that the SOAP Connector operates in report mode. **Example:** `MODE=reference` There is no default value for this parameter.
`STYLE`	The `STYLE` parameter indicates the call type. Select from one of the following values: • Remote Procedure Call (`rpc`)—The `rpc` value specifies that the SOAP Connector use a remote procedure call type • `document`—The `document` value specifies that the SOAP Connector use a document call type. Always use `document` when `Mode=Report`. **Example:** `STYLE=rpc` There is no default value for this parameter.

Required SOAP Connector parameters, continued

Parameter	Description
`URL`	The `URL` parameter identifies the Web service location endpoint (the exact form is platform dependent). The `URL` parameter supports the use of DBCS characters. **Example:** `URL=http://myserver:9080/WebServicesAPIs/sca/com/` There is no default value for this parameter.
`ROOT`	The `ROOT` parameter specifies a tag name used as the root tag for the response of a Web service call. The `ROOT` parameter is only valid for Web service systems that support multiple return items. The `ROOT` parameter is also used if the Web service returns a response with multiple structs or array values. **Example:** `ROOT=root` The default value for this parameter is `root`.
`SOAPACTION`	The `SOAPACTION` parameter specifies the method in use. The `SOAPACTION` parameter is optional on some Web service platforms. On the .NET Web service platform this parameter is required. Refer to the SOAP requirements for your platform vendor. **Example:** `SOAPACTION=urn` There is no default value for this parameter.
`DEBUGLEVEL`	The `DEBUGLEVEL` parameter lets the user perform debugging on the SOAP Connector. Information is sent to the console. Select from one of the following values: • `INFO`—The `INFO` value opens the request for read operations and responses include any header information. For write operations, the `INFO` value opens the URL endpoint when the write (to the Web service) occurs. • `ALL`—The `ALL` value provides information from `INFO` value and HTTP Header information and provides the server return code and a dump of all the initialization file properties. **Example:** `DEBUGLEVEL=INFO` The default value for this parameter is `INFO`.

12.3 Testing the SOAP Connector

For best results, before using the SOAP Connector in production, you should test your configuration. Testing the configuration of the SOAP Connector helps to prevent errors that can occur during production runs.

To test the SOAP Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
    - All data source files to be used with the connector
    - Initialization file with specified parameters
2. Verify that you have assigned all of the relevant data files to the connector object.
3. Verify that you have assigned an output queue to the connector object.
4. Package the application and run the engine with the ONDEMAND switch in your control file.
5. Verify that your output is correct based on the configuration of the SOAP Connector.

Chapter 13: The Watched Directory Connector

The Watched Directory Connector lets you dynamically control data access and file processing by specifying the location of a specific directory, the type of files to be read, and the disposition of those files after their data has been accessed. You specify a directory location (a watched directory) to watch for data source files. The Watched Directory Connector polls a directory looking for work to process. When the directory receives a completely written file, the Watched Directory Connector reads the file and sends the appropriate information to the engine. Exstream uses the data to create the required documents and store them in an output file. The Watched Directory Connector is an efficient way to import file-based data directly into the packaging process.

For example, suppose you want to offer customers the opportunity to request insurance quotes on demand. Using Watched Directory Connector, you can set up your existing database with the Exstream production environment to wait for each customer request and produce the insurance quote as soon as the request is made.

This chapter discusses the following topics:

- "Preparing the Watched Directory Connector" below
- "Configuring the Watched Directory Connector" on the next page
- "Testing the Watched Directory Connector" on page 248

13.1 Preparing the Watched Directory Connector

To prepare the Watched Directory Connector for configuration, testing, and use in production, you must complete the following steps:

1. From My Support, download the appropriate ZIP file for your operating system.

   The naming convention is `WatchedDirectory_[VERSION]_[PLATFORM].zip`.

   For example, `WatchedDirectory_1_5_008_AIX.zip` is Watched Directory Connector version 1.5.008 for the AIX environment.

2. Extract the contents of the ZIP file you selected to the directory that contains the Exstream engine.

3. Create a watched directory for the data source files that you want the Watched Directory Connector to read and pass to the engine.

4. For best results, make sure any applications that include test files package and run in batch mode with no errors.

13.2 Configuring the Watched Directory Connector

The Watched Directory connector constantly scans the watched directory, looking for files and works under the presumption that only files with a specific file extension are to be used as customer driver files for the application. Specifying a file extension for the Watched Directory Connector lets you keep other files, such as log files or report files used in engine runs that include the connector, in the same directory without the Watched Directory Connector accessing the files and causing errors. To configure the Watched Directory Connector, you must create a connector object in the Design Manager Library and assign the connector to the appropriate data files and data sources.

To configure the Watched Directory Connector, you must complete the following tasks:

1. "Configuring the Watched Directory Connector in Design Manager" on the next page
2. "Setting Up Your Initialization File" on page 244
3. "Watched Directory Connector Parameters" on page 245

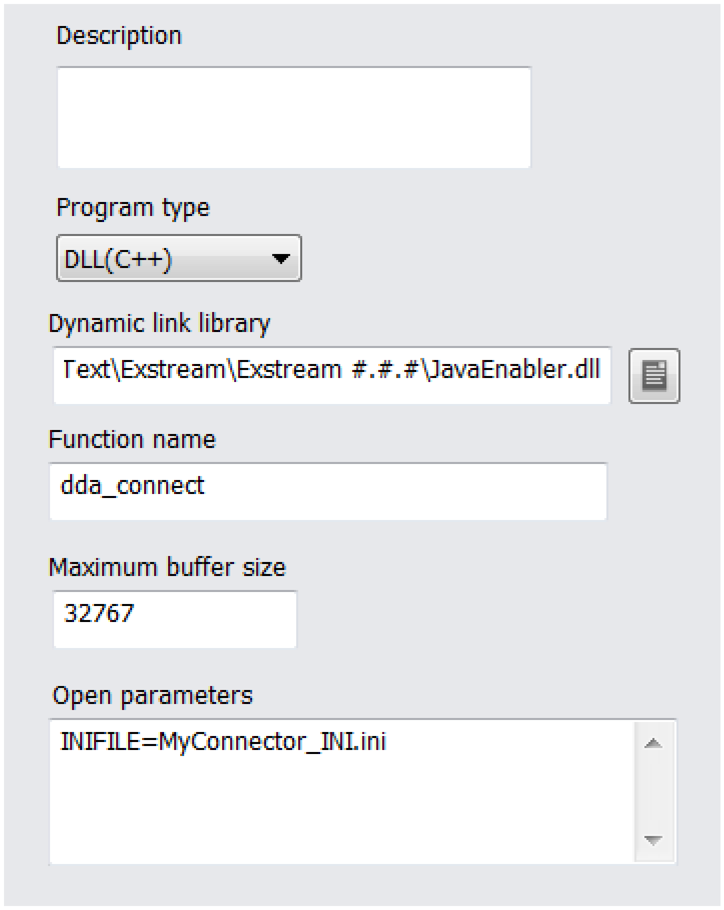
13.2.1 Configuring the Watched Directory Connector in Design Manager

To configure the Watched Directory Connector, you must complete the following steps:

1. Create a connector object in Design Manager with the following settings:

To	Do this
Specify the program type	From the Program Type drop-down list, select **DLL(C++)**.
Specify the DLL	In the **Dynamic Link Library** box, enter the path and file name to the Watched Directory Connector DLL, or click 🗎 to go to the file. • For Windows, the file name is `WatchedDirectory.dll` • For all UNIX and Linux, except HP-UX, the file name is `WatchedDirectory.so` • For HP-UX, the file name is `WatchedDirectory.sl`
Specify the type of function name	In the **Function name** box, enter `processRec`.
Specify the buffer size for data handling	In the **Maximum buffer size** box, enter the buffer size accommodate data handling requirements. Typically, this is set to `32767`, which is the maximum buffer size the production environment allows. If you set the buffer size to an amount higher than 32767, the engine processes information 32,767 bytes at a time.
Specify parameters	Set up your initialization file and enter each parameter on a single line. For more information about setting up your initialization file to define parameters, see "Setting Up Your Initialization File" on page 244.

Example of the completed connector object properties for the Watched Directory Connector



Program type: DLL(C++)

Dynamic link library: Text\Exstream\Exstream #.#.#\JavaEnabler.dll

Function name: dda_connect

Maximum buffer size: 32767

Open parameters: INIFILE=MyConnector_INI.ini

For more information about creating a connector object in Design Manager, see "Configuring Dynamic Data Access" on page 21.

2. Assign the connector to a data file for data input.

   For more information about assigning a connector to a data file, see "Assigning a Connector Object to a Data File for Data Input" on page 24.

3. Place your data source files into the watched directory that you created during pre-configuration tasks

   For more information about setting up the watched directory, see "Preparing the Watched Directory Connector" on page 240

   You can add files to the watched directory you created at any time the engine is still running. In each watch cycle, the connector looks for files with a particular file extension.

13.2.2 Setting Up Your Initialization File

After you configure the Watched Directory Connector in Design Manager and set additional configurations, such as assigning the connector to a data file, you must set up your initialization file for your configuration and control parameters. An initialization file contains operational details using both required and optional parameters. Parameters act as containers in which you specify certain definitions that control the overall behavior of your connector. The initialization file that you set up for connectors lets you use a text file as a single source for the remaining configuration settings. You specify name of the initialization file in the **Open parameters** box on the connector object properties in Design Manager. If you reference the initialization file in Design Manager instead of listing each parameter in the **Open parameters** box in Design Manager, you can change parameters on demand when applications require different connector behavior.

To create an initialization file for the Watched Directory connector parameters:

1. Open a text editor such as Notepad or WordPad.
2. In your text editor, list each parameter that you want to use, both required and optional, on a separate line.
3. Save the file with the `.ini` file extension.
4. In Design Manager, on the connector object properties, enter the name of the initialization file in the **Open parameters** box.
5. Save the connector object.

For information about Watched Directory parameters, see “Watched Directory Connector Parameters” on the next page.

Specifying the Type of Initialization File to Use

Some of the parameters available for a connector support the use of double-byte characters. The use of double-byte characters in a connector parameter gives the user more flexibility in defining connector behavior. If the connector you configure allows for parameters that support the use double-byte characters, you must first specify the type of initialization file you will be using to define the parameters. For example, if your initialization file for the Watched Directory Connector lists the `DIRECTORY` parameter with a value that uses DBCS characters, you must first specify a Unicode initialization file by defining the `UNICODEINIFILE` parameter.

Parameters for specifying an initialization file

Parameter	Description
`INIFILE`	The `INIFILE` parameter specifies an initialization file containing one or more of the parameters listed below. When specifying connector behavior with an initialization file, the `INIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. **Example:** `INIFILE=MyINIFile.ini` There is no default value for this parameter.
`UNICODEINIFILE`	The `UNICODEINIFILE` parameter specifies an initialization file that contains one or more supported parameters with Unicode characters. When specifying connector behavior with an initialization file that contains Unicode characters in parameters, the `UNICODEINIFILE` parameter must be the only parameter you specify in the **Open parameters** box on the connector object's properties in Design Manager. The `UNICODEINIFILE` parameter is necessary for the engine to interpret Unicode characters correctly when the engine does not detect a byte order marker. **Example:** `UNICODEINIFILE=MyUnicodeINIFILE.ini` There is no default value for this parameter.

13.2.3 Watched Directory Connector Parameters

The following information details the parameters you can use with the Watched Directory Connector. Additionally, you have optional engine commands that you can use with the Watched Directory Connector.

This section discusses the following topics:

- "Watched Directory Parameters" below
- "Specifying Engine Commands for the Watched Directory Connector" on page 247

Watched Directory Parameters

To configure the parameters for the Watched Directory Connector, you must include at least the DIRECTORY parameter. All other parameters are optional.

To configure parameters for the Watched Directory Connector, you must specify values for the following parameters.

Watched Directory Connector parameters

Parameter	Description
DIRECTORY	The DIRECTORY parameter specifies the file path name to the directory that the connector watches. The DIRECTORY parameter is the only required parameter. The DIRECTORY parameter supports the use of DBCS characters. **Example:** `DIRECTORY=C:\Users\Administrator\Documents\WDConnector_Docs` There is no default value for this parameter.
TRACE	The TRACE parameter specifies whether to trace connector activity. Select from one of the following values: • NO—The NO value specifies that no connector activity is written. • YES—The YES value specifies that connector activity is written to the standard output location. • LOG—The LOG value specifies the name of the file to write connector activity. If no file location is specified, connector activity is written to the WatchedDirectory.log default location. The LOG option supports the use of DBCS characters in the file name. **Example:** `TRACE=NO` The default value for this parameter is NO.
DRIVER_ EXTENSION	The DRIVER_EXTENSION parameter specifies which file extension that the connector checks for in the watched directory. The connector only accesses files having the specified extension. **Example:** `DRIVER_EXTENSION=dat` The default value for this parameter is dat. The DRIVER_EXTENSION parameter supports the use of DBCS characters.
WAIT_PERIOD	The WAIT_PERIOD parameter specifies the number of seconds the connector waits following an unsuccessful attempt, before attempting to locate a file in the watched directory. **Example:** `WAIT_PERIOD=2` The default value for this parameter is 2.
INACTIVITY_ COUNT	The INACTIVITY_COUNT parameter specifies the number of times the connector checks for a file during a watch cycle. **Example:** `INACTIVITY_COUNT=0` The default value for this parameter is 0.

Watched Directory Connector parameters, continued

Parameter	Description
`DISPOSE`	The `DISPOSE` parameter specifies that the connector delete, rename, or move an accessed data source file. Select from one of the following values: • `Delete`—The `Delete` value deletes each accessed data source file. • `Rename`—The `Rename` value renames the data source file extension according to the file extension that you specify. The default extension is `DAA`. The `Rename` value supports the use of DBCS characters when specifying the extension. • `Move`—The `MOVE` value moves files to another directory that you specify. If you do not specify a directory, the default is the watched directory with its last character changed to another value. The `Move` option supports the use of DBCS characters when specifying the directory. **Example:** `DISPOSE=Rename` The default value for this parameter is `Rename`.

Specifying Engine Commands for the Watched Directory Connector

As an optional part of the configuration of the Watched Directory Connector, you can specify the run-time behavior of the Watched Directory Connector by placing a control file in the watched directory.

The control file must be named `WATCHED_DIRECTORY.CONTROL`. The Watched Directory Connector checks that this file exists. If the file exists, the Watched Directory Connector reads the file and the engine processes the following commands.

Engine commands to use with Watched Directory Connector

Engine command	Description
`STOP`	The `STOP` engine command shuts engine down immediately.
`SHUTDOWN`	The `SHUTDOWN` engine command shuts engine down when there are no further files to process.
`SLEEP`	The `SLEEP` engine command sleeps for the specified number of seconds.
`FLUSH`	The `FLUSH` engine command closes and reopens the engine report and message files.

Note: If you have multiple engines watching the same directory, leave the control file in your watched directory until all engines have had a chance to process the commands. If you use a control file to shut down your engines, do not forget to remove the control file before restarting the engines.

For more information on control files, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

The Watched Directory Connector supports multiple engine runs, but you must use one of the following setup options to avoid errors:

- **Rename the extension**—You can set up each engine to check for a different extension in the same directory. For example, engine 1 checks for extension `xyz` and changes it to `abc`. Engine 2 checks for extension `abc`, and can either change the extension, move the files, or delete the files. Engine 2 should not change the extension back to `xyz` because engine 1 would try to read it again.
- **Delete the file**—You can set up each engine to delete files from different directories. For example, engine 1 checks for the file in directory X, reads it, and deletes it. Engine 2 checks for the file in directory Z, reads it, and deletes it.
- **Move the file**—You can set up each engine to move files into different directories. For example, engine 1 checks for the file in directory X, reads it, and moves it to directory Y. Engine 2 checks for the file in directory Y, reads it, and can either change its extension, move it, or delete it. Engine 2 should not move the file back to directory X because engine 1 would try to read it again.

13.3 Testing the Watched Directory Connector

For best results, before using the Watched Directory Connector in production, you should test your configuration. Testing the configuration of the Watched Directory Connector helps to prevent errors that can occur during production runs.

To test the Watched Directory Connector, you must complete the following steps:

1. Verify that you have added the following files to your application:
   - All data source files to be used with the connector
   - Watched directory that contains your data source files
   - Initialization file and/or watched directory control file that contains parameters and commands
2. Verify that you have assigned the connector object to all relevant data files in your application.
3. Package your application and run the engine with the ONDEMAND engine switch in your control file.
4. Verify that your output is correct based on the configuration of the Watched Directory Connector.