



Building Custom Adapters

**SAP Sybase Event Stream
Processor 5.1 SP04**

DOCUMENT ID: DC01982-01-0514-01

LAST REVISED: November 2013

Copyright © 2013 by SAP AG or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and other countries. Please see <http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark> for additional trademark information and notices.

Contents

CHAPTER 1: Introduction	1
Input and Output Adapters	1
Managed and Unmanaged Adapters	2
Subscribing to Data with Input Adapters	3
Publishing Data with Output Adapters	3
Editing Adapter Property Sets	4
Adapter Logging Configuration	5
CHAPTER 2: Event Stream Processor Adapter Toolkit	9
Preconfigured Adapters Included with the Adapter Toolkit	10
Create a Custom Adapter	12
Accessing Adapter Toolkit API Reference Information ...	13
Transporter Modules	14
Transporters Currently Available from SAP	14
File Input Transporter Module Configuration	17
File Output Transporter Module Configuration	20
FTP Input Transporter Module Configuration	21
FTP Output Transporter Module Configuration ...	23
HTTP Output Transporter Module Configuration	24
JDBC Input Transporter Module Configuration ...	27
JDBC Output Transporter Module Configuration	29
JMS Input Transporter Module Configuration	31
JMS Output Transporter Module Configuration ..	33
Socket Input Transporter Module Configuration .	34

Socket Output Transporter Module	
Configuration	36
Building a Custom Transporter Module	37
Formatter Modules	40
Formatters Currently Available from SAP	40
CSV String to ESP Formatter Module	
Configuration	43
ESP to CSV String Formatter Module	
Configuration	44
ESP to JSON Stream Formatter Module	
Configuration	45
ESP to Object List Formatter Module	
Configuration	47
ESP to String List Formatter Module	
Configuration	47
ESP to XML String Formatter Module	
Configuration	48
ESP to XMLDOC String Formatter Module	
Configuration	48
JSON String to ESP Formatter Module	
Configuration	49
JSON Stream to JSON String Formatter Module	
Configuration	51
Object List to ESP Formatter Module	
Configuration	51
Stream to String Formatter Module	
Configuration	52
String to Stream Formatter Module	
Configuration	53
String List to ESP Formatter Module	
Configuration	54
XML String to ESP Formatter Module	
Configuration	54
XMLDoc Stream to ESP Formatter Module	
Configuration	54

Datatype Mapping for Formatters	56
Building a Custom Formatter Module	58
Batch Processing	61
Schema Discovery	62
Implementing Schema Discovery in a Custom Adapter	63
Guaranteed Delivery	64
Enabling Guaranteed Delivery for an Input Transporter	64
EspConnector Modules	66
EspSubscriber Configuration	66
EspMultiStreamSubscriber Configuration	67
EspPublisher Configuration	69
EspMultiStreamPublisher Configuration	71
Event Stream Processor Configuration	73
Configuring a New Adapter	75
Starting an Adapter	86
Stopping an Adapter	88
Adapter Toolkit Examples	90
Running an Adapter Example	91
Running the Schema Discovery Adapter Example	92
Adapter Toolkit: Sample cxml File for Input Adapters ...	93
Adapter Toolkit: Sample cxml File for Output Adapters	93
Debugging a Custom Adapter	94

CHAPTER 3: Creating Custom External Adapters

Using SDKs	97
Java External Adapters	97
Connecting to a Project	97
Creating a Publisher	97
Sample Java Code for addRow	98
Subscribing Using Callback	98
Subscribe Using Direct Access Mode	100

- Publish Using Callback100
- C/C++ External Adapters101**
 - Getting a Project101
 - Publishing and Subscribing101
 - handleData103
- .Net External Adapters103**
 - Connecting to the Event Stream Processor Server ...103
 - Connecting to a Project104
 - Publishing104
 - Connecting to a Subscriber105
 - Subscribing Using Callback Mode105

- CHAPTER 4: Adapter Integration Framework109**
 - Cnxml Configuration File109**
 - Creating a Cnxml File for a Custom Adapter110**
 - Example Cnxml Configuration File112
 - External Adapter Properties114
 - External Adapter Commands116
 - User-Defined Parameters and Parameter Substitution
.....117
 - Autogenerated Parameter Files119
 - configFilename Parameter120

- APPENDIX A: Adapter Parameter Datatypes121**

- APPENDIX B: Date and Timestamp Formats for Input
Adapters125**

- APPENDIX C: Date and Timestamp Formats for
Output Adapters127**

- APPENDIX D: Internal Adapter API129**

The Adapter Shared Utility Library	129
Sample Model File	129
The Adapter Configuration File	130
DLL Export Functions	131
Adapter Setup Functions	131
Adapter Life Cycle Functions	131
Miscellaneous Functions	133
Schema Discovery for Internal Custom Adapters	134
DLL Import Functions	134
Error Handler Functions	135
Parameter Handler Functions	135
Data Conversion Functions	136
Callback Functionality	139
Adapter Run States	141
Sample Custom Internal Adapter Implementation	141
Index	143

Contents

CHAPTER 1 Introduction

SAP® Sybase® Event Stream Processor includes an extensive set of input and output adapters that you can use to subscribe to and publish data. Additional specialized adapters for Event Stream Processor are available from SAP as optional add-ons. You can also write your own adapters to integrate into the ESP Server, design them to handle a variety of external requirements that standard adapters cannot.

Event Stream Processor provides an adapter toolkit for building custom external (Java) adapters.

Event Stream Processor also provides a variety of SDKs that you can use to build custom external adapters in a number of programming languages, such as C, C++, Java, and .NET (C#, Visual Basic, and so on). For versions supported for SDKs, see the *Installation Guide*.

Finally, Event Stream Processor also provides an internal API that you can use to build custom internal adapters. However, SAP recommends that you use the adapter toolkit instead of the internal API as:

- The adapter toolkit offers a quicker, easier, and more flexible method for creating custom adapters.
- The adapter toolkit offers standard modules that you can re-use to create your own custom adapters.
- Internal adapters do not have any advantages over external adapters.
- The internal adapter API is being deprecated.

Input and Output Adapters

Input and output adapters enable Event Stream Processor to send and receive messages from dynamic and static external sources and destinations.

External sources or destinations can include:

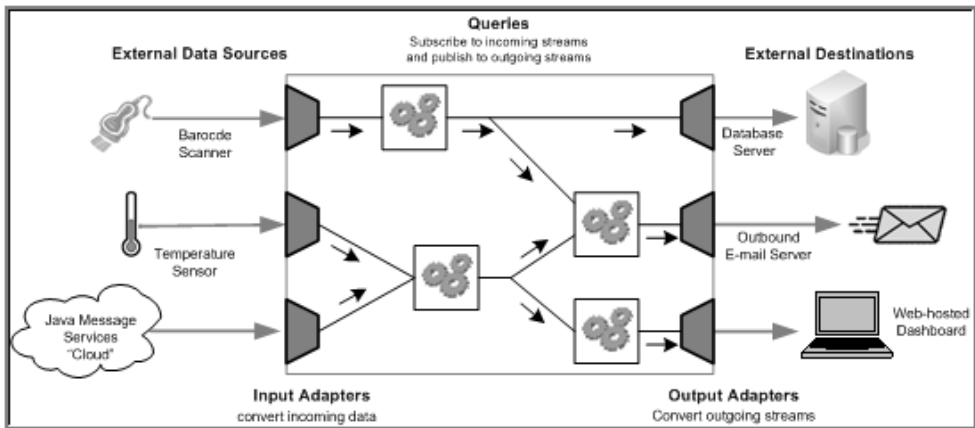
- Data feeds
- Sensor devices
- Messaging systems
- Radio frequency identification (RFID) readers
- E-mail servers
- Relational databases

Input adapters connect to an external datasource and translate incoming messages from the external sources into a format that is accepted by the ESP Server. Output adapters translate

rows that are published by Event Stream Processor into message formats that are compatible with external destinations and send those messages downstream.

The following illustration shows a series of input adapters that translate messages from a temperature sensor, bar code scanner, and a Java Message Service (JMS) cloud into formats that are compatible with Event Stream Processor. After the data is processed using various queries within Event Stream Processor, output adapters convert the resulting rows into updates that are sent to an external database server, e-mail server, and Web services dashboard.

Figure 1: Adapters in Event Stream Processor



Managed and Unmanaged Adapters

An external adapter can be developed and configured to support running in managed-only mode, unmanaged-only mode, or to support both modes.

Managed external adapters:

- Are started and stopped by the ESP Server with an ESP project.
- Have a cnxml adapter configuration file that is configured within the ESP Studio.
- Are referenced in a CCL **ATTACH ADAPTER** statement.

Unmanaged external adapters:

- Start and stop independently of the ESP Server and ESP projects.
- Are configured independently.
- Are not referenced in a CCL **ATTACH ADAPTER** statement.

Subscribing to Data with Input Adapters

Subscribe to data from an external datasource, and use an input stream or window to send the data to Event Stream Processor.

Backslashes in CCL files generate errors, because in that context, they are control characters.

1. Assess the input data, determining which sets or subsets of data you want to pull into Event Stream Processor.

2. Choose an input adapter that is suited for this task.

If the datasource uses datatypes that are not supported by the ESP Server, the Server maps the data to an accepted datatype. Review the associated mapping description for your adapter in the *Adapters Guide*.

3. Create an input stream or window.

4. Use the **CREATE SCHEMA** statement to define the structure for incoming data within this stream or window.

5. (Skip this step if using an unmanaged adapter) Use the **ATTACH ADAPTER** statement to attach your adapter to the newly created stream or window, and specify values for the adapter properties.

Use the **DECLARE** block and **parameters** qualifier to define default adapter parameter values before you attach your adapter. Once you create the **ATTACH ADAPTER** statement, you can set the adapter properties to the parameter values you declared.

Note: You can bind declared parameters to a new value only when a module or project is loaded.

6. Start the ESP project. If you are using an unmanaged adapter, start it manually.

Next

For detailed information about configuring individual Event Stream Processor-supplied adapters, datatype mappings, and schema discovery, see the *Adapters Guide*. For detailed information on CCL queries and statements, such as the **ATTACH ADAPTER**, **CREATE SCHEMA**, and **DECLARE** statements, see the *Programmers Reference Guide*.

Publishing Data with Output Adapters

Use an output stream or window to publish data from Event Stream Processor to an external datasource.

Backslashes in CCL files generate errors, because in that context, they are control characters.

CHAPTER 1: Introduction

1. Assess the output data, determining which sets or subsets of data you want to send to an external datasource.
2. Choose an output adapter suited for this task.

If the output destination uses datatypes that are not supported by the ESP Server, the Server maps the data to an accepted datatype. Review the associated mapping description for your adapter in the *Adapters Guide* to ensure that the resulting datatype is permitted by the external data destination.

3. Create an output stream or window.
4. Use the **CREATE SCHEMA** statement to define the structure for outgoing data within this stream or window.
5. (Skip this step if using an unmanaged adapter) Use the **ATTACH ADAPTER** statement to attach your adapter to the output stream or window, and set values for the adapter properties.

To declare default parameters for your adapter properties, use the **DECLARE** block and **parameter** qualifier to define default parameter values before you attach your adapter. Once you create the **ATTACH ADAPTER** statement, you can set the adapter properties to the parameter values you declared.

Note: You can bind declared parameters to a new value only when a module or project is loaded.

6. Start the ESP project. If you are using an unmanaged adapter, start the adapter manually.

Next

For detailed information on configuring individual Event Stream Processor-supplied adapters, datatype mappings, and schema discovery, see the *Adapters Guide*. For detailed information on CCL queries and statements, such as the **ATTACH ADAPTER**, **CREATE SCHEMA**, and **DECLARE** statements, see the *Programmers Reference Guide*.

Editing Adapter Property Sets

Adapter property sets are reusable groups of properties that are stored in the project configuration file. Use the ESP Studio CCR Project Configuration editor to define adapter property sets and store them in the associated .ccr file.

Property sets appear in a tree format, and individual property definitions are shown as children to property sets.

1. In the CCR Project Configuration editor, select the **Adapter Properties** tab.
2. (Optional) To create a list of adapter property sets that correspond to the **ATTACH ADAPTER** statements in the main CCL file for the project, click **Add from CCL**.
3. To create a new adapter property node, click **Add**.

4. In the Property Set Details pane, define a name for the property node.
5. To add a new property to a property set, right-click the set and select **New > Property**.

Note: You can add as many property items to a property set as required.

6. To configure a property:
 - a) In the Property Details pane, define a name for the property.
 - b) Enter a value for the property.
7. (Optional) To encrypt the property value:
 - a) Select the property value and click **Encrypt**.
 - b) Enter the required fields, including Cluster URI and credential fields.
 - c) Click **Encrypt**.

The value, and related fields, are filled with randomized encryption characters.

Note: To reset the encryption, click **Encrypt** beside the appropriate field. Change the values, as appropriate, then click **Reset**.

8. To remove items from the All Adapter Properties list:
 - Right-click a property set and select **Remove**, or
 - Right-click a property and select **Delete**.

Adapter Logging Configuration

Specific adapters that are currently available from SAP use the `log4j` API to log errors, warnings, and debugging messages.

To specify the location of the logging file, use the **Log4jProperty** parameter within the adapter configuration file. You can modify the logging levels within this file or the `%ESP_HOME%\adapters\\config\log4j.properties` file, which is used by default. Set the `ADAPTER_CLASSPATH` environment variable to point to the configuration directory of each adapter for which you are configuring logging.

The logging levels in `log4j.properties` are:

Level	Description
OFF	Logs no events.
FATAL	Logs severe errors that prevent the application from continuing.
ERROR	Logs potentially recoverable application errors.
WARN	Logs events that possibly lead to errors.
INFO	Logs events for informational purposes.

Level	Description
DEBUG	Logs general debugging events.
TRACE	Logs fine-grained debug messages that capture the flow of the application.
ALL	Logs all events.

Note: Setting the log level to DEBUG or ALL may result in large log files. The default value is INFO.

Here is a sample `log4j.properties` file:

```
# Set root logger level to INFO and set appenders to stdout, file and
email
log4j.rootLogger=INFO, stdout, R

# stdout appender
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{MM-dd-yyyy
HH:mm:ss.SSS} %p [%t] (%C{1}.%M) %m%n
log4j.appender.stdout.Threshold=INFO

# file appender
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
log4j.appender.R.File=logs/rtviewadapter.log
log4j.appender.R.DatePattern='.'yyyy-MM-dd
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{MM-dd-yyyy
HH:mm:ss.SSS} %p [%t] (%C{1}.%M) %m%n
log4j.appender.R.Threshold=INFO

# email appender
log4j.appender.email=org.apache.log4j.net.SMTPAppender
log4j.appender.email.To=your.name@yourcompany.com
log4j.appender.email.From=alert.manager@yourcompany.com
log4j.appender.email.SMTPHost=yourmailhost
log4j.appender.email.BufferSize=1
log4j.appender.email.Subject=RTView Adapter Error
log4j.appender.email.layout=org.apache.log4j.PatternLayout
log4j.appender.email.layout.ConversionPattern=%d{MM-dd-yyyy
HH:mm:ss.SSS} %p [%t] (%C{1}.%M) %m%n
log4j.appender.email.Threshold=ERROR

log4j.logger.com.sybase.esp=INFO
```

The `log4j.rootLogger` option sets the default log behavior for all the sub-loggers in the adapter. In addition to the root logger, the adapter contains various sub-loggers that control logging for specific adapter functions.

Setting the `log4j.rootLogger` to any value more verbose than INFO may produce excess information. If you explicitly set the log level for a sub-logger, you overwrite the

default setting for that particular logger. In this way, you can make sub-loggers more verbose than the default. The names for Event Stream Processor related loggers contain the string `com.sybase.esp`.

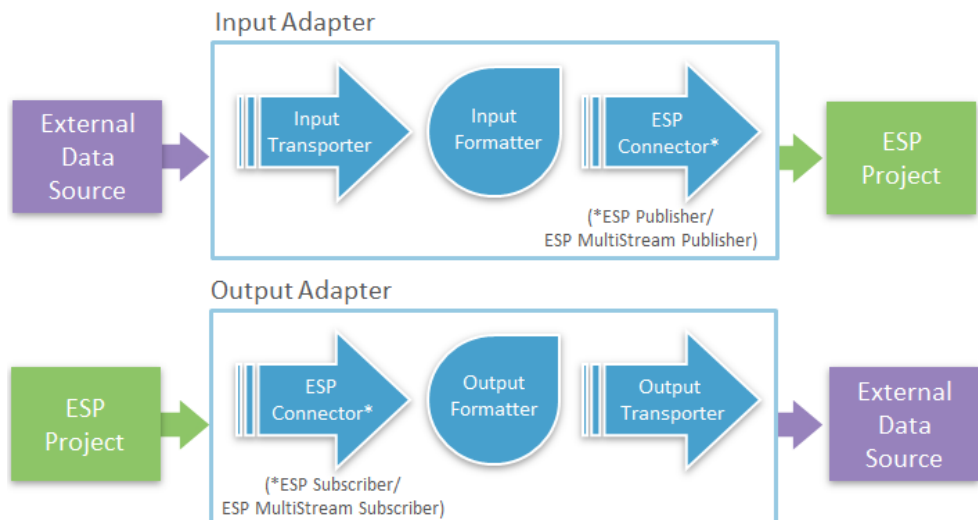
Event Stream Processor Adapter Toolkit

Use the Event Stream Processor adapter toolkit to quickly build custom external adapters using Java. Adapters built using the toolkit consist of various component modules configured together to deliver data to and publish data from Event Stream Processor. Module types include transporters (for interacting with external transports), formatters (for converting data from one format to another), and ESP connectors (for subscribing or publishing to ESP).

The toolkit includes numerous transporters, formatters, and ESP connectors that can be configured in various combinations by an administrator. You can also combine these out-of-the-box modules with custom modules created by a Java developer.

The adapter toolkit allows you to implement:

- An input adapter to act as a datasource for the ESP Server.
- An output adapter to act as a data destination and deliver data from streams in ESP.
- A dual-direction adapter to act as both a datasource and data destination for Event Stream Processor.
- Guaranteed delivery (GD) to minimize loss of data during transfer of input data.
- Schema discovery to automatically discover schema for your custom input and output adapter.



Preconfigured Adapters Included with the Adapter Toolkit

Event Stream Processor includes various preconfigured and ready for use adapters that have been created using the adapter toolkit.

You can use these adapters as reference examples when creating your own adapters. Additionally, you can reuse individual transporter and formatter modules from these adapters in your own custom adapters.

For detailed information about these adapters, see the *Adapters Guide*.

Table 1. Input Adapters

Adapter Name	Description
File CSV Input	Obtains CSV data from files on a local hard disk and publishes it to Event Stream Processor.
File JSON Input	Takes JSON messages from JSON files, and publishes them to Event Stream Processor.
File XML Document Input	Loads data from an XML document into a project in Event Stream Processor.
File XML Record Input	Reads XML list text files and inputs this data into Event Stream Processor.
FTP CSV Input	Obtains CSV data from an FTP server and publishes it to Event Stream Processor.
FTP XML Input	Reads data from an XML document on an FTP server into Event Stream Processor.
JDBC Input	Receives data from tables in a database and inputs it into Event Stream Processor.
JMS CSV Input	Reads CSV data from a JMS server and outputs this data into Event Stream Processor.
JMS Object Array Input	Receives object array data from a JMS server and publishes it to Event Stream Processor.
JMS XML Input	Obtains XML list string messages from a JMS server and publishes them to Event Stream Processor.
Socket CSV Input	Obtains CSV string data from a Socket server and publishes it to Event Stream Processor.

Adapter Name	Description
Socket JSON Input	Obtains streaming data from the socket server, formats data into JSON format and inputs it into Event Stream Processor.
Socket XML Input	Obtains XML data from a Socket server and publishes it to Event Stream Processor.

Table 2. Output Adapters

Adapter Name	Description
File CSV Output	Reads rows from Event Stream Processor and writes this data into a specified CSV file.
File JSON Output	Takes data from Event Stream Processor, formats it into JSON format, and sends it to a JSON file.
File XML Document Output	Outputs data from a project in Event Stream Processor into an XML document.
File XML Record Output	Reads rows from Event Stream Processor and writes this data into XML list files.
FTP CSV Output	Takes data from Event Stream Processor, formats it to CSV format, and saves it to a file on an FTP server.
FTP XML Output	Reads XML data from an ESP project, writes it to an XML document, and uploads this file to the FTP server.
HTTP Client Output	Obtains data from an ESP stream and outputs it to an HTTP server.
JDBC Output	Sends data from Event Stream Processor to a database table.
JMS CSV Output	Sends CSV data from Event Stream Processor to a JMS server.
JMS Object Array Output	Takes data from Event Stream Processor, formats it into object array format, and sends it to a JMS server.
JMS XML Output	Takes XML data from Event Stream Processor, formats it to XML list format, and sends it to a JMS server.

Adapter Name	Description
Socket CSV Output	Takes data from Event Stream Processor, formats it into CSV format, and outputs it to a Socket server.
Socket JSON Output	Takes JSON data from Event Stream Processor, formats it to ByteBuffer, and transports it to the Socket server in streaming format.
Socket XML Output	Takes data from Event Stream Processor, formats it to XML list format, and outputs it to a Socket server.

Create a Custom Adapter

Use the ESP adapter toolkit to create a custom adapter. You can do this by combining transporter and formatter modules that are provided with the adapter toolkit, by writing your own custom transporter and formatter modules, or by combining existing modules with custom ones.

1. *Building a Custom Transporter Module*

Use the ESP adapter toolkit to build a custom transporter module to use within the adapter instance of your choice.

2. *Building a Custom Formatter Module*

Use the ESP adapter toolkit to build a custom formatter module to use within the adapter instance of your choice.

3. *Enabling Guaranteed Delivery for an Input Transporter*

(Optional) Enable guaranteed delivery (GD) in a custom input transporter by implementing the `com.sybase.esp.adapter.framework.event.AdapterRowEventListener` interface, registering the `GdAdapterEventListener` class, and adding and setting the `<GDMode>` parameter to true for the `EspPublisher` or `EspMultistreamPublisher`.

4. *Implementing Schema Discovery in a Custom Adapter*

(Optional) Use interfaces and functions from the adapter toolkit to implement schema discovery in a transporter and formatter module. There are two types of schema discovery: nonsampling and sampling. Use non-sampling schema discovery when the transporter can fully determine schema on its own. Use sampling schema discovery when the transporter cannot determine the schema and passes this data to the formatter to generate the schema.

5. *Configuring a New Adapter*

Configure a new adapter by creating a configuration file for it. The configuration file defines the adapter component chain through which data is processed, and the connection to Event Stream Processor.

6. *Creating a Cnxml File for a Custom Adapter*

Create a cnxml configuration file for your custom external adapter so that you can configure the adapter in the ESP Studio, and start and stop it with an ESP project.

7. *Starting an Adapter*

You can start an adapter either in unmanaged or managed mode. In unmanaged mode, the adapter is started separately from the ESP project, and in managed mode, the adapter is started with the ESP project.

8. *Stopping an Adapter*

You can stop an adapter either in unmanaged or managed mode. In unmanaged mode, the adapter is stopped separately from the ESP project, and in managed mode, the adapter is stopped with the ESP project.

See also

- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Debugging a Custom Adapter* on page 94
- *Formatter Modules* on page 40
- *Transporter Modules* on page 14
- *EspConnector Modules* on page 66

Accessing Adapter Toolkit API Reference Information

API documentation about methods, functions, and other programming building blocks.

To access the API documentation:

1. Navigate to %ESP_HOME%\doc\adapertoolkit on Windows or \$ESP_HOME/doc/adapertoolkit on Linux/Solaris.
2. Launch `index.html`.

See also

- *Building a Custom Transporter Module* on page 37
- *Transporters Currently Available from SAP* on page 14
- *Building a Custom Formatter Module* on page 58
- *Formatters Currently Available from SAP* on page 40
- *Create a Custom Adapter* on page 12
- *Debugging a Custom Adapter* on page 94
- *Formatter Modules* on page 40
- *Transporter Modules* on page 14
- *EspConnector Modules* on page 66

Transporter Modules

A transporter module is the interface that interacts with external data sources by obtaining data from a data source or outputting data to a data destination.

Event Stream Processor supports two types of transporters: row-based and stream-based.

Row-based transporters obtain and output data in row format, such as a database transporter. These transporters work with `AdapterRow` instances, which are containers for one or more records or rows as they flow from one module (transporter, formatter, or ESP connector) to the next. You can add multiple records as objects within a `List` of a single `AdapterRow` object. The `AdapterRow` has a timestamp and block flags that control how its records are communicated to and from Event Stream Processor. See *Envelopes and Transactions* for additional details.

Stream-based transporters deal with streaming data, such as a socket transporter. These transporters work with `ByteStream` or `ByteBuffer` instances, which represent a continuous stream of data.

See also

- *Formatter Modules* on page 40
- *EspConnector Modules* on page 66
- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Create a Custom Adapter* on page 12
- *Debugging a Custom Adapter* on page 94

Transporters Currently Available from SAP

The adapter toolkit includes numerous transporter modules with Event Stream Processor. You can reuse these modules to create a custom adapter instance.

Input transporters obtain data from external data sources and output this data into Event Stream Processor. The format of this output data is specified in the Output Datatype column of the table below.

`AdapterRow` is a container for one or more records or rows as they flow from one module (transporter, formatter, or ESP connector) to the next. You can add multiple records as objects within a `List` of a single `AdapterRow` object. The `AdapterRow` has a timestamp and block flags that control how its records are communicated to and from Event Stream Processor. See *Envelopes and Transactions* for additional details.

`AepRecord` is the class that represents stream records. This is the type that the ESP publisher expects as the data member in the `AdapterRow` instances that it receives from the previous module. This is also the type used in the `AdapterRow` instances that the ESP subscriber passes on to the next module.

Table 3. Standard Input Transporters

Name	Mode (Streaming/ Row)	Output Datatype	Description
File Input Transporter	Can be both (depends on configuration)	Java.lang.String or Java.nio.ByteBuffer	<p>In row mode, the transporter reads data from local files, wraps data with string, and sends it to the next module that is configured in the adapter configuration file.</p> <p>In streaming mode, the transporter reads data from local files, wraps it with ByteStream, and passes it to the next module that is configured in the adapter configuration file.</p>
FTP Input Transporter	Streaming	Java.nio.ByteBuffer	Reads binary data from files on an FTP server, wraps it up with ByteStream, and passes it to the next module that is configured in the adapter configuration file.
JDBC Input Transporter	Row	Java.util.List<Java.lang.Object>	Reads database records from a database using JDBC, and sends data records to the next module that is configured in the adapter configuration file.
JMS Input Transporter	Row	Java.lang.String or AepRecord or Java.util.List<Java.lang.Object>	Receives JMS messages from a JMS server, and sends this data to the next module that is configured in the adapter configuration file.

Name	Mode (Streaming/ Row)	Output Datatype	Description
Socket Input Transporter	Streaming	Java.nio.ByteBuffer	Reads binary data from a socket interface, wraps it with ByteStream, and passes it to the next module that is configured in the adapter configuration file.

Output transporters obtain data from Event Stream Processor and output it to external data sources. The format of this input data is specified in the Input Datatype column of the table below.

Table 4. Standard Output Transporters

Name	Mode (Streaming/ Row)	Input Datatype	Description
File Output Transporter	Can be both (depends on configuration)	Java.lang.String or Java.nio.ByteBuffer	In row mode, the transporter obtains string data from the previous module configured in the adapter configuration file, and writes this data to a local file. In streaming mode, the transporter obtains ByteStream data from the previous module configured in the adapter configuration file, and writes this data to a local file.
FTP Output Transporter	Streaming	Java.nio.ByteBuffer	Obtains ByteStream data from the previous module that is configured in the adapter configuration file, and saves it to a file on an FTP server.

Name	Mode (Streaming/Row)	Input Datatype	Description
HTTP Output Transporter	Row	AepRecord	Obtains data from an ESP stream and outputs it to an HTTP server.
JDBC Output Transporter	Row	Java.util.List<Java.lang.Object>	Obtains row-based data from the previous module that is configured in the adapter configuration file, and saves it into a database table using JDBC.
JMS Output Transporter	Row	Java.lang.String or AepRecord or Java.util.List<Java.lang.Object>	Obtains data from the previous module that is configured in the adapter configuration file and sends it to a JMS server.
Socket Output Transporter	Streaming	Java.nio.ByteBuffer	Obtains ByteStream data from the previous module that is configured in the adapter configuration file, and outputs it through a socket interface.

See also

- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Building a Custom Transporter Module* on page 37

File Input Transporter Module Configuration

The File Input transporter reads data from local files, wraps the data with `string`, and sends it to the next module specified in the adapter configuration file. Set values for this transporter in the adapter configuration file.

The File Input transporter supports schema discovery.

XML Element	Description
Dir	<p>Type: string</p> <p>(Required) Specify the absolute path to the data files which you want the adapter to read. For example, <username>/<foldername>. No default value.</p> <p>To use Hadoop system files, use an HDFS folder uri instead of a local file system folder. For example, <code>hdfs://<hdfsserver>:9000/<foldername>/<subfoldername>/<leafname></code>.</p> <p>To use Hadoop, download the binaries for Hadoop version 1.2.1 from http://hadoop.apache.org. Copy the <code>hadoop-core.jar</code> file (for example, for version 1.2.1 <code>hadoop-core-1.2.1.jar</code>) to <code>%ESP_HOME%\adapters\framework\libj</code>. Ensure you use a stable version rather than a beta.</p> <p>Use a forward slash for both UNIX and Windows paths.</p>
File	<p>Type: string</p> <p>(Required) Specify the file you want the adapter to read or the regex pattern to filter the files on a given directory. See the DynamicMode element. No default value.</p>
AccessMode	<p>Type: string</p> <p>(Required) Specify an access mode:</p> <ul style="list-style-type: none"> • rowBased – the adapter reads one text line at a time. • Streaming – the adapter reads a preconfigured size of bytes into a buffer. <p>No default value.</p>

XML Element	Description
DynamicMode	<p>Type: <code>string</code></p> <p>(Advanced) Specify a dynamic mode:</p> <ul style="list-style-type: none"> • Static – the adapter reads the file specified in the Dir and File elements. • dynamicFile – the adapter reads the file specified in the Dir and File elements and keeps polling the new appended content. The polling period is specified in the PollingPeriod element. • dynamicPath – the adapter polls all the new files under the Dir element. Also, the File element acts as a regex pattern and filters out the necessary files. <p>The default value is <code>Static</code>. If DynamicMode has been set to <code>dynamicPath</code> and you leave the File element empty, the adapter reads all the files under the specified directory.</p> <p>An example regex pattern is <code>".*\.\.txt"</code>, which selects only files that end with <code>".txt"</code>. In regex patterns, you must include an escape character, <code>"\"</code>, before meta chars to include them in the pattern string.</p>
PollingPeriod	<p>Type: <code>integer</code></p> <p>(Advanced) Define the period, in seconds, to poll the specified file or directory. Set this element only if the value of the DynamicMode element is set to <code>dynamicFile</code> or <code>dynamicPath</code>.</p> <p>The default value is <code>0</code>, which, along with all other values less than <code>0</code>, turns off polling.</p>
RemoveAfterProcess	<p>Type: <code>boolean</code></p> <p>(Optional) If this property is set to <code>true</code>, the file is removed from the directory after the adapter processes it. This element takes effect if the value of the DynamicMode element is set to <code>dynamicPath</code> and ignored if it is set to <code>dynamicFile</code> instead.</p> <p>The default value is <code>false</code>.</p>

XML Element	Description
ScanDepth	<p>Type: integer</p> <p>(Optional) Specify the depth of the schema discovery. The adapter reads the number of rows specified by this element value when discovering the input data schema.</p> <p>The default value is three.</p>

File Output Transporter Module Configuration

The File Output transporter obtains data from the previous module specified in the adapter configuration file and writes it to local files. Set values for this transporter in the adapter configuration file.

XML Element	Description
Dir	<p>Type: string</p> <p>(Required) Specify the absolute path to the data files that you want the adapter to write to. For example, <username>/<foldername>. The default value is ".", meaning the current directory in which the adapter is running.</p> <p>To use Hadoop system files, use an HDFS folder instead of a local file system folder. For example, <code>hdfs://<hdfsserver>:9000/<foldername>/<subfoldername>/<leaffoldername></code>.</p> <p>To use Hadoop, download the binaries for Hadoop version 1.2.1 from http://hadoop.apache.org. Copy the <code>hadoop-core.jar</code> file (for example, for version 1.2.1 <code>hadoop-core-1.2.1.jar</code>) to <code>%ESP_HOME%\adapters\framework\libj</code>. Ensure you use a stable version rather than a beta.</p> <p>Use a forward slash for both UNIX and Windows paths.</p>
File	<p>Type: string</p> <p>(Required) Specify the file to which the adapter writes.</p>

XML Element	Description
AccessMode	Type: <code>string</code> (Required) Specify an access mode: <ul style="list-style-type: none"> • rowBased – the adapter writes one text line at a time into the file. • Streaming – the adapter writes the raw data in <code>ByteBuffer</code> into the file. No default value.
AppendMode	Type: <code>boolean</code> (Optional) If set to true, the adapter appends the data into the existing file. If set to false, the adapter overwrites existing content in the file. Default value is false.

FTP Input Transporter Module Configuration

The FTP Input transporter reads binary data from files on an FTP server, wraps it up with `ByteBuffer`, and sends it to the next module that is configured in the adapter configuration file. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	Type: <code>string</code> (Required) Specify the server name or IP address of the FTP server to which you are connecting.
Port	Type: <code>integer</code> (Required) Specify the port address for the FTP server to which you are connecting. The default value is 21.
LoginType	Type: <code>enum</code> (Required) Specify the login type for the FTP server: <code>normal</code> or <code>anonymous</code> .
User	Type: <code>string</code> (Required if LoginType is set to <code>normal</code>) Specify the login account for the FTP server.

XML Element	Description
Password	Type: <code>string</code> (Required if LoginType is set to normal) Specify the login password for the FTP server.
FtpFilePath	Type: <code>string</code> (Required) Specify the absolute path to the data files in the FTP server.
FtpFileName	Type: <code>string</code> (Required) Specify the data files in the FTP server.
MaxBlockSize	Type: <code>int</code> (Required) Specify the maximum size of any data blocks transferred from the FTP server. The default value is 2048.
TransferMode	Type: <code>string</code> (Required) Specify the transfer mode for the FTP connection: active or passive. The default value is active.
RetryPeriod	Type: <code>second</code> (Required) Specify the period of time, in seconds, during which to try and reconnect to the FTP server if you disconnect unexpectedly. The default value is 30.
RetryCount	Type: <code>integer</code> (Required) Specify the number of times to attempt to reconnect to the FTP server if you disconnect unexpectedly. The default value is 0.

FTP Output Transporter Module Configuration

The FTP Output transporter obtains data from the previous module configured in the adapter configuration file, and saves it to files on the FTP server. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	Type: <code>string</code> (Required) Specify the server name or IP address of the FTP server to which you are connecting.
Port	Type: <code>integer</code> (Required) Specify the port address for the FTP server to which you are connecting. The default value is 21.
LoginType	Type: <code>enum</code> (Required) Specify the login type for the FTP server. There are two valid types: <code>normal</code> and <code>anonymous</code> .
User	Type: <code>string</code> (Required if LoginType is set to <code>normal</code>) Specify the login account for the FTP server.
Password	Type: <code>string</code> (Required if LoginType is set to <code>normal</code>) Specify the login password for the FTP server.
FtpFilePath	Type: <code>string</code> (Required) Specify the absolute path to the data files in the FTP server.
FtpFileName	Type: <code>string</code> (Required) Specify the data files in the FTP server.
MaxBlockSize	Type: <code>int</code> (Required) Specify the max data block size to transfer to the FTP server. The default value is 2048.

XML Element	Description
Overwrite	Type: <code>boolean</code> (Required) If set to true, the transporter overwrites the file on the FTP server, if it exists. If this element is set to false, the transporter appends the output to the end of the existing file. The default value is false.
TransferMode	Type: <code>string</code> (Required) Specify the transfer mode for the FTP connection. There are two valid values: active or passive. The default value is active.
RetryPeriod	Type: <code>second</code> (Required) Specify the period of time, in seconds, during which to try and reconnect to the FTP server if you disconnect unexpectedly. The default value is 30.
RetryCount	Type: <code>integer</code> (Required) Specify the number of times to attempt to reconnect to the FTP server if you disconnect unexpectedly. The default value is 0.

HTTP Output Transporter Module Configuration

The HTTP Output transporter obtains data from an ESP stream and outputs it to an HTTP server. Set values for this transporter in the adapter configuration file.

XML Element	Description
ProxyHost	Type: <code>string</code> (Advanced) Specify the proxy server hostname. No default value.
ProxyPort	Type: <code>integer</code> (Advanced) Specify the proxy server port. No default value.

XML Element	Description
RetryNumber	Type: <code>integer</code> (Advanced) Specify the number of times to try reconnecting to the HTTP server if the adapter disconnects unexpectedly. The default value is 0.
BodyCharset	Type: <code>string</code> (Optional) Specify the name of a supported charset. The default value is US-ASCII.
UriColumn	Type: <code>integer</code> (Dependent required) Specify the column index of the ESP stream to which the adapter is attached containing the URL to use for sending HTTP requests. This property is required if RequestUrl is empty.
UsernameColumn	Type: <code>integer</code> (Optional) Specify the column index of the ESP stream to which the adapter is attached which contains the user name to use if the HTTP server requires authentication.
PasswordColumn	Type: <code>integer</code> (Optional) Specify the column index of the ESP stream to which the adapter is attached which contains the password to use if the HTTP server requires authentication.
BodyColumn	Type: <code>integer</code> (Required) Specify the column index of the ESP stream to which the adapter is attached which contains the message body content for the HTTP request.
RequestUrl	Type: <code>string</code> (Optional) Specify a URL to use for sending HTTP requests. This URL is used if the UriColumn element is not specified, or if the ESP stream does not have a URL column.

XML Element	Description
RequestUsername	<p>Type: <code>string</code></p> <p>(Optional) Specify the user name to use if the HTTP server requires authentication. This user name is used if the UsernameColumn element is not specified, or if the ESP stream does not have a user name column.</p>
RequestPassword	<p>Type: <code>string</code></p> <p>(Optional) Specify the password to use if the HTTP server requires authentication. This password is used if the PasswordColumn element is not specified, or if ESP stream does not have a password column.</p> <p>This property includes an encrypted attribute which indicates whether the password value is encrypted. If set to true, the password is decrypted using the RSAKeyStore, RSAKeyStorePassword, and RSAKeyStoreAlias properties.</p>
RequestBody	<p>Type: <code>string</code></p> <p>(Optional) Specify the message body content for the HTTP request. This value is used if the BodyColumn element is not specified, or if the ESP stream does not have a body column.</p>
RSAKeyStore	<p>Type: <code>string</code></p> <p>(Dependent required) Specify the location of an RSA keystore file which contains the key used to encrypt and decrypt the password set in the RequestPassword element. This element is required if the password value set in RequestPassword is encrypted.</p>
RSAKeyStorePassword	<p>Type: <code>string</code></p> <p>(Dependent required) Stores the password to the RSA keystore file specified in the RSAKeyStore element. This element is required if the password value set in RequestPassword is encrypted.</p>

XML Element	Description
RSAKeyStoreAlias	Type: <code>string</code> (Dependent required) Specifies the location of the keystore alias. This element is required if the password value set in RequestPassword is encrypted.

JDBC Input Transporter Module Configuration

The JDBC Input transporter reads database records using JDBC and sends them to the next module specified in the adapter configuration file. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	Type: <code>string</code> (Required) Specify the server name of the database to which you are connecting the adapter.
Port	Type: <code>integer</code> (Required) Specify the port number for connecting to the database server.
Username	Type: <code>string</code> (Required) Specify the user name you are using to connect to the database server.
Password	Type: <code>string</code> (Required) Specify the password for connecting to the database server. Includes an "encrypted" attribute that indicates whether the password value is encrypted. If set to true, the password value is decrypted using the RSAKeyStore , RSAKeyStorePassword , and RSAKeyStoreAlias elements.
DbName	Type: <code>string</code> (Required) Specify the database to which you want to connect.

XML Element	Description
DBType	Type: <code>string</code> (Required) Specify the database type of the database to which you want to connect.
DbDriver	Type: <code>string</code> (Required) Specify the JDBC driver class for your JDBC driver.
Table	Type: <code>string</code> (Optional) Specify the name of the table in the target database from which you want the adapter to read.
Query	Type: <code>string</code> (Optional) Specify which SQL query you want the adapter to execute. No default value. Set either the Table or Query element. If you define both elements, the adapter only uses the Query element.
RSAKeyStore	Type: <code>string</code> (Dependent required) Specify the location of an RSA keystore file which contains the key used to encrypt or decrypt the password set in the Password element. This element is required if the password value is encrypted.
RSAKeyStorePassword	Type: <code>string</code> (Dependent required) Stores the password to the RSA keystore file specified in the RSAKeyStore element. This element is required if the password value is encrypted.
RSAKeyStoreAlias	Type: <code>string</code> (Dependent required) Specifies the keystore alias. This element is required if the password value is encrypted.

JDBC Output Transporter Module Configuration

The JDBC Output transporter obtains data from the previous module specified in the adapter configuration file and writes it into a database table using JDBC. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	Type: <code>string</code> (Required) Specify the server name of the database to which you are connecting the adapter.
Port	Type: <code>integer</code> (Required) Specify the port number for connecting to the database server.
Username	Type: <code>string</code> (Required) Specify the username you are using to connect to the database server.
Password	Type: <code>string</code> (Required) Specify the password for connecting to the database server. Includes an "encrypted" attribute indicating whether the password value is encrypted. If set to true, the password value is decrypted using the RSAKeyStore , RSAKeyStorePassword , and RSAKeyStoreAlias elements.
DbName	Type: <code>string</code> (Required) Specify the database name of the database to which you want to connect.
DBType	Type: <code>string</code> (Required) Specify the database type of the database to which you want to connect.
DbDriver	Type: <code>string</code> (Required) Specify the JDBC driver class for your JDBC driver.

XML Element	Description
Table	Type: <code>string</code> (Optional) Specify the name of the table in the target database to which you want the adapter to write.
SqlInsert	Type: <code>string</code> (Optional) Specify the SQL clause you want the adapter to execute. No default value. Set either the Table or SqlInsert element. If you define both elements, the adapter uses only the SqlInsert element.
RSAKeyStore	Type: <code>string</code> (Dependent required) Specify the location of an RSA keystore file which contains the key used to encrypt or decrypt the password set in the Password element. This element is required if the password value is encrypted.
RSAKeyStorePassword	Type: <code>string</code> (Dependent required) Stores the password to the RSA keystore file specified in the RSAKeyStore element. This element is required if the password value is encrypted.
RSAKeyStoreAlias	Type: <code>string</code> (Dependent required) Specifies the keystore alias. This element is required if the password value is encrypted.
BatchSize	Type: <code>integer</code> (Optional) Specify the number of records to process as a batch.
BatchPeriod	Type: <code>integer</code> (Optional) Specify the period for processing a batch, in milliseconds.

JMS Input Transporter Module Configuration

The JMS Input transporter receives JMS messages from a JMS server, and sends it to the next module that is configured in the adapter configuration file. Set values for this transporter in the adapter configuration file.

The transporter sends different Java objects to the next module if different messages are received:

Message Type Received	Java Object to Send
TEXT (TextMessage)	String
OBJARRAY (ObjectMessage)	Object

XML Element	Description
ConnectionFactory	Type: <code>string</code> (Required) Specify the connection factory class name. No default value.
JndiContextFactory	Type: <code>string</code> (Required) Specify the context factory for JNDI context initialization. No default value.
JndiUrl	Type: <code>string</code> (Required) Specify the JNDI URL. No default value.
Destination Type	Type: <code>string</code> (Required) Specify the destination type. Valid values are: <code>QUEUE</code> and <code>TOPIC</code> . The default value is <code>QUEUE</code> .
DestinationName	Type: <code>string</code> (Required) Specify the destination name. No default value.

XML Element	Description
MessageType	<p>Type: <code>string</code></p> <p>(Required) Specify the message type you want the JMS transporter to process:</p> <ul style="list-style-type: none"> • <code>TEXT</code> – for receiving and sending messages in text string. • <code>OBJARRAY</code> – for receiving and sending messages in custom format. <p>No default value.</p>
SubscriptionMode	<p>Type: <code>string</code></p> <p>(Optional) Specify the subscription mode for TOPIC (see the Destination Type element). Valid values are <code>DURABLE</code> and <code>NONDURABLE</code>.</p> <p>Default value is <code>NONDURABLE</code>.</p>
ScanDepth	<p>Type: <code>integer</code></p> <p>(Optional) Specify the depth of the schema discovery. The adapter reads the number of rows specified by this element value when discovering the input data schema.</p> <p>The default value is three.</p>
ClientID	<p>Type: <code>string</code></p> <p>(Required for <code>DURABLE</code> subscription mode) Specifies the client identifier for a JMS client. Can be any string, but must be unique for each topic. No default value.</p> <p>Example: <code>client1</code>.</p>
SubscriptionName	<p>Type: <code>string</code></p> <p>(Required for <code>DURABLE</code> subscription mode) Specifies a unique name identifying a durable subscription. Can be any string, but must be unique within a given client ID. No default value.</p> <p>Example: <code>subscription1</code>.</p>

JMS Output Transporter Module Configuration

The JMS Output transport obtains data from the previous module that is configured in the adapter configuration file, wraps it up, and sends it to a JMS server. Set values for this transporter in the adapter configuration file.

The transporter sends different JMS messages if it receives different datatypes from the previous module:

Datatype Received	JMS Message Type to Send
List<String>	TextMessage
AepRecord	ObjectMessage
Object (others)	ObjectMessage

XML Element	Description
ConnectionFactory	Type: <code>string</code> (Required) Specify the connection factory class name. No default value.
JndiContextFactory	Type: <code>string</code> (Required) Specify the context factory for JNDI context initialization. No default value.
JndiUrl	Type: <code>string</code> (Required) Specify the JNDI URL. No default value.
Destination Type	Type: <code>string</code> (Required) Specify the destination type. Valid values are: QUEUE and TOPIC. The default value is QUEUE.
DestinationName	Type: <code>string</code> (Required) Specify the destination name. No default value.

XML Element	Description
MessageType	<p>Type: <code>string</code></p> <p>(Required) Specify the message type you want the JMS transporter to process:</p> <ul style="list-style-type: none"> • <code>TEXT</code> – for receiving and sending messages in text string. • <code>OBJARRAY</code> – for receiving and sending messages in custom format. <p>No default value.</p>
DeliveryMode	<p>Type: <code>string</code></p> <p>(Optional) Specify the delivery mode type. Valid values are:</p> <ul style="list-style-type: none"> • <code>PERSISTENT</code> • <code>NON_PERSISTENT</code> <p>Default value is <code>PERSISTENT</code>.</p>

Socket Input Transporter Module Configuration

The Socket Input transporter reads binary data from the socket interface, wraps it with `ByteBuffer`, and sends it to the next module that is configured in the adapter configuration file. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	<p>Type: <code>string</code></p> <p>(Required if <code>EpFile</code> is set to null) If the transporter is acting as a socket client, specify the socket server name. If the transporter is acting as a socket server, do not set this element. No default value.</p>
Port	<p>Type: <code>integer</code></p> <p>(Required if <code>EpFile</code> is set to null) Specify the socket server port. If you set this to <code>-1</code>, the adapter reads from the ephemeral port file, which is specified in the <code>EpFile</code> element. The default value is <code>12345</code>.</p>

XML Element	Description
EpFile	<p>Type: <code>string</code></p> <p>(Required if Host and Port are set to null) Specify the file that contains the socket server name/IP and port number. No default value.</p>
Retryperiod	<p>Type: <code>integer</code></p> <p>(Advanced) When the transporter is acting as a socket server, this element designates the length of time to wait for the first incoming connection before switching to the continuous state.</p> <p>When the transporter is acting as a socket client, this element designates the time period for attempting to re-establish an outgoing connection, in seconds. The default value is 0.</p>
BlockSize	<p>Type: <code>integer</code></p> <p>(Advanced) Define the size of the data block when transporting data from the socket server to the socket client. The default value is 1024.</p>
KeepAlive	<p>Type: <code>boolean</code></p> <p>(Advanced) If set to true, the adapter disconnects from the socket server if there are no data transports for the duration of time specified in your router configuration. For example, if you set your router configuration to two hours and there are no messages during that time, the adapter disconnects from the socket server.</p> <p>The default value is false.</p>

Socket Output Transporter Module Configuration

The Socket Output transporter obtains data from the previous module configured in the adapter configuration file, and outputs it using the socket interface. Set values for this transporter in the adapter configuration file.

XML Element	Description
Host	Type: <i>string</i> (Required if EpFile is set to null) If the transporter is acting as a socket client, specify the socket server name. If the transporter is acting as a socket server, do not set this element. No default value.
Port	Type: <i>integer</i> (Required if EpFile is set to null) Specify the socket server port. If you set this to -1, the adapter reads from the ephemeral port file, which is specified in the EpFile element. The default value is 12345.
EpFile	Type: <i>string</i> (Required if Host and Port are set to null) Specify the file that contains the socket server name/IP and port number. No default value.
Retryperiod	Type: <i>integer</i> (Advanced) When the transporter is acting as a socket server, this element designates the length of time to wait for the first incoming connection before switching to the continuous state. When the transporter is acting as a socket client, this element designates the time period for attempting to re-establish an outgoing connection, in seconds. The default value is 0.

XML Element	Description
KeepAlive	<p>Type: boolean</p> <p>(Advanced) If set to true, the adapter disconnects from the socket server if there are no data transports for the duration of time specified in your router configuration. For example, if you set your router configuration to two hours and there are no messages during that time, the adapter disconnects from the socket server.</p> <p>The default value is false.</p>

Building a Custom Transporter Module

Use the ESP adapter toolkit to build a custom transporter module to use within the adapter instance of your choice.

Prerequisites

(Optional) See the `$ESP_HOME/adapters/framework/examples/src` directory for source code for sample transporters.

Task

1. Create a class that extends the `com.sybase.esp.adapter.framework.module.Transporter` Java class that is included with the adapter toolkit.
2. Implement the **init()** function.

Prepare your input or output transporter module for the actions it is responsible for performing. For example, create a database connection or obtain properties from the adapter configuration file.
3. Implement the **start()** function.

Perform any necessary tasks when the adapter is started.
4. Implement the **execute()** function.

When the adapter framework calls this method, it is expected to run continuously until the adapter is requested to stop or until the adapter completes its work. Therefore, the code excerpt below might be found within a loop, or inside a callback method invoked by the transport when an event occurs, or inside a listener monitoring transport events.

`AepRecord` is a single record or row in ESP format and has an operation code that can be set. `AdapterRow` represents records as they flow from one module to the next. You can add multiple records as objects within a List of a single `AdapterRow` object. The `AdapterRow` has a timestamp, and block flags that control how its records are communicated to and from Event Stream Processor. See *Envelopes and Transactions* for additional details.

CHAPTER 2: Event Stream Processor Adapter Toolkit

The actions performed by this function depend on whether the transporter is input (datasource) or output (data sink). For example, for an input transporter that gets data from “myDataSource”, the **execute()** function might look like this:

```
public void execute() throws Exception {
    String value = myDataSource.getNextValue();

    AepRecord record = new AepRecord();
    record.getValues().add(value);
    AdapterRow row = utility.createRow(record);
    utility.sendRow(row);
}
```

For an output transporter that sends data to “myDataSink”, the **execute()** function might look like this:

```
public void execute() throws Exception {

    AdapterRow row = utility.getRow();
    if(row != null)
    {
        AepRecord record = (AepRecord) row.getData(0);
        if(record != null) {
            String value = record.getValues().toString();
            myDataSink.send(value);
        }
    }
}
```

The difference between input and output transporters is that input transporters call **utility.sendRow()** to send data to a formatter or ESP publisher, while output transporters call **utility.getRow()** to obtain data from a formatter or ESP subscriber.

For transporters that operate in streaming mode, call **utility.sendRowsBuffer()** (input) and **utility.getRowsBuffer()** (output).

See the `$ESP_HOME/adapters/framework/examples/src` directory for source code for sample transporters.

5. Implement the **stop()** function.

Perform any necessary tasks when the adapter is stopped.

6. Implement the **destroy()** function.

Perform any cleanup tasks for your input or output transporter.

7. (Optional) Call one of the following functions within the functions listed in the steps above:

- Call **utility.getParameters()** to get parameters that are defined in the adapter configuration file.
- Call **utility.sendRow()** to send data to the next module that is defined in the adapter configuration file.
- Call **utility.getRow()** to obtain data from the previous module that is defined in the adapter configuration file.

- Call `utility.isStopRequested()` to determine whether a stop command has been issued.
8. Register the implemented Java class to `$ESP_HOME/adapters/framework/config/modulesdefine.xml`. For example:

```
<TransporterDefn>
  <Name>MyOutputTransporter</Name>
  <Class>com.my.MyOutputTransporter</Class>
  <InputData>String</InputData>
</TransporterDefn>
```

9. Add the schema definitions for any unique parameters of the newly created module to the `$ESP_HOME/adapters/framework/config/parametersdefine.xsd` file.

If any of the parameters for the newly created module are the same as parameters for the standard transporter modules, you do not need to add schema definitions for these parameters.

10. Copy the `.jar` file containing the Java class you previously implemented and any other `.jar` files used by the custom adapter to `$ESP_HOME/adapters/framework/libj`.
11. (Optional) Start the adapter instance by issuing:

```
$ESP_HOME/adapters/framework/bin/start.bat <config file> or
$ESP_HOME/adapters/framework/bin/start.sh <config file>
```

where *<config file>* is the adapter configuration file in which you specified the adapter instance using the newly created transporter module.

12. (Optional) Stop the adapter instance by issuing:

```
$ESP_HOME/adapters/framework/bin/stop.bat <config file> or
$ESP_HOME/adapters/framework/bin/stop.sh <config file>
```

where *<config file>* is the adapter configuration file in which you specified the adapter instance using the newly created transporter module.

Refer to `$ESP_HOME/adapters/framework/examples` for additional details and transporter examples, as well as `$ESP_HOME/adapters/framework/examples/src` for the source code for these examples.

Next

Create an adapter configuration (`.xml`) file to define which adapter instance uses this newly created transporter module.

See also

- *Building a Custom Formatter Module* on page 58
- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Transporters Currently Available from SAP* on page 14

Formatter Modules

A formatter module converts between the data format of the transporter module and Event Stream Processor.

Event Stream Processor supports two types of formatters: row-based and stream-based formatters.

Row-based formatters obtain and output data in row format. They work with `AdapterRow` instances which are containers for one or more records or rows as they flow from one module (transporter, formatter, or ESP connector) to the next. You can add multiple records as objects within a `List` of a single `AdapterRow` object. The `AdapterRow` has a timestamp and block flags that control how its records are communicated to and from Event Stream Processor. See *Envelopes and Transactions* for additional details.

Stream-based formatters deal with streaming data. These formatters work with `ByteStream` instances which represent a continuous stream of data.

See also

- *Transporter Modules* on page 14
- *EspConnector Modules* on page 66
- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Create a Custom Adapter* on page 12
- *Debugging a Custom Adapter* on page 94

Formatters Currently Available from SAP

The adapter framework provides numerous formatter modules that come standard with Event Stream Processor. You can reuse these modules to create a custom adapter instance.

The `Input Datatype` column specifies the format of the incoming data, while the `Output Datatype` column specifies the format that the formatter translates this incoming data into.

`AdapterRow` is a container for one or more records or rows as they flow from one module (transporter, formatter, or ESP connector) to the next. You can add multiple records as objects within a `List` of a single `AdapterRow` object. The `AdapterRow` has a timestamp and block flags that control how its records are communicated to and from Event Stream Processor. See *Envelopes and Transactions* for additional details.

`AepRecord` is the class that represents stream records. This is the type that the ESP publisher expects as the data member in the `AdapterRow` instances that it receives from the previous module. This is also the type used in the `AdapterRow` instances that the ESP subscriber passes on to the next module.

Several of the formatters come with example data:

- **CSV to ESP Formatter**, see `$ESP_HOME/adapters/framework/instances/file_csv_input/data/input.csv`
- **JSON String to ESP Formatter**, see `$ESP_HOME/adapters/framework/instances/file_json_input/data/article_1.json`
- **XML String to ESP Formatter**, see `$ESP_HOME/adapters/framework/instances/file_xmllist_input/data/input.xml`
- **XMLDoc Stream to ESP Formatter**, see `$ESP_HOME/adapters/framework/instances/file_xmldoc_input/datetimeExample/data/data_1.xml`

Also, see `$ESP_HOME/adapters/framework/examples/stringlist_input` and `$ESP_HOME/adapters/framework/examples/stringlist_output` for examples containing the String List to ESP Formatter.

Name	Input Data-type	Mode (Streaming/ Row)	Output Data-type	Description
CSV to ESP Formatter	<code>Java.lang.String</code>	Row	<code>AepRecord</code>	Translates CSV string data to <code>AepRecord</code> objects.
ESP to CSV Formatter	<code>AepRecord</code>	Row	<code>Java.lang.String</code>	Translates <code>AepRecord</code> objects to CSV string data.
ESP to JSON Formatter	<code>AepRecord</code>	Row	<code>Java.lang.String</code>	Translates <code>AepRecord</code> objects to JSON string data.
ESP to Object List Formatter	<code>AepRecord</code>	Row	<code>Java.util.List<Java.lang.Object></code>	Converts <code>AepRecord</code> objects to Java object list.
ESP to String List Formatter	<code>AepRecord</code>	Row	<code>Java.util.List<Java.lang.String></code>	Converts <code>AepRecord</code> objects to string list.
ESP to XML String Formatter	<code>AepRecord</code>	Row	<code>Java.lang.String</code>	Translates <code>AepRecord</code> objects to ESP XML string.

Name	Input Data-type	Mode (Streaming/ Row)	Output Data-type	Description
ESP to XMLDoc String Formatter	AepRecord	Row	String	Translates AepRecord objects to XML format string according to the schema file configured in the adapter configuration file.
JSON String to ESP Formatter	Java.lang.String	Row	AepRecord	Translates JSON string to AepRecord objects.
JSON Stream to JSON String Formatter	InputStream	Streaming	Java.lang.String	Splits ByteStream data into standalone JSON message string data.
Object List to ESP Formatter	Java.util.List<Java.lang.Object>	Row	AepRecord	Converts Java object list to AepRecord objects.
Stream to String Formatter	InputStream	Streaming	Java.lang.String	Splits ByteStream data into strings according to the value specified in the delimiter property.
String to Stream Formatter	Java.lang.String	Streaming	OutputStream	Merges strings into ByteStream data.
String List to ESP Formatter	Java.util.List<Java.lang.String>	Row	AepRecord	Converts string list to AepRecord objects.
XML String to ESP Formatter	Java.lang.String	Row	AepRecord	Translates ESP XML strings to AepRecord objects.

Name	Input Data-type	Mode (Streaming/ Row)	Output Data-type	Description
XMLDoc Stream to ESP Formatter	Input-Stream	Streaming	AepRecord	Parses XML strings, extracts data according to the schema file configured in the adapter configuration file, and translates the data to AepRecord objects.

See also

- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Building a Custom Formatter Module* on page 58

CSV String to ESP Formatter Module Configuration

The CSV String to ESP formatter translates CSV strings to AepRecord objects. Set values for this formatter in the adapter configuration file.

This formatter is row-based and can connect two row-based transporters rather than stream-based transporters.

XML Element	Description
ExpectStreamNameOpcode	<p>Type: boolean</p> <p>(Required) If set to true, the adapter interprets the first two fields of the incoming CSV line as stream name and opcode respectively. The adapter discards messages with unmatched stream names.</p> <p>The accepted opcodes are:</p> <ul style="list-style-type: none"> • i or I: INSERT • d or D: DELETE • u or U: UPDATE • p or P: UPSERT • s or S: SAFEDELETE <p>The default value is false.</p>

XML Element	Description
Delimiter	Type: string (Advanced) The symbols used to separate the column. The default value is a comma (,).
HasHeader	Type: boolean (Advanced) Determines whether the first line of the file contains field descriptions. Default value is false.
DateFormat	Type: string (Advanced) The format string for parsing date values. For example, yyyy-MM-dd'T'HH:mm:ss.
TimestampFormat	Type: string (Advanced) Format string for parsing timestamp values. For example, yyyy-MM-dd'T'HH:mm:ss.SSS.

ESP to CSV String Formatter Module Configuration

The ESP to CSV String formatter translates AepRecord objects to CSV strings. Set values for this formatter in the adapter configuration file.

This formatter is row-based and can connect two row-based transporters rather than streaming-based transporters.

XML Element	Description
PrependStreamNameOpcode	Type: boolean (Optional) If set to true, the adapter prepends the stream name and the opcode in each row of generated data. The default value is false.
Delimiter	Type: string (Advanced) The symbols used to separate the column. The default value is a comma (,).
HasHeader	Type: boolean (Advanced) Determines whether the first line of the file contains the description of the fields. The default value is false.

XML Element	Description
DateFormat	Type: string (Advanced) The format string for date values. For example, yyyy-MM-dd 'T' HH:mm:ss.
TimestampFormat	Type: string (Advanced) Format string for timestamp values. For example, yyyy-MM-dd 'T' HH:mm:ss.SSS.

ESP to JSON Stream Formatter Module Configuration

The ESP to JSON Stream formatter translates AepRecord objects to JSON strings, and sends the JSON strings to next streaming output transporter that is configured in the adapter configuration file. Set values for this formatter in the adapter configuration file.

This formatter is streaming based and can connect two stream-based transporters rather than row-based transporters.

XML Element	Description
ColsMapping	(Required) Element containing the Column element.

XML Element	Description
<p>Column</p>	<p>Type: <code>complextype</code></p> <p>(Required) Specify JSONPath expressions for the JSON data that you want to map to columns of an ESP stream. You can have multiple Column elements.</p> <p>For example, if you had the following JSON data about a person,</p> <pre data-bbox="716 482 1180 991"> { "firstName": "John", "lastName": "Smith", "phoneNumbers": [{ "type": "home", "number": "212 555-1234" }, { "type": "fax", "number": "646 555-4567" }], "friends": [["female1", "female2", "female3"], ["male1", "male2", "male3"]] } </pre> <p>you could get the individual's first name by using the JSONPath expression <code>firstname</code>. If you want the first phone number, specify <code>phoneNumbers[0].number</code> as the JSONPath expression. Each JSONPath expression must represent one record.</p> <p>The first <code><Column/></code> element and its value are mapped to the first column of an ESP stream, the second <code><Column/></code> element and its value are mapped to the second column of an ESP stream, and so on.</p>
<p>DateFormat</p>	<p>Type: <code>string</code></p> <p>(Advanced) The format string for date values. For example, <code>yyyy-MM-dd 'T' HH:mm:ss</code>.</p>

XML Element	Description
TimestampFormat	Type: string (Advanced) Format string for timestamp values. For example, <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code> .

ESP to Object List Formatter Module Configuration

The ESP to Object List formatter converts an ESP row to an object list. Set values for this formatter in the adapter configuration file.

XML Element	Description
OutputAsSQLDatetimeFormat	Type: boolean (Optional) Specify whether the Event Stream Processor date, timestamp, and bigdatetime datatypes are output as <code>java.sql.Date</code> or <code>java.sql.Timestamp</code> . The default output is <code>java.util.Date</code> .

ESP to String List Formatter Module Configuration

The ESP to String List formatter converts an ESP row to a string list. Set values for this formatter in the adapter configuration file.

XML Element	Description
DateFormat	Type: string (Optional) Format string for date values from an ESP project. For example, <code>yyyy-MM-dd'T'HH:mm:ss</code> .
TimestampFormat	Type: string (Optional) Format string for timestamp values from an ESP project. For example, <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code> .

ESP to XML String Formatter Module Configuration

The ESP to XML String formatter translates AepRecord objects to ESP XML string. Set values for this formatter in the adapter configuration file.

This formatter is row-based and can connect two row-based transporters rather than stream-based transporters.

XML Element	Description
DateFormat	Type: string (Optional) Format string for date values from an ESP project. For example, <code>yyyy-MM-dd'T'HH:mm:ss</code> .
TimestampFormat	Type: string (Optional) Format string for timestamp values from an ESP project. For example, <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code> .

ESP to XMLDOC String Formatter Module Configuration

The ESP to XMLDOC String formatter translates AepRecord objects to XML format string according to the schema file specified in the adapter configuration file. Set values for this formatter in the adapter configuration file.

This formatter is stream-based and can connect two stream-based transporters rather than row-based transporters.

XML Element	Description
XMLSchemaFilePath	Type: string (Required) The path to the XML schema file that the XML output document builds against. No default value.
GlobalElementLocalName	Type: string (Required) Specify a global element to use as the root element in the generated XML document. No default value.
ColsMapping	(Required) Element containing the Column element.

XML Element	Description
Column	<p>Type: <code>string</code></p> <p>(Required) Specify which attributes or child elements, generated by the global element, to match by a pattern path expression and map to columns of an ESP row.</p> <p>For example, [<code><Column>XPath expression</Column></code>]+.</p> <p>The XPath expression is any valid XPath expression specified by an XPath specification. The first <code><Column/></code> element value is mapped to the first column of an ESP row, the second <code><Column/></code> element value is mapped to the second column of an ESP row, and so on.</p>

JSON String to ESP Formatter Module Configuration

The JSON String to ESP formatter translates JSON strings to AepRecord objects. Set values for this formatter in the adapter configuration file.

This formatter is row based and can connect two row based transporters rather than streaming based transporters.

XML Element	Description
ColumnMappings	(Required) Element containing the ColsMapping element.
ColsMapping	<p>Type: <code>complexType</code></p> <p>(Required) Element which contains the Column element. You can have multiple ColsMapping elements if you are using an <code>ESPMultiStreamPublisher</code>.</p> <p>This element has two attributes:</p> <ul style="list-style-type: none"> • streamname – the ESP stream to which you want to publish JSON data. • rootpath – a root path for the JSON data. This must be an array. See the Column element for an example of a root path value.

XML Element	Description
<p>Column</p>	<p>Type: string</p> <p>(Required) Specify a JSONPath expression for the JSON data that you want to map to columns of an ESP stream. This expression is matched to the value specified in the rootpath attribute of the ColsMapping element, if applicable. You can have multiple Column elements.</p> <p>There are two types of JSON data: array or object. For example, if you had the following JSON data about a person,</p> <pre data-bbox="716 578 1177 1086"> { "firstName": "John", "lastName": "Smith", "phoneNumbers": [{ "type": "home", "number": "212 555-1234" }, { "type": "fax", "number": "646 555-4567" }], "friends": [["female1", "female2", "female3"], ["male1", "male2", "male3"]] } </pre> <p>you could get the individual's first name by using the JSONPath expression <code>firstName</code>. If you want the first phone number, specify <code>phoneNumbers[0].number</code> as the JSONPath expression. You would not have to specify a rootpath value.</p> <p>If you want the numbers and types of phone numbers, specify <code>phoneNumbers</code> as the rootpath value and <code>numbers</code> and <code>type</code> as the JSONPath expressions in the Column element.</p> <p>You can also specify <code>*</code> for Column which indicates you want all the data in an array (that does not have key value data within it). For example, if</p>

XML Element	Description
	<p>you wanted all the female friends data, specify friends[1] for rootpath and * for Column.</p> <p>The first <Column/> element and its value are mapped to the first column of an ESP stream, the second <Column/> element and its value are mapped to the second column of an ESP stream, and so on.</p>
DateFormat	<p>Type: string</p> <p>(Advanced) The format string for parsing date values. For example, yyyy-MM-dd'T'HH:mm:ss.</p>
TimestampFormat	<p>Type: string</p> <p>(Advanced) Format string for parsing timestamp values. For example, yyyy-MM-dd'T'HH:mm:ss.SSS.</p>

JSON Stream to JSON String Formatter Module Configuration

The JSON Stream to JSON String formatter reads data from `InputStream`, splits it into standalone JSON message strings, and sends these message strings to the next module that is configured in the adapter configuration file. Set values for this formatter in the adapter configuration file.

XML Element	Description
CharsetName	<p>Type: string</p> <p>(Optional) Specify the name of a supported charset. The default value is US-ASCII.</p>

Object List to ESP Formatter Module Configuration

The Object List to ESP formatter converts an object list to an ESP row. Set values for this formatter in the adapter configuration file.

XML Element	Description
DateFormat	<p>Type: string</p> <p>(Optional) Format string for parsing date values. For example, yyyy-MM-dd'T'HH:mm:ss.</p>

XML Element	Description
TimestampFormat	Type: <code>string</code> (Optional) Format string for parsing timestamp values. For example, <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code> .

Stream to String Formatter Module Configuration

The Stream to String formatter reads streaming data from an input stream, and splits it into Java strings. Set values for this formatter in the adapter configuration file.

XML Element	Description
Delimiter	Type: <code>string</code> (Required) Specify the symbol that separates columns. The default value is <code>"\n"</code> .
IncludeDelimiter	Type: <code>boolean</code> (Required) If set to true, the delimiter is part of current row. If set to false, the delimiter is not part of the current row. The default value is false.
AppendString	Type: <code>string</code> (Required if IncludeDelimiter is set to true) If set to true, specify the string to append to the end of each result row. No default value.
AppendPosition	Type: <code>string</code> (Required if IncludeDelimiter is set to true) Specify the position at which the AppendString element takes effect. Valid values: <code>front</code> and <code>end</code> . The default value is <code>front</code> .
IgnoreSpace	Type: <code>boolean</code> (Required) Specify whether to trim the space char. The default value is true.
CharsetName	Type: <code>string</code> (Advanced) Specify the name of a supported charset. The default value is <code>US-ASCII</code> .

String to Stream Formatter Module Configuration

The String to Stream formatter writes Java strings to output streams. Set values for this formatter in the adapter configuration file.

XML Element	Description
Delimiter	Type: <code>string</code> (Required) Specify the symbol that separates columns. The default value is <code>"\n"</code> .
IncludeDelimiter	Type: <code>boolean</code> (Required) If set to true, the delimiter is part of current row. If set to false, the delimiter is not part of the current row. The default value is false.
AppendString	Type: <code>string</code> (Required if IncludeDelimiter is set to true) If set to true, specify the string to append to the end of each result row. No default value.
AppendPosition	Type: <code>string</code> (Required if IncludeDelimiter is set to true) Specify the position to which the AppendString element takes effect. There are two valid values: front and end. The default value is front.
IgnoreSpace	Type: <code>boolean</code> (Required) Specify whether to ignore the space char. The default value is false.
CharsetName	Type: <code>string</code> (Advanced) Specify the name of a supported charset. The default value is US-ASCII.

String List to ESP Formatter Module Configuration

The String List to ESP formatter converts a string list to an ESP row. Set values for this formatter in the adapter configuration file.

XML Element	Description
DateFormat	Type: string (Optional) Format string for parsing date values. For example, yyyy-MM-dd'T'HH:mm:ss.
TimestampFormat	Type: string (Optional) Format string for parsing timestamp values. For example, yyyy-MM-dd'T'HH:mm:ss.SSS.

XML String to ESP Formatter Module Configuration

The XML String to ESP formatter translates ESP XML strings to AepRecord objects. Set values for this formatter in the adapter configuration file.

This formatter is row based and can connect two row based transporters rather than streaming based transporters.

XML Element	Description
DateFormat	Type: string (Optional) Format string for parsing date values. For example, yyyy-MM-dd'T'HH:mm:ss.
TimestampFormat	Type: string (Optional) Format string for parsing timestamp values. For example, yyyy-MM-dd'T'HH:mm:ss.SSS.

XMLDoc Stream to ESP Formatter Module Configuration

The XMLDoc Stream to ESP formatter parses XML format strings, extracts data according to the schema file specified in the adapter configuration file, and translates this data to AepRecord objects. Set values for this formatter in the adapter configuration file.

This formatter is streaming based and can connect two streaming based transporters rather than row based transporters.

XML Element	Description
XmlElemMappingRowPattern	<p>Type: <code>string</code></p> <p>(Required) Specify a pattern to indicate which XML elements in the XML doc are processed by the formatter. The matched elements are mapped to ESP rows whose attributes and child elements are mapped as columns of an ESP row. The adapter ignores any XML elements that do not match this pattern.</p> <p>This pattern is a subset of XPath expressions. The <code>[/?NCName[NCName]*</code> path expression is the only supported expression, where NCName (Non-Colonized Name) is the local name element without a prefix or namespace.</p> <p>If the elements in the path expression include a namespace URI (prefix), all of these elements belong to the same namespace. Provide the namespace in the XmlElemNamespace element.</p> <p>Here are some examples of valid path expressions:</p> <ul style="list-style-type: none"> • <code>/RootElement</code> • <code>ParentElement</code> • <code>ParentElement/ChildElement</code> • <code>/RootElement/ParentElement</code>
XmlElemNamespace	<p>Type: <code>string</code></p> <p>(Required) Specify the namespace URI for elements that appear in the pattern path expression.</p>
ColsMapping	<p>(Required) Element containing the Column element.</p>

XML Element	Description
Column	<p>Type: <code>string</code></p> <p>(Required) Specify which attributes or child elements of the XML elements, which are matched by pattern path expression, to map to columns of the ESP row. For example, [<code><Column>XPath expression</Column></code>]+.</p> <p>The XPath expression is any valid XPath expression specified by an XPath specification. The XPath expression can begin only from the last XML element that appears in the path pattern expression or its decedent elements.</p> <p>The first <code><Column/></code> element value is mapped to the first column of an ESP row, the second <code><Column/></code> element value is mapped to the second column of an ESP row, and so on.</p>
DateFormat	<p>Type: <code>string</code></p> <p>(Optional) Format string for parsing date values. For example, <code>yyyy-MM-dd'T'HH:mm:ss</code>.</p>
TimestampFormat	<p>Type: <code>string</code></p> <p>(Optional) Format string for parsing timestamp values. For example, <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code>.</p>

Datatype Mapping for Formatters

Mapping information for ESP to Java datatypes and Java to ESP datatypes.

Table 5. ESP to Java Objects Datatype Mappings

ESP Datatype	Java Datatype
<code>string</code>	<code>java.lang.String</code>
<code>boolean</code>	<code>java.lang.Boolean</code>
<code>integer</code>	<code>java.lang.Integer</code>
<code>long or interval</code>	<code>java.lang.Long</code>

ESP Datatype	Java Datatype
double	java.lang.Double
timestamp	java.util.Date or java.sql.Timestamp (if OutputAsSQL-DatetimeFormat is set to true)
date	java.util.Date or java.sql.Timestamp (if OutputAsSQL-DatetimeFormat is set to true)
bigdatetime	java.lang.String (the output format is yyy-MM-ddTHH:mm:ss:UUUUUU where U stands for microseconds and the timezone is in UTC) or java.sql.Timestamp (if OutputAsSQL-DatetimeFormat is set to true)
time	java.lang.String (the output format is HH:mm:ss) java.sql.Time (if OutputAsSQL-DatetimeFormat is set to true)
fixeddecimal	java.math.BigDecimal
money, money01 - money15	java.math.BigDecimal (it has the same precision as corresponding money (xx) object)
binary	byte[]

Table 6. Java Objects to ESP Datatype Mappings

Java Datatype	ESP Datatype
Java.util.Date (including its child classes)	bigdatetime or time
byte[]	binary
Java.lang.Object (including its child classes)	string
Any Java classes with the toString() method, and the result string is [true,false]	boolean

Java Datatype	ESP Datatype
Any Java classes with the toString() method, and the result string can be used in Integer.valueOf(result) .	<code>integer</code>
Any Java classes with the toString() method, and the result string can be used in Long.valueOf(result) .	<code>long</code> or <code>interval</code>
Any Java classes with the toString() method, and the result string can be used in Double.valueOf(result) .	<code>double</code>
Any Java classes with the toString() method. The result string consists of the decimal digits except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value, or an ASCII plus sign '+' ('\u002B') to indicate a positive value, or the period may occur in the middle of the string. The part after the period is the implied precision, and the precision matches the ESP datatype.	<code>money</code> , <code>money01</code> - <code>money15</code>
Any Java classes with the toString() method, and the result string represents a fixed decimal in the standard ESP format.	<code>fixeddecimal</code>

Building a Custom Formatter Module

Use the ESP adapter toolkit to build a custom formatter module to use within the adapter instance of your choice.

Prerequisites

(Optional) See the `$ESP_HOME/adapters/framework/examples/src` directory for source code for sample formatters.

Task

1. Create a class that extends one of these Java classes:
 - (Row-based formatter) `com.sybase.esp.adapter.framework.module.RowFormatter`
 - (Streaming-based formatter) `com.sybase.esp.adapter.framework.module.StreamingFormatter`

Make row-based formatters a subclass of `RowFormatter`, and stream-based formatters a subclass of `StreamingFormatter`. Use row-based formatters with row-based transporters, and stream-based formatters with stream-based transporters.

2. For row-based formatters, implement these functions:

a) The **init()** function.

Prepare your formatter module to convert between data formats; for example, obtain properties from the adapter configuration file and perform any required initialization tasks.

b) The **destroy()** function.

Perform cleanup actions for your formatter.

c) The **convert()** function.

Here is a simple example of a **convert()** function that converts Java objects to strings:

```
public AdapterRow convert(AdapterRow in) throws Exception {
    Object obj = in.getData(0);
    in.setData(0, obj.toString());
    return in;
}
```

3. For stream-based formatters, implement these functions:

a) The **init()** function.

Prepare your formatter module to convert between data formats; for example, obtain properties from the adapter configuration file and perform any required initialization tasks.

b) The **start()** function.

Perform any necessary tasks when the adapter is started.

c) The **execute()** function.

Here is an example of the **execute()** function for a formatter that converts row-based data into stream-based:

```
public void execute() throws Exception {
    OutputStream output = utility.getOutputStream();
    while(!utility.isStopRequested())
    {
        AdapterRow row = utility.getRow();
        if(row != null)
        {
            AepRecord record = (AepRecord)row.getData(0);
            String str = record.getValues().toString() + "\n";
            output.write(str.getBytes());
        }
    }
}
```

For a formatter that converts from stream-based data into row-based, use:

- **utility.getInputStream()** to obtain the `InputStream`
- **utility.createRow()** to create the `AdapterRow` objects

- **utility.sendRow()** to send the rows to the next module specified in the adapter configuration file
 - d) The **stop()** function.
Perform any necessary tasks when the adapter is stopped.
 - e) The **destroy()** function.
Perform clean up actions for your formatter.
4. (Optional) Call one of the following functions within the functions listed in the steps above:
- Call **utility.getParameters()** to get parameters that are defined in the adapter configuration file.
 - Call **utility.sendRow()** to send data to the next module defined in the adapter configuration file.
 - Call **utility.getRow()** to obtain data from the previous module defined in the adapter configuration file.
 - Call **utility.isStopRequested()** to determine whether a stop command has been issued.

5. Register the implemented Java class to `$ESP_HOME/adapters/framework/config/modulesdefine.xml`. For example:

```
<FormatterDefn>
<Name>SampleFormatter</Name>
<Class>com.sybase.esp.adapter.formatters.SampleFormatter</Class>
<InputData>String</InputData>
<OutputData>ESP</OutputData>
<ParametersNodeName>SampleFormatterParameters</
ParametersNodeName>
</FormatterDefn>
```

where `<ParametersNodeName>` is the optional node that represents the formatter subnode name in the adapter configuration file.

6. Add the schema definitions for any unique parameters of the newly created module to the `$ESP_HOME/adapters/framework/config/parametersdefine.xsd` file.

If any of the parameters for the newly created module are the same as parameters for the standard formatter modules, you need not to add schema definitions for these parameters.

7. Copy the .jar file containing the class you previously implemented to `$ESP_HOME/adapters/framework/libj`.

8. (Optional) Start the adapter instance by issuing:

```
$ESP_HOME/adapters/framework/bin/start.bat <config file> or
$ESP_HOME/adapters/framework/bin/start.sh <config file>
```

where *<config file>* is the adapter configuration file in which you specified the adapter instance using the newly created formatter module.

9. (Optional) Stop the adapter instance by issuing:

```
$ESP_HOME/adapters/framework/bin/stop.bat <config file> or
$ESP_HOME/adapters/framework/bin/stop.sh <config file>
```

where *<config file>* is the adapter configuration file in which you specified the adapter instance using the newly created formatter module.

Refer to `$ESP_HOME/adapters/framework/examples` for additional details and formatter examples, as well as `$ESP_HOME/adapters/framework/examples/src` for the source code for these examples.

Next

Create an adapter configuration (.xml) file to define which adapter instance uses this newly created formatter module

See also

- *Building a Custom Transporter Module* on page 37
- *Enabling Guaranteed Delivery for an Input Transporter* on page 64
- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Formatters Currently Available from SAP* on page 40

Batch Processing

Details about how to control the manner in which AdapterRow instances are sent and processed by Event Stream Processor.

Sending individual AdapterRow instances to Event Stream Processor results in minimal latency, but can reduce overall throughput due to network overhead. Sending AdapterRow instances using batch processing or blocks can improve overall throughput with some reduction in latency. See *Batch Processing* in the *Programmers Guide*.

When not part of a block, AdapterRow instances are published individually over the network to ESP when the **sendRow()** or **sendRowData()** method is invoked.

You can demarcate a block explicitly within the source code, or implicitly using the adapter configuration file. A block is demarcated explicitly when either a BATCH_START flag (sends the block using envelopes) or a TRANS_START flag (sends the block using transactions) is set in the current or preceding AdapterRow instances. The end of a block is demarcated by an AdapterRow instance with a BLOCK_END flag, or when an AdapterRow is sent to ESP with the **sendEnd()** method instead of **sendRow()** or **sendRowData()**.

If the AdapterRow instance is not explicitly part of a block, and the optional EspPublisher module property **MaxPubPoolSize** is set to a value greater than 1, the adapter framework automatically uses blocks to transmit the records. The **MaxPubPoolSize** parameter specifies the maximum size of the record pool before all records in the pool are published.

If the optional `EspPublisher` module property **MaxPubPoolTime** is configured, this also causes publishing of the record pool in blocks. **MaxPubPoolTime** specifies the maximum period of time, in milliseconds, for which records are pooled before being published.

If the threshold value of either **MaxPubPoolSize** or **MaxPubPoolTime** is reached, the record pool is published using blocks. A third optional configuration property, **UseTransactions**, controls whether the blocks are published using envelopes or transactions.

If event blocks are neither demarcated explicitly in the adapter code, nor implicitly used based on the adapter configuration, records are published individually.

A typical transaction block using `AdapterRow` might look like this:

1. An `AdapterRow` with a block flag that is set to `TRANS_START`
2. Various `AdapterRows` with block flags that are set to `BLOCK_DATA`
3. An `AdapterRow` with a block flag set to `BLOCK_END`

For `AdapterRow` instances with no data and only the start or end flag, set each data position in the instance to null. See the `$ESP_HOME/adapters/framework/examples/src/com/sybase/esp/adapter/framework/examplemodules/ExampleRowInputTransporter.java` example for publishing `AdapterRow` instances in various ways.

An `AdapterRow` instance cannot contain records from multiple transactions. If using blocks, records from an `AdapterRow` instance are published in one transaction or envelope.

See also

- *EspPublisher Configuration* on page 69

Schema Discovery

Discover external schemas and create CCL schemas, streams, or windows based on the format of the data from the datasource that is connected to an adapter.

Every row in a stream or window must have the same structure, or schema, which includes the column names, the column datatypes, and the order in which the columns appear. Multiple streams or windows can use the same schema, but each stream or window can only have one schema.

Rather than manually creating a new schema in your ESP project, you can use schema discovery to discover and automatically create a schema, stream, or window based on the format of the data from the datasource to which your adapter connects. For example, you create a table in your SAP HANA database and use the SAP HANA Output adapter to connect to the database. You can then use schema discovery to discover and create a schema, stream, or window in your ESP project that corresponds to the schema of the table you created in your SAP HANA database.

While using discovery is a convenient way to create your CCL schema, pay particular attention to the data types that the CCL columns inherit from the external datasource. For

example, whenever possible, discovery maintains the same level of precision or greater when mapping source data types to ESP data types. Some databases, such as SAP Sybase IQ, support microsecond precision for the `SQL_TIMESTAMP` and `SQL_TYPE_TIMESTAMP` data types. As such, schema discovery maps these types to the ESP data type `bigdatetime`, which also supports microsecond precision. If your ESP project does not require this level of precision, you can, after generating your schema through discovery, modify the schema to use a lower-precision data type, such as `timestamp` (millisecond precision).

To discover a schema, first configure the adapter properties. Each adapter that supports schema discovery has unique properties that must be set to enable schema discovery.

Implementing Schema Discovery in a Custom Adapter

(Optional) Use interfaces and functions from the adapter toolkit to implement schema discovery in a transporter and formatter module. There are two types of schema discovery: nonsampling and sampling. Use non-sampling schema discovery when the transporter can fully determine schema on its own. Use sampling schema discovery when the transporter cannot determine the schema and passes this data to the formatter to generate the schema.

1. Add the `x_winCmdDisc` (Windows) or `x_unixCmdDisc` (UNIX) parameter to the `cnxml` file for your custom adapter.

See `$ESP_HOME/adapters/framework/examples/discover/ExampleAdapterForDiscovery.cnxml` for an example of a `cnxml` file with the discovery command.

2. Implement schema discovery in your custom modules:

- (For transporter modules only) To implement non-sampling schema discovery, implement the `com.sybase.esp.adapter.framework.discovery.TableDiscovery` and `com.sybase.esp.adapter.framework.discovery.ColumnDiscovery` interfaces. For an example of an adapter with non-sampling schema discovery, see `$ESP_HOME/adapters/framework/examples/discover`. For the source code of a discoverable transporter module, see `$ESP_HOME/adapters/framework/examples/src/com/Sybase/esp/adapter/framework/examplemodules/ExampleDiscoverableInputTransporter.java`.
- (For input adapters only) To implement sampling schema discovery:
 - a. For the transporter module, implement the `com.sybase.esp.adapter.framework.discovery.TableDiscoveryWithSample` interface.
 - b. For the formatter module, implement the `com.sybase.esp.adapter.framework.discovery.ColumnDiscovery` interface.

See also

- *Enabling Guaranteed Delivery for an Input Transporter* on page 64
- *Configuring a New Adapter* on page 75

Guaranteed Delivery

Guaranteed delivery (GD) is a delivery mechanism that guarantees data is processed from a stream to an adapter.

GD ensures that data continues to be processed when:

- The ESP Server fails.
- The destination (third-party server) fails.
- The destination (third-party server) does not respond for a period of time.

Input adapters support GD using facilities provided by the external datasource to which the input transporter connects.

Enabling Guaranteed Delivery for an Input Transporter

(Optional) Enable guaranteed delivery (GD) in a custom input transporter by implementing the `com.sybase.esp.adapter.framework.event.AdapterRowEventListener` interface, registering the `GdAdapterEventListener` class, and adding and setting the `<GDMode>` parameter to true for the `EspPublisher` or `EspMultistreamPublisher`.

Prerequisites

Create a custom input transporter module.

Task

1. In the adapter configuration file, add the `<GDMode>` parameter and set it to true for the `EspPublisher` or `EspMultiStreamPublisher`:

- For `EspPublisher`:

```
<EspPublisherParameters>
  <ProjectName>EspProject1</ProjectName>
  <StreamName>MyInStream</StreamName>
  <GDMode>true</GDMode>
</EspPublisherParameters>
```

- For `EspMultiStreamPublisher`:

```
<EspMultiStreamPublisherParameters>
  <Streams>
    <Stream>
      <ProjectName>EspProject1</ProjectName>
      <StreamName>MyInStream1</StreamName>
      <GDMode>true</GDMode>
    </Stream>
  </Streams>
</EspMultiStreamPublisherParameters>
```


2. Implement the `com.sybase.esp.adapter.framework.event.AdapterRowEventListener` interface. For example,

```
public class GdAdapterEventListener implements
AdapterRowEventListener
{
    public void adapterEventPerformed(AdapterRowEvent event) {
        List<AdapterRow> rows = event.getAdapterRows();
        switch(event.getType())
        {
            case PUBLISH_SUCCESS:
                processPublishSuccess();
                break;
            case PUBLISH_FAILURE:
                processPublishFailure();
                break;
            case FORMAT_FAILURE:
                processFormatFailure();
                break;
            default:
                break;
        }
    }
}
```

3. Create the class `GdAdapterEventListener` and register it to enable GD in the input transporter when it starts. For example,

```
GdAdapterEventListener gdProcessor = new
GdAdapterEventListener();

utility.getAdapterUtility().registerRowEventListener(gdProcessor,
EventType.PUBLISH_SUCCESS);
utility.getAdapterUtility().registerRowEventListener(gdProcessor,
EventType.PUBLISH_FAILURE);
utility.getAdapterUtility().registerRowEventListener(gdProcessor,
EventType.FORMAT_FAILURE);
```

4. Keep track of the last row successfully published to ESP, either by using your external datasource or the transporter itself.

If you are using the input transporter, you can get the last row successfully published from the `PUBLISH_SUCCESS` message. If you are publishing in transactions or envelopes, the `PUBLISH_SUCCESS` message contains all rows in the last transaction or envelope to be successfully published. If you publish in single rows, the message contains the last single row that was successfully published.

See also

- *Building a Custom Formatter Module* on page 58
- *Implementing Schema Discovery in a Custom Adapter* on page 63

EspConnector Modules

The EspConnector modules are responsible for connecting to Event Stream Processor. Connector module types include: EspSubscriber, EspMultiStreamSubscriber, EspPublisher, and EspMultiStreamPublisher.

EspSubscriber subscribes to a stream in an ESP project and outputs data to the next module configured in the adapter configuration file (for example, a formatter or transporter).

EspMultiStreamSubscriber has the same functionality but can subscribe to multiple streams.

EspPublisher takes data from a transporter module and publishes it to a stream in an ESP project. EspMultiStreamPublisher has the same functionality but can publish data to multiple streams.

See also

- *Formatter Modules* on page 40
- *Transporter Modules* on page 14
- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Create a Custom Adapter* on page 12
- *Debugging a Custom Adapter* on page 94

EspSubscriber Configuration

The ESP Subscriber module obtains data from an ESP project and passes it along to a transporter or formatter module. Specify values for the EspSubscriber module in the adapter configuration file. Specify this module for an output adapter only.

XML Element	Description
ProjectName	Type: string (Required if running adapter in standalone mode; optional if running in managed mode) Specifies the unique project tag of the ESP project to which the adapter is connected. For example, EspProject2. This is the same project tag that you specify later in the adapter configuration file in the Name element within the Event Stream Processor (EspProjects) element. If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this element as the adapter automatically detects the project name.

XML Element	Description
StreamName	<p>Type: <code>string</code></p> <p>(Required if running adapter in standalone mode; optional if running in managed mode) Name of the ESP stream from which the adapter subscribes to data.</p> <p>If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this element as the adapter automatically detects the stream name.</p>

See also

- *EspMultiStreamSubscriber Configuration* on page 67
- *EspPublisher Configuration* on page 69
- *EspMultiStreamPublisher Configuration* on page 71
- *Event Stream Processor Configuration* on page 73
- *Configuring a New Adapter* on page 75

EspMultiStreamSubscriber Configuration

Specify values for the `EspMultiStreamSubscriber` module in the adapter configuration file. This module is specified only for an output adapter.

XML Element	Description
Streams	(Required) Element containing the Stream , ProjectName , StreamName , and ColumnMapping elements.
Stream	(Required) Element containing details for the target project and streams to which the adapter is connected. Contains the ProjectName , StreamName , and ColumnMapping elements. You can specify multiple Stream elements.

XML Element	Description
ProjectName	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if running in managed mode) The name of the ESP project to which the adapter belongs. The same project as specified in the Name element of the ESP projects section of the adapter configuration file.</p> <p>If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you need not set this property as the adapter automatically connects to the project that started it.</p>
StreamName	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if running in managed mode) The name of the source stream to which the adapter connects.</p> <p>If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you need not set this property as the adapter automatically connects to the stream to which it is attached within the project.</p>
ColumnMapping	<p>Type: <code>string</code></p> <p>(Optional) The column list of the source stream to which the adapter connects. Separate the columns with a space char. This element contains the "enumtype" attribute which has these two valid values:</p> <ul style="list-style-type: none"> • index – the index of the column. • name – the column name. <p>The default value is name.</p>

See also

- *EspSubscriber Configuration* on page 66
- *EspPublisher Configuration* on page 69
- *EspMultiStreamPublisher Configuration* on page 71
- *Event Stream Processor Configuration* on page 73

- *Configuring a New Adapter* on page 75

EspPublisher Configuration

The ESP Publisher module obtains data from a transporter or formatter module and publishes it to an ESP project. Specify values for the EspPublisher module in the adapter configuration file. Specify this module for the input adapter only.

XML Element	Description
<p>ProjectName</p>	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if it is running in managed mode) Name of the ESP project to which the adapter is connected. For example, <code>EspProject2</code>.</p> <p>This is the same project tag that you specify later in the adapter configuration file in the Name element within the Event Stream Processor (Es-pProjects) element.</p> <p>If you are starting the adapter with the ESP project to which it is attached (that is, running the adapter in managed mode), you need not set this element as the adapter automatically detects the project name.</p>
<p>StreamName</p>	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if it is running in managed mode) Name of the ESP stream to which the adapter publishes data.</p> <p>If you are starting the adapter with the ESP project to which it is attached (that is, running the adapter in managed mode), you need not set this element as the adapter automatically detects the stream name.</p>

XML Element	Description
MaxPubPoolSize	<p>Type: <code>positive integer</code></p> <p>(Optional) Maximum size of the record pool. Record pooling, also referred to as block or batch publishing, allows for faster publication since there is less overall resource cost in publishing multiple records together, compared to publishing records individually.</p> <p>Record pooling is disabled if this value is 1. The default value is 256.</p>
MaxPubPoolTime	<p>Type: <code>positive integer</code></p> <p>(Optional) Maximum period of time, in milliseconds, for which records are pooled before being published. If not set, pooling time is unlimited and the pooling strategy is governed by maxPubPoolSize. No default value.</p>
UseTransactions	<p>Type: <code>boolean</code></p> <p>(Optional) If set to true, pooled messages are published to Event Stream Processor in transactions. If set to false, they are published in envelopes. The default value is false.</p>
SafeOps	<p>Type: <code>boolean</code></p> <p>(Advanced) Converts the opcodes INSERT and UPDATE to UPSERT, and converts DELETE to SAFEDELETE. The default value is false.</p>
SkipDels	<p>Type: <code>boolean</code></p> <p>(Advanced) Skips the rows with opcodes DELETE or SAFEDELETE. The default value is false.</p>
GDMode	<p>Type: <code>boolean</code></p> <p>(Optional) If set to true, the adapter runs in guaranteed delivery (GD) mode. See <i>Enabling Guaranteed Delivery for an Input Transporter</i> for additional details on enabling GD.</p> <p>Default value is false.</p>

See also

- *EspSubscriber Configuration* on page 66
- *EspMultiStreamSubscriber Configuration* on page 67
- *EspMultiStreamPublisher Configuration* on page 71
- *Event Stream Processor Configuration* on page 73
- *Configuring a New Adapter* on page 75
- *Batch Processing* on page 61

EspMultiStreamPublisher Configuration

Specify values for the ESP MultiStream Publisher module in the adapter configuration file. This module is specified only for an input adapter.

XML Element	Description
Streams	(Required) Element containing the Stream , ProjectName , StreamName , and ColumnMapping elements.
Stream	(Required) Element containing details for the target project and streams to which the adapter is connected. Contains the ProjectName , StreamName , and ColumnMapping elements. You can specify multiple Stream elements.
Filter	(Optional) Element containing the MatchString element.
MatchString	(Optional) Filters records with one or more column values. Contains a value attribute for specifying the value you want to filter by. No default value.

XML Element	Description
ProjectName	<p>Type: <code>string</code></p> <p>(Required if running adapter in standalone mode; optional if running in managed mode) The name of the ESP project to which the adapter belongs. This is the same project as specified in the Name element of the ESP projects section of the adapter configuration file.</p> <p>If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this property as the adapter automatically connects to the project that started it.</p>
StreamName	<p>Type: <code>string</code></p> <p>(Required if running adapter in standalone mode; optional if running in managed mode) The name of the target stream to which the adapter connects.</p> <p>If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this property as the adapter automatically connects to the stream to which it is attached within the project.</p>
ColumnMapping	<p>Type: <code>string</code></p> <p>(Optional) The column index list in the source row to be published to the target stream. Separate the columns using space char. If you do not set this element, all columns are published to the target stream.</p>
SafeOps	<p>Type: <code>boolean</code></p> <p>(Advanced) Converts the opcodes INSERT and UPDATE to UPSERT, and converts DELETE to SAFEDELETE. The default value is false.</p>
SkipDels	<p>Type: <code>boolean</code></p> <p>(Advanced) Skips the rows with opcodes DELETE or SAFEDELETE. The default value is false.</p>

See also

- *EspSubscriber Configuration* on page 66
- *EspMultiStreamSubscriber Configuration* on page 67
- *EspPublisher Configuration* on page 69
- *Event Stream Processor Configuration* on page 73
- *Configuring a New Adapter* on page 75

Event Stream Processor Configuration

Use Event Stream Processor elements, which are defined in the adapter configuration file, to configure communication between Event Stream Processor and the adapter instance.

XML Element	Description
EspProjects	(Required) Element containing elements for connecting to Event Stream Processor.
EspProject	(Required) Element containing the Name and Uri elements. Specifies information for the ESP project to which the adapter is connected.
Name	Type: string (Required) Specifies the unique project tag of the ESP project which the EspConnector (publisher/subscriber) module references.
Uri	Type: string (Required) Specifies the total project URI to connect to the ESP project. For example, <code>esp://localhost:19011/ws1/p1</code> .
Security	(Required) Element containing all the authentication elements below. Specifies details for the authentication method used for Event Stream Processor.
User	Type: string (Required) Specifies the user name required to log in to Event Stream Processor (see AuthType). No default value.

XML Element	Description
Password	<p>Type: <code>string</code></p> <p>(Required) Specifies the password required to log in to Event Stream Processor (see espAuthType).</p> <p>Includes an "encrypted" attribute indicating whether the Password value is encrypted. The default value is false. If set to true, the password value is decrypted using RSAKeyStore and RSAKeyStorePassword.</p>
AuthType	<p>Type: <code>string</code></p> <p>(Required) Method used to authenticate to the Event Stream Processor. Valid values are:</p> <ul style="list-style-type: none"> • server_rsa – RSA authentication using key-store • kerberos – Kerberos authentication using ticket-based authentication • user_password – LDAP, SAP BI, and Native OS (user name/password) authentication <p>If the adapter is operated as a Studio plug-in, AuthType is overridden by the Authentication Mode Studio start-up parameter.</p>
RSAKeyStore	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies the location of the RSA keystore, and decrypts the password value. Required if AuthType is set to <code>server_rsa</code>, or the encrypted attribute for Password is set to true, or both.</p>
RSAKeyStorePassword	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies the keystore password, and decrypts the password value. Required if AuthType is set to <code>server_rsa</code>, or the encrypted attribute for Password is set to true, or both.</p>
KerberosKDC	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies host name of Kerberos key distribution center. Required if AuthType is set to <code>kerberos</code>.</p>

XML Element	Description
KerberosRealm	Type: <code>string</code> (Dependent required) Specifies the Kerberos realm setting. Required if AuthType is set to <code>kerberos</code> .
KerberosService	Type: <code>string</code> (Dependent required) Specifies the Kerberos principal name that identifies an Event Stream Processor cluster. Required if AuthType is set to <code>kerberos</code> .
KerberosTicketCache	Type: <code>string</code> (Dependent required) Specifies the location of the Kerberos ticket cache file. Required if AuthType is set to <code>kerberos</code> .
EncryptionAlgorithm	Type: <code>string</code> (Optional) Used when the encrypted attribute for Password is set to <code>true</code> . If left blank, RSA is used as default.

See also

- *EspSubscriber Configuration* on page 66
- *EspMultiStreamSubscriber Configuration* on page 67
- *EspPublisher Configuration* on page 69
- *EspMultiStreamPublisher Configuration* on page 71
- *Configuring a New Adapter* on page 75

Configuring a New Adapter

Configure a new adapter by creating a configuration file for it. The configuration file defines the adapter component chain through which data is processed, and the connection to Event Stream Processor.

Prerequisites

Create any custom transporters and formatters that you want to use in this adapter instance.

Task

1. Create an <Adapter> element and include all the elements from the steps below within this element.
2. Add a <Name> element and specify a name for the adapter instance.
3. Add a <Description> element and specify the purpose of the adapter.
4. Add a <Modules> element to contain all of the modules for your adapter instance.
5. For each module, specify:

Parameter	Description
InstanceName	Type: <code>string</code> (Required) Instance name of the specific module you want to use. For example, <code>MyInputTransporter</code> .
Name	Type: <code>string</code> (Required) The name of the module as defined in the <code>modulesdefine.xml</code> file. This should be a unique name. For example, <code>MyCustomInputTransporter</code> .
Next	Type: <code>string</code> (Required if another module follows this one) Instance name of the module that follows this one.
BufferMaxSize	Type: <code>integer</code> (Advanced) Capacity of the buffer queue between this module and the next. The default value is 10240.
Parallel	Type: <code>boolean</code> (Optional; applies only to row-based formatters) If set to true, the module runs as a separated thread. If set to false, the module shares thread with other modules. The default value is true.

Parameter	Description
Parameters	<p>(Required) Parameters for the current module. For a custom module, the subelement can reflect the name or type of the module, for example <code><MyCustomInputTransporterParameters></code>.</p> <p>EspPublisher, EspMultiStreamPublisher, EspSubscriber, and EspMultiStreamSubscriber all have set parameters that need to be configured specifically.</p>

6. Configure one of these modules for your adapter:

For the EspPublisher, add a `<EspPublisherParameters>` sub-element and specify:

Parameter	Description
ProjectName	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if it is running in managed mode) Name of the ESP project to which the adapter is connected. For example, <code>EspProject2</code>.</p> <p>This is the same project tag that you specify later in the adapter configuration file in the Name element within the Event Stream Processor (EspProjects) element.</p> <p>If you are starting the adapter with the ESP project to which it is attached (that is, running the adapter in managed mode), you need not set this element as the adapter automatically detects the project name.</p>
EspPublisherParameters	(Required) Element containing elements for the ESP publisher.

Parameter	Description
StreamName	<p>Type: <code>string</code></p> <p>(Required if adapter is running in standalone mode; optional if it is running in managed mode) Name of the ESP stream to which the adapter publishes data.</p> <p>If you are starting the adapter with the ESP project to which it is attached (that is, running the adapter in managed mode), you need not set this element as the adapter automatically detects the stream name.</p>
MaxPubPoolSize	<p>Type: <code>positive integer</code></p> <p>(Optional) Maximum size of the record pool. Record pooling, also referred to as block or batch publishing, allows for faster publication since there is less overall resource cost in publishing multiple records together, compared to publishing records individually.</p> <p>Record pooling is disabled if this value is 1. The default value is 256.</p>
MaxPubPoolTime	<p>Type: <code>positive integer</code></p> <p>(Optional) Maximum period of time, in milliseconds, for which records are pooled before being published. If not set, pooling time is unlimited and the pooling strategy is governed by maxPubPoolSize. No default value.</p>
UseTransactions	<p>Type: <code>boolean</code></p> <p>(Optional) If set to true, pooled messages are published to Event Stream Processor in transactions. If set to false, they are published in envelopes. The default value is false.</p>
SafeOps	<p>Type: <code>boolean</code></p> <p>(Advanced) Converts the opcodes INSERT and UPDATE to UPSERT, and converts DELETE to SAFEDELETE. The default value is false.</p>

Parameter	Description
SkipDels	Type: <code>boolean</code> (Advanced) Skips the rows with opcodes DELETE or SAFEDELETE. The default value is false.

For `EspMultiStreamPublisher`, add a `<EspMultiStreamPublisherParameters>` sub-element and specify:

Parameter	Description
Streams	(Required) Section containing the Stream , ProjectName , StreamName , and ColumnMapping parameters.
Stream	(Required) Section containing details for the target project and streams to which the adapter is connected. Contains the ProjectName , StreamName , and ColumnMapping parameters. You can specify multiple Stream sections.
Filter	(Optional) Section containing the MatchString parameter.
MatchString	(Optional) Filters records with one or more column values. Contains a value attribute for specifying the value you want to filter by. No default value.
ProjectName	Type: <code>string</code> (Required) The name of the ESP project to which the adapter belongs. This is the same project as specified in the Name parameter of the ESP projects section of the adapter configuration file.
StreamName	Type: <code>string</code> (Required) The name of the target stream to which the adapter connects.

Parameter	Description
ColumnMapping	Type: string (Optional) The column index list in the source row to be published to the target stream. Separate the columns using space char. If you do not set this parameter, all columns are published to the target stream.

For EspSubscriber, add a <EspSubscriberParameters> subelement and specify:

Parameter	Description
ProjectName	Type: string (Required if running adapter in standalone mode; optional if running in managed mode) Specifies the unique project tag of the ESP project to which the adapter is connected. For example, EspProject2. This is the same project tag that you specify later in the adapter configuration file in the Name element within the Event Stream Processor (EspProjects) element. If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this element as the adapter automatically detects the project name.
StreamName	Type: string (Required if running adapter in standalone mode; optional if running in managed mode) Name of the ESP stream from which the adapter subscribes to data. If you are starting the adapter with the ESP project to which it is attached (running the adapter in managed mode), you do not need to set this element as the adapter automatically detects the stream name.

For EspMultiStreamSubscriber, add a <EspMultiStreamSubscriberParameters> subelement and specify:

Parameter	Description
Streams	(Required) Section containing the Stream , ProjectName , StreamName , and ColumnMapping parameters.
Stream	(Required) Section containing details for the target project and streams to which the adapter is connected. Contains the ProjectName , StreamName , and ColumnMapping parameters. You can specify multiple Stream elements.
ProjectName	Type: string (Required) The name of the ESP project to which the adapter belongs. This is the same project as specified in the Name parameter of the ESP projects section of the adapter configuration file.
StreamName	Type: string (Required) The name of the source stream to which the adapter connects.
ColumnMapping	Type: string (Optional) The column list of the source stream to which the adapter connects. Separate the columns with a space char. This parameter contains the "enumtype" attribute which has these valid values: <ul style="list-style-type: none"> • index – the index of the column. • name – the column name. The default value is name.

7. Specify a connection to Event Stream Processor:

XML Element	Description
EspProjects	(Required) Element containing elements for connecting to Event Stream Processor.
EspProject	(Required) Element containing the Name and Uri elements. Specifies information for the ESP project to which the adapter is connected.

XML Element	Description
Name	<p>Type: string</p> <p>(Required) Specifies the unique project tag of the ESP project which the EspConnector (publisher/subscriber) module references.</p>
Uri	<p>Type: string</p> <p>(Required) Specifies the total project URI to connect to the ESP project. For example, <code>esp://localhost:19011/ws1/p1</code>.</p>
Security	<p>(Required) Element containing all the authentication elements below. Specifies details for the authentication method used for Event Stream Processor.</p>
User	<p>Type: string</p> <p>(Required) Specifies the user name required to log in to Event Stream Processor (see Auth-Type). No default value.</p>
Password	<p>Type: string</p> <p>(Required) Specifies the password required to log in to Event Stream Processor (see espAuth-Type).</p> <p>Includes an "encrypted" attribute indicating whether the Password value is encrypted. The default value is false. If set to true, the password value is decrypted using RSAKeyStore and RSAKeyStorePassword.</p>

XML Element	Description
AuthType	<p>Type: <code>string</code></p> <p>(Required) Method used to authenticate to the Event Stream Processor. Valid values are:</p> <ul style="list-style-type: none"> • server_rsa – RSA authentication using keystore • kerberos – Kerberos authentication using ticket-based authentication • user_password – LDAP, SAP BI, and Native OS (user name/password) authentication <p>If the adapter is operated as a Studio plug-in, AuthType is overridden by the Authentication Mode Studio start-up parameter.</p>
RSAKeyStore	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies the location of the RSA keystore, and decrypts the password value. Required if AuthType is set to <code>server_rsa</code>, or the encrypted attribute for Password is set to true, or both.</p>
RSAKeyStorePassword	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies the keystore password, and decrypts the password value. Required if AuthType is set to <code>server_rsa</code>, or the encrypted attribute for Password is set to true, or both.</p>
KerberosKDC	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies host name of Kerberos key distribution center. Required if AuthType is set to <code>kerberos</code>.</p>
KerberosRealm	<p>Type: <code>string</code></p> <p>(Dependent required) Specifies the Kerberos realm setting. Required if AuthType is set to <code>kerberos</code>.</p>

XML Element	Description
KerberosService	Type: string (Dependent required) Specifies the Kerberos principal name that identifies an Event Stream Processor cluster. Required if AuthType is set to kerberos.
KerberosTicketCache	Type: string (Dependent required) Specifies the location of the Kerberos ticket cache file. Required if AuthType is set to kerberos.
EncryptionAlgorithm	Type: string (Optional) Used when the encrypted attribute for Password is set to true. If left blank, RSA is used as default.

8. (Optional) Add a GlobalParameters element. This node is visible to all modules that are configured within the adapter configuration file.
 - a) Define XML schema for the GlobalParameter in the \$ESP_HOME/adapters/framework/parametersdefine.xsd file.
 - b) Call the **Utility.getGlobalParameterNode()** or **utility.getGlobalParameters()** function to get the XML object for this node.

Here is an example of a configuration file for the Socket JSON Input adapter:

```
<?xml version="1.0" encoding="utf-8"?>
<Adapter>
  <Name>socket_json_input</Name>
  <Description>An adapter which receives JSON message from socket
server, transforms to ESP data format, and publishes to ESP stream.</
Description>
  <Log4jProperty>./log4j.properties</Log4jProperty>
  <Modules>
    <Module type="transporter">
      <InstanceName>MyExampleSocketInTransporter</InstanceName>
      <Name>SocketInputTransporter</Name>
      <Next>MyJsonStreamToJsonStringFormatter</Next>
      <Parameters>
        <SocketInputTransporterParameters>
          <Host>localhost</Host>
          <Port>9998</Port>
          <EpFile></EpFile>
          <Retryperiod>60</Retryperiod>
          <Blocksize>512</Blocksize>
          <KeepAlive>true</KeepAlive>
        </SocketInputTransporterParameters>
      </Parameters>
    </Module>
  </Modules>
</Adapter>
```

```

    <Module type="formatter">
      <InstanceName>MyJsonStreamToJsonStringFormatter</
InstanceName>
      <Name>JsonStreamToJsonStringFormatter</Name>
      <Next>MyJsonInFormatter</Next>
      <Parameters />
    </Module>

    <Module type="formatter">
      <InstanceName>MyJsonInFormatter</InstanceName>
      <Name>JsonStringToEspFormatter</Name>
      <Next>MyInStream_Publisher</Next>
      <Parameters>
        <JsonStringToEspFormatterParameters>
          <DateFormat>yyyy-MM-dd HH:mm:ss.SSS</DateFormat>
          <TimestampFormat>yyyy/MM/dd HH:mm:ss</
TimestampFormat>
          <ColumnMappings>
            <ColsMapping streamname="EntityStream"
rootpath="entities">
              <Column>display_text</Column>
              <Column>domain_role</Column>
              <Column>offset</Column>
              <Column>length</Column>
            </ColsMapping>
          </ColumnMappings>
        </JsonStringToEspFormatterParameters>
      </Parameters>
    </Module>

    <Module type="espconnector">
      <InstanceName>MyInStream_Publisher</InstanceName>
      <Name>EspPublisher</Name>
      <Parameters>
        <EspPublisherParameters>
          <ProjectName>EspProject1</ProjectName>
          <StreamName>EntityStream</StreamName>
          <MaxPubPoolSize>1</MaxPubPoolSize>
          <UseTransactions>>false</UseTransactions>
          <SafeOps>>true</SafeOps>
          <SkipDels>>true</SkipDels>
        </EspPublisherParameters>
      </Parameters>
    </Module>

  </Modules>

  <EspProjects>
    <EspProject>
      <Name>EspProject1</Name>
      <Uri>esp://localhost:19011/sample_workspace/
socket_json_input</Uri>
      <Security>
        <User></User>
        <Password encrypted="false"></Password>
      </Security>
    </EspProject>
  </EspProjects>

```

```

        <AuthType>user_password</AuthType>
        <!-- <RSAKeyStore>/keystore/keystore.jks</RSAKeyStore>
<RSAKeyStorePassword>Sybase123</RSAKeyStorePassword> -->
        <!-- <KerberosKDC>KDC</KerberosKDC>
<KerberosRealm>REALM</KerberosRealm>
        <KerberosService>service/instance</KerberosService>
<KerberosTicketCache>/tmp/krb5cc_user</KerberosTicketCache> -->
        <EncryptionAlgorithm>RSA</EncryptionAlgorithm>
    </Security>
</EspProject>
</EspProjects>
<GlobalParameters></GlobalParameters>
</Adapter>

```

See also

- *Implementing Schema Discovery in a Custom Adapter* on page 63
- *Creating a Cnxml File for a Custom Adapter* on page 110
- *EspSubscriber Configuration* on page 66
- *EspMultiStreamSubscriber Configuration* on page 67
- *EspPublisher Configuration* on page 69
- *EspMultiStreamPublisher Configuration* on page 71
- *Event Stream Processor Configuration* on page 73

Starting an Adapter

You can start an adapter either in unmanaged or managed mode. In unmanaged mode, the adapter is started separately from the ESP project, and in managed mode, the adapter is started with the ESP project.

Mode	Steps
Unmanaged	<ol style="list-style-type: none"> 1. Start the ESP node by running <code>start_node.bat</code> or <code>start_node.sh</code> in the adapter instance folder. 2. Start the adapter by running <code>start_adapter.bat</code> or <code>start_adapter.sh</code>

Mode	Steps
Managed	<ol style="list-style-type: none"> <li data-bbox="396 215 821 244">1. Create a cxml file for your adapter. <li data-bbox="396 251 1184 314">2. Add an ATTACH ADAPTER statement in the CCL file of your project to reference the adapter. For example, <pre data-bbox="434 326 1130 499" style="background-color: #f0f0f0; padding: 5px;">ATTACH INPUT ADAPTER Generic_Input_Adapter__external_1 TYPE genericinputadapter to BaseInput PROPERTIES configFilePath = 'C:/sybase/ESP-5_1/adapters/framework/instances/ file_csv_input/adapter_config.xml' ;</pre> <li data-bbox="396 510 588 539">3. Start the node. <p data-bbox="434 545 542 574">Windows:</p> <pre data-bbox="434 586 1013 661" style="background-color: #f0f0f0; padding: 5px;">cd %ESP_HOME%\cluster\nodes\<nodename> %esp_home%\bin\esp_server="" <nodename>.xml<="" --cluster-node="" pre=""> <p data-bbox="434 687 458 716">or</p> <p data-bbox="434 722 512 751">UNIX:</p> <pre data-bbox="434 760 995 835" style="background-color: #f0f0f0; padding: 5px;">cd \$ESP_HOME/cluster/nodes/<nodename> \$ESP_HOME/bin/esp_server --cluster-node <nodename>.xml</pre> </nodename>></pre> <li data-bbox="396 843 753 873">4. Compile CCL to create CCX. <p data-bbox="434 878 542 907">Windows:</p> <pre data-bbox="434 920 1143 968" style="background-color: #f0f0f0; padding: 5px;">%ESP_HOME%\bin\esp_compiler -i <modelname>.ccl -o <modelname>.ccx</pre> <p data-bbox="434 994 458 1024">or</p> <p data-bbox="434 1029 512 1058">UNIX:</p> <pre data-bbox="434 1067 1130 1116" style="background-color: #f0f0f0; padding: 5px;">\$ESP_HOME/bin/esp_compiler -i <modelname>.ccl -o <modelname>.ccx</pre> <li data-bbox="396 1124 790 1154">5. Deploy the project on the cluster. <p data-bbox="434 1159 542 1189">Windows:</p> <pre data-bbox="434 1201 1180 1350" style="background-color: #f0f0f0; padding: 5px;">%ESP_HOME%\bin\esp_cluster_admin" --uri=esp:// localhost:19011 --username=sybase --password=sybase --add_project -- workspacename= <workspacename> --project-name=<projectname> -- ccx=<modelname>.ccx</pre> <p data-bbox="434 1376 458 1406">or</p> <p data-bbox="434 1411 512 1440">UNIX:</p> <pre data-bbox="434 1449 1180 1546" style="background-color: #f0f0f0; padding: 5px;">\$ESP_HOME/bin/esp_cluster_admin" --uri=esp:// localhost:19011 --username=sybase --password=sybase --add_project -- workspacename=</pre>

Mode	Steps
	<pre><workspacename> --project-name=<projectname> -- ccx=<modelname>.ccx</pre> <p>6. Start the deployed project on the cluster.</p> <p>Windows:</p> <pre>%ESP_HOME%\bin\esp_cluster_admin" --uri=esp:// localhost:19011 --username=sybase --password=sybase --start_project -- workspace-name=<workspacename> --project- name=<projectname></pre> <p>or</p> <p>UNIX:</p> <pre>\$ESP_HOME/bin/esp_cluster_admin" --uri=esp:// localhost:19011 --username=sybase --password=sybase --start_project -- workspace-name=<workspacename> --project- name=<projectname></pre>

See also

- *Creating a Cnxml File for a Custom Adapter* on page 110
- *Stopping an Adapter* on page 88
- *Chapter 4, Adapter Integration Framework* on page 109
- *Adapter Toolkit: Sample cnxml File for Output Adapters* on page 93
- *Adapter Toolkit: Sample cnxml File for Input Adapters* on page 93

Stopping an Adapter

You can stop an adapter either in unmanaged or managed mode. In unmanaged mode, the adapter is stopped separately from the ESP project, and in managed mode, the adapter is stopped with the ESP project.

Mode	Steps
Unmanaged	<ol style="list-style-type: none"> 1. Stop the ESP node by running <code>stop_node.bat</code> or <code>stop_node.sh</code> in the adapter instance folder. 2. Stop the adapter by running <code>stop_adapter.bat</code> or <code>stop_adapter.sh</code>

Mode	Steps
Managed	<p>Prerequisites: Create a .cnxml file for your adapter, and add an ATTACH ADAPTER statement in the CCL file of your project to reference the adapter.</p> <p>1. Stop the project to which the adapter is attached.</p> <p>Windows:</p> <pre>cd %ESP_HOME%\cluster\nodes\<node_name> <project-name><="" <workspace-name>="" --="" --uri="esp[s]://<host>:19011" --username="<name>" esp_cluster_admin="" password="<pass>" pre="" project="" stop=""> <p>UNIX:</p> <pre>cd \$ESP_HOME/cluster/nodes/<node_name> esp_cluster_admin --uri=esp[s]://<host>:19011 --username=<name> -- password=<pass> -- stop project <workspace-name>/ <project-name></pre> <p>You can also stop the adapter by issuing the stop adapter command from the esp_client tool, and restart it by issuing the start adapter command.</p> </node_name>></pre>

See also

- *Starting an Adapter* on page 86
- *Creating a Cnxml File for a Custom Adapter* on page 110
- *Chapter 4, Adapter Integration Framework* on page 109
- *Adapter Toolkit: Sample cnxml File for Output Adapters* on page 93
- *Adapter Toolkit: Sample cnxml File for Input Adapters* on page 93

Adapter Toolkit Examples

The `%ESP_HOME%/adapters/framework/examples` directory contains various example modules for the adapter toolkit.

Folder Name	Description
<code>discover</code>	Contains an example adapter that has schema discovery implemented. Before running this example, copy the <code>ExampleAdapterForDiscovery.cnxml</code> file to the <code>\$ESP_HOME/ESP-5_1/lib/adapters</code> directory
<code>dual_direction_transporter</code>	Contains an example adapter that concurrently processes input and output data.
<code>esp_pipe</code>	Contains an example adapter that pipes two ESP projects together.
<code>multistream_publisher</code>	Contains an example adapter that uses the multi-stream publisher module.
<code>multistream_subscriber</code>	Contains an example adapter that uses multi-stream subscriber module.
<code>output</code>	Contains an example output adapter that outputs ESP stream data to the console.
<code>polling_input</code>	Contains an example of a polling input adapter.
<code>row_input</code>	Contains an example input adapter which publishes pooled messages in transactions.
<code>single_inputtransporter</code>	Contains an example adapter which has only two modules: a transporter and an <code>espconnector</code> .
<code>src</code>	Contains the source code for the example modules.
<code>streaming_input</code>	Contains an example streaming input adapter.
<code>streaming_output</code>	Contains an example streaming output adapter.
<code>stringlist_input</code>	Contains an example adapter which uses the String List to ESP Formatter.

Folder Name	Description
stringlist_output	Contains an example adapter which uses the ESP to String List Formatter.

Running an Adapter Example

Use the sample content within the `$ESP_HOME/adapters/framework/examples` directory to run an adapter example.

Prerequisites

1. Stop any other external adapter that is using one of the transporter or formatter modules provided by SAP.
2. Back up the `$ESP_HOME/adapters/framework/config/modulesdefine.xml` file.
3. Copy `$ESP_HOME/adapters/framework/examples/modulesdefine.xml` to the `$ESP_HOME/adapters/framework/config` directory.
4. Edit the `set_example_env.bat` or `set_example_env.sh` script and the `adapter_config.xml` file for each example to specify correct values for the user name and password parameters. If you run the project against the node started by the `set_example_env.bat` or `set_example_env.sh` script, you can find the username and password in `$ESP_HOME%/cluster/examples/csi_local.xml`. The default user is sybase and the default password is sybase.

Task

These steps are applicable to all examples except from the schema discovery example located in the `$ESP_HOME/adapters/framework/examples/discover` directory.

1. Start the cluster manager by executing `start_node.bat` or `start_node.sh`.
2. Start the ESP project by executing `start_project.bat` or `start_project.sh`.
3. Start a subscription by using one of the various `subscribe*.bat/subscribe*.sh` or `statistic_subscribe*.bat/statistic_subscribe*.sh` scripts to validate the results of the adapter.
4. Start the adapter by running the `start_adapter.bat` or `start_adapter.sh` commands.
5. Run one of the examples in the `$ESP_HOME/adapters/framework/examples` directory.

Next

Restore the `$ESP_HOME/adapters/framework/config/modulesdefine.xml` file.

Running the Schema Discovery Adapter Example

Use the sample content within the `$ESP_HOME/adapters/framework/examples/discover` directory to run the schema discovery adapter example.

Prerequisites

1. Stop any other external adapter that is using one of the transporter or formatter modules provided by SAP.
2. Back up the `$ESP_HOME/adapters/framework/config/modulesdefine.xml` file.
3. Copy `$ESP_HOME/adapters/framework/examples/modulesdefine.xml` to the `$ESP_HOME/adapters/framework/config` directory.
4. Copy the file `$ESP_HOME/adapters/framework/examples/discover/ExampleAdapterForDiscovery.cnxml` to the `$ESP_HOME/lib/adapters` directory.
5. Edit the `set_example_env.bat` or `set_example_env.sh` script and the `adapter_config.xml` file for each example to specify correct values for the user name and password parameters. If you run the project against the node started by the `set_example_env.bat` or `set_example_env.sh` script, you can find the username and password in `$ESP_HOME%/cluster/examples/csi_local.xml`. The default user is sybase and the default password is sybase.

Task

1. Start ESP Studio.
2. In the ESP Studio Authoring perspective, drag and drop the **Example Adapter for schema discovery** onto the canvas.
3. Edit the adapter's Adapter Directory Path property and point it to the directory where the discover example resides. For example, `$ESP_HOME/adapters/framework/examples/discover`.
4. Click the adapter's schema discovery icon and complete the steps in the discovery wizard to add a new element to the project matching the discovered schema.

Next

Restore the `$ESP_HOME/adapters/framework/config/modulesdefine.xml` file.

Adapter Toolkit: Sample cxml File for Input Adapters

Adapter type: `toolkit_input`. Sample cxml file for an input adapter created using the adapter toolkit. You can use this file as reference for creating your own cxml file for your custom adapter.

This file is located in the `$ESP_HOME/lib/adapters` directory. Cxml files are required if you want to manage your custom external adapter using the ESP Server. Set these properties in the ESP Studio adapter properties dialog.

If you use the CCL **ATTACH ADAPTER** statement to attach an adapter, you must supply the adapter type.

Property Label	Description
Adapter Configuration File	Property ID: <code>configFilePath</code> Type: <code>filename</code> (Required) Specify the path to the adapter configuration file.

See also

- *Adapter Toolkit: Sample cxml File for Output Adapters* on page 93
- *Chapter 4, Adapter Integration Framework* on page 109
- *Starting an Adapter* on page 86
- *Creating a Cxml File for a Custom Adapter* on page 110
- *Stopping an Adapter* on page 88

Adapter Toolkit: Sample cxml File for Output Adapters

Adapter type: `toolkit_output`. Sample cxml file for an output adapter created using the adapter toolkit. You can use this file as reference for creating your own cxml file for your custom adapter.

This file is located in the `$ESP_HOME/lib/adapters` directory. Cxml files are required if you want to manage your custom external adapter using the ESP Server. Set these properties in the ESP Studio adapter properties dialog.

If you use the CCL **ATTACH ADAPTER** statement to attach an adapter, you must supply the adapter type.

Property Label	Description
Adapter Configuration File	Property ID: configFilePath Type: <code>filename</code> (Required) Specify the path to the adapter configuration file.

See also

- *Adapter Toolkit: Sample cxml File for Input Adapters* on page 93
- *Chapter 4, Adapter Integration Framework* on page 109
- *Starting an Adapter* on page 86
- *Creating a Cxml File for a Custom Adapter* on page 110
- *Stopping an Adapter* on page 88

Debugging a Custom Adapter

Debug a custom Java adapter (that was built using the adapter toolkit) by starting it in debug mode and using an Integrated Development Environment (IDE) that supports remote debugging, such as Eclipse.

Prerequisites

- Install Java Runtime Environment (JRE) version 1.6 or 1.7.
- If debugging the adapter using Eclipse, install Eclipse version 3.7 (Indigo) or higher.

Task

The steps below describe how to debug a custom or an example adapter using an Eclipse IDE. You can use similar steps to debug a custom adapter using another IDE that supports remote debugging.

1. Choose an example adapter from `%ESP_HOME%\adapters\framework\examples` or another adapter of your choice.
2. Compile the transporter and formatter module Java files to .class files (debug version) and compress them to .jar files using the **jar.exe** tool.
3. Copy these debug files to the `%ESP_HOME%\adapters\framework\libj` directory.
4. Back up the `modulesdefine.xml` file in the `$ESP_HOME/adapters/framework/config` directory.
5. (Perform only if debugging an example adapter) Prepare the adapter configuration file:

- a) Copy the `$ESP_HOME/adapters/framework/examples/modulesdefine.xml` file to the `$ESP_HOME/adapters/framework/config`.
- b) Edit the `set_example_env.bat` or `set_example_env.sh` files and the adapter configuration file for the example adapter. Specify values for the username and password elements in the example adapter configuration file.

6. Start the ESP cluster node:

Windows

```
cd %ESP_HOME%\cluster\nodes\
%ESP_HOME%\bin\esp_server --cluster-node <nodename>.xml
```

or

UNIX

```
cd $ESP_HOME/cluster/nodes/<nodename>
$ESP_HOME/bin/esp_server --cluster-node <nodename>.xml
```

7. Deploy the ESP project on the cluster:

Windows

```
%ESP_HOME%\bin\esp_cluster_admin" --uri=esp://localhost:19011
--username=sybase --password=sybase --add_project --
workspacename=
<workspacename> --project-name=<projectname> --
ccx=<modelname>.ccx
```

or

UNIX

```
$ESP_HOME/bin/esp_cluster_admin" --uri=esp://localhost:19011
--username=sybase --password=sybase --add_project --
workspacename=
<workspacename> --project-name=<projectname> --
ccx=<modelname>.ccx
```

8. Start the deployed project on the cluster:

Windows

```
%ESP_HOME%\bin\esp_cluster_admin" --uri=esp://localhost:19011
--username=sybase --password=sybase --start_project --
workspace-name=<workspacename> --project-name=<projectname>
```

or

UNIX

```
$ESP_HOME/bin/esp_cluster_admin" --uri=esp://localhost:19011
--username=sybase --password=sybase --start_project --
workspace-name=<workspacename> --project-name=<projectname>
```

9. Modify the `start.bat` or `start.sh` script file in the `%ESP_HOME%\adapters\framework\bin` directory to set the **suspend** debug parameter to `y`.
For example,

```
set DEBUG_PARA=-Xdebug
Xrunjdpw:transport=dt_socket,address=8998,server=y,suspend=y
```

10. Start the adapter in debug mode:

Windows

```
%ESP_HOME%\adapters\framework\bin\start.bat
<ADAPTER_EXAMPLE_CONFIG_FILE> -debug
```

or

```
$ESP_HOME/adapters/framework/bin/start.sh
<ADAPTER_EXAMPLE_CONFIG_FILE> -debug
```

where <ADAPTER_EXAMPLE_CONFIG_FILE> specifies the full path to the configuration file of the adapter you are debugging.

11. Launch Eclipse.

12. Select **Run > Debug Configurations**.

13. On the left-hand side of the Debug Configurations window, select **Remote Java Application**, then right-click and select **New** to create a connection to the adapter you wish to debug.

14. On the right-hand side of the Debug Configuration window, select the **Connect** tab:

- a) Specify the name of your adapter in the **Name** field.
- b) Use the **Browse...** button to select the ESP project to which your adapter is connected.
- c) Select **Standard (Socket Attach)** from the drop down menu for Connection Type.
- d) Specify localhost for the **Host** connection property.
- e) Specify 8998 for the **Port** connection property.
- f) Click **Apply**.

15. Select **Run > Toggle Breakpoint** to create a breakpoint at a specific line.

Set all breakpoints before advancing to the next step.

16. Click **Debug**.

See also

- *Accessing Adapter Toolkit API Reference Information* on page 13
- *Create a Custom Adapter* on page 12
- *Formatter Modules* on page 40
- *Transporter Modules* on page 14
- *EspConnector Modules* on page 66

Creating Custom External Adapters Using SDKs

Follow general guidelines to create a custom external adapter using one of the SDKs (Java, C/C++, or .NET).

1. Obtain an SDK instance.
2. Create credentials for the required authentication type.
3. Connect to an ESP project using these credentials.
4. Create a publisher to publish data to the ESP Server.
5. Create a subscriber to subscribe to records from a project in the ESP Server.
6. Publish or subscribe to data in Event Stream Processor.

Java External Adapters

Use the Java SDK to build a custom Java external adapter.

Connecting to a Project

Connect to a project using your authentication credentials.

1. Get the project:

```
String projectUriStr = "esp://localhost:19011/ws1/p1";  
Uri uri = new Uri.Builder(projectUriStr).create();  
project = sdk.getProject(uri, credentials);
```

2. Connect to the project:

```
project.connect(60000);
```

Here, 60000 refers to the time, in milliseconds, that the ESP Server waits for the connection call to complete before timing out.

Creating a Publisher

Create and connect to a publisher, then publish a message.

1. Create and connect to a publisher:

```
Publisher pub = project.createPublisher();  
pub.connect();
```

2. To create and publish a message, call a stream and the stream name, call the message writer, call the row writer, and publish:

CHAPTER 3: Creating Custom External Adapters Using SDKs

```
String streamName = "Stream1";
Stream stream = project.getStream(streamName);
MessageWriter mw = pub.getMessageWriter(streamName);
RelativeRowWriter writer= mw.getRelativeRowWriter();
mw.startEnvelope(0); // can also be mw.startTransaction() for
transactions.
for (int i = 0; i < recordsToPublish.length; i++) {
    addRow(writer, incomingRecords[i], stream);
}
mw.endBlock();
pub.publish(mw);
```

Sample Java Code for addRow

The addRow operation adds a single record row to messages published to the ESP Server.

Use opcodes to update the table with a new row.

```
Schema schema = stream.getEffectiveSchema();
DataType[] colTypes = schema.getColumnTypes();
rowWriter.startRow();
rowWriter.setOperation(Stream.Operation.UPSERT);
for (int fieldIndex = 0; fieldIndex < schema.getColumnCount();
fieldIndex++) {
    String name = (String) colNames[fieldIndex];
    attValue = record.get(fieldIndex);
switch(dataType){
    case BOOLEAN:      writer.setBoolean((Boolean) attValue); break;
    case INTEGER:      writer.setInteger((Integer) attValue); break;
    case TIMESTAMP:    writer.setTimestamp((Date) attValue);
break;
} //switch
} //for loop
rowWriter.endRow();
```

Subscribing Using Callback

Perform callbacks for new data.

1. Create the subscriber options:

```
SubscriberOptions.Builder builder = new
SubscriberOptions.Builder();
builder.setAccessMode(AccessMode.CALLBACK);
builder.setPulseInterval(pulseInterval);
SubscriberOptions opts = builder.create();
```

Set the access mode to CALLBACK and the pulse interval for how often you want to make the callback.

2. Create the subscriber and register the callback:

```
Subscriber sub = project.createSubscriber(opts);
sub.setCallback(EnumSet.allOf(SubscriberEvent.Type.class),
this);
sub.subscribeStream(streamName);
sub.connect();
```

`sub.setCallback` is the class that implements the `processEvent` method and gets called by the callback mechanism.

3. Create the callback class, which registers with the subscriber.

a) Implement `Callback<SubscriberEvent>`.

b) Implement the `getName()` and `processEvent(SubscriberEvent)` methods.

```

        public void processEvent(SubscriberEvent event) {
            switch (event.getType()) {
                case SYNC_START:    dataFromLogstore=true;    break;
                case SYNC_END:     dataFromLogStore=false;
break;
                case ERROR:       handleError(event);
break;
                case DATA:       handleData(event);         break;
                case DISCONNECTED: cleanupExit();           break;
            }
        }

```

A separate method named `handleData` is declared in this example, which is referenced in Step 4. The name of the method is variable.

Note: When the event is received, the callback mechanism calls `processEvent` and passes the event to it.

4. (Optional) Use `handleData` to complete a separate method to retrieve and use subscribed data. Otherwise, data can be directly processed in `processEvent`:

```

public void handleData(SubscriberEvent event) {
    MessageReader reader = event.getMessageReader();
    String streamName= event.getStream().getName();
    while ( reader.hasNextRow() ) {
        RowReader row = reader.nextRowReader();
        int ops= row.getOperation().code();
        String[] colNames=row.getSchema().getColumnNames();
        List record = new ArrayList<Object>();
        for (int j = 0; index = 0; j <
row.getSchema().getColumnCount(); ++j) {
            if ( row.isNull(j) ) { record.add(index,null); index
++; continue; }
            switch ( row.getSchema().getColumnTypes()[j]) {
                case BOOLEAN: record.add(j,
row.getBoolean(j));break;
                case INTEGER: record.add(j,
row.getInteger(j));break;
                case TIMESTAMP: record.add(j,
row.getTimestamp(j)); break;
            }//switch
        }//for loop
        sendRecordToExternalDataSource(record);
    }//while loop
}//handleData

```

The `handleData` event contains a message reader, gets the stream name, and uses the row reader to search for new rows as long as there is data being subscribed to. Datatypes are specified.

Subscribe Using Direct Access Mode

SAP recommends direct access mode only for testing purposes.

```
Subscriber sub = p.createSubscriber(); sub.connect();
sub.subscribeStream("stream1");
while (true) {
    SubscriberEvent event = sub.getNextEvent();
    handleEvent(event);
}
```

Publish Using Callback

Although you can publish in callback mode in special cases, SAP recommends that you do not.

```
PublisherOptions.Builder builder = new PublisherOptions.Builder();
builder.setAccessMode(AccessMode.CALLBACK);
builder.setPulseInterval(pulseInterval);
PublisherOptions opts = builder.create();
Publisher pub = project.createPublisher(opts);
pub.setCallback(EnumSet.allOf(PublisherEvent.Type.class), new
PublisherHandler(project));
pub.connect();
```

`PublisherHandler` implements `Callback<PublisherEvent>`. It also implements the two methods: `getName()` and `processEvent(PublisherEvent event)`.

The script for implementing `processEvent` should look like this:

```
public void processEvent(PublisherEvent event) {
    switch (event.getType()) {
        case CONNECTED: mwriter =
event.getPublisher().getMessageWriter(mstr);
rowwriter = mwriter.getRelativeRowWriter(); break;
        case READY: mwriter.startTransaction(0);
for (int j = 0; j < 100; ++j) {
mrowwriter.startRow();
mrowwriter.setOperation(Operation.INSERT);
for (int i = 0; i < mschema.getColumnCount(); ++i)
{
switch (mtypes[i]) {
case INTEGER: mrowwriter.setInteger(int_value+
+);break;
case DOUBLE: mrowwriter.setDouble(double_value+=1.0);
break;
}
} //columns
mrowwriter.endRow();
} //for
event.getPublisher().publish(mwriter);
case ERROR: break;
case DISCONNECTD:break;
} //switch
} //processEvent
```

C/C++ External Adapters

Use the C/C++ SDK to build custom C/C++ external adapters.

Getting a Project

Create your authentication credentials, and use them to create a project.

All calls to SDK are available as external C calls.

1. Create a credentials object for authentication:

```
#include <sdk/esp_sdk.h>
#include <sdk/esp_credentials.h>
    EspError* error = esp_error_create();
    esp_sdk_start(error);
    EspCredentials * m_creds =
esp_credentials_create(ESP_CREDENTIALS_USER_PASSWORD, error);
    esp_credentials_set_user(espuser.c_str(), error);
    esp_credentials_set_password(m_creds,
esppass.c_str(), error);
```

2. Create a project:

```
    EspUri* m_espUri = NULL; EspProject* m_project = NULL;
    if ( isCluster){
        m_espUri = esp_uri_create_string(project_uri.c_str(),
error);
        m_project = esp_project_get(m_espUri, m_creds ,NULL,error);
        esp_project_connect (m_project,error);
```

Publishing and Subscribing

Create a publisher and subscriber, and implement a callback instance.

1. Create the publisher:

```
    EspPublisherOptions* publisherOptions =
    esp_publisher_options_create (error);
    Int rc
    EspPublisher * m_publisher = esp_project_create_publisher
    (m_project,publisherOptions,error);
    EspStream* m_stream = esp_project_get_stream (m_project,m_opts-
>target.c_str(), error);
    rc = esp_publisher_connect (m_publisher,error);
```

2. Publish (this sample code includes syntax for adding rows to messages):

```
    EspMessageWriter* m_msgwriter = esp_publisher_get_writer
    (m_publisher,m_stream,error);
    EspRelativeRowWriter* m_rowwriter =
    esp_message_writer_get_relative_rowwriter(m_msgwriter, error);
    const EspSchema* m_schema = esp_stream_get_schema
    (m_stream,error);
    int numColumns;
```

```

    rc = esp_schema_get_numcolumns (m_schema, &numColumns,error);
    rc = esp_message_writer_start_envelope(m_msgwriter, 0, error);
    rc = esp_relative_rowwriter_start_row(m_rowwriter, error);
    rc = esp_relative_rowwriter_set_operation(m_rowwriter, (const
ESP_OPERATION_T)opcode, error);
    int32_t colType;
    for (int j = 0;j < numColumns;j++){
        rc = esp_schema_get_column_type (m_schema,j,&colType,error);
        switch (type){
            case ESP_DATATYPE_INTEGER:
                memcpy (&integer_val,(int32_t *)
(dataValue),sizeof(uint32_t));
                rc = esp_relative_rowwriter_set_integer(m_rowwriter,
integer_val, error);
                break;
            case ESP_DATATYPE_LONG:
                memcpy (&long_val,(int64_t *)
(dataValue),sizeof(int64_t));
                rc = esp_relative_rowwriter_set_long(m_rowwriter,
long_val, error);
                break;
        }
    }
}
rc = esp_relative_rowwriter_end_row(m_rowwriter, error);
rc = esp_message_writer_end_block(m_msgwriter, error);
rc = esp_publisher_publish(m_publisher, m_msgwriter, error);

```

3. Create the subscriber options:

```

EspSubscriberOptions * m_subscriberOptions =
esp_subscriber_options_create (error);
    int rc = esp_subscriber_options_set_access_mode(options,
CALLBACK_ACCESS, m_error);
    EspSubscriber * m_subscriber = esp_project_create_subscriber
(m_project,m_subscriberOptions,error);
    rc = esp_subscriber_options_free(options, m_error);
    rc = esp_subscriber_set_callback(subscriber ,
ESP_SUBSCRIBER_EVENT_ALL,
    subscriber_callback, NULL, m_error);
    subscriber_callback is global function which will get called
up.

```

4. Subscribe using callback:

```

void subscriber_callback(const EspSubscriberEvent * event, void *
data) {
    uint32_t type;
    rc = esp_subscriber_event_get_type(event, &type, error);
    switch (type) {
        case ESP_SUBSCRIBER_EVENT_CONNECTED:
            init(event,error);break;
        case ESP_SUBSCRIBER_EVENT_SYNC_START:      fromLogStore =
true; break;
        case ESP_SUBSCRIBER_EVENT_SYNC_END:      fromLogStore
= false; break;
        case ESP_SUBSCRIBER_EVENT_DATA:
            handleData(event,error); break;
    }
}

```

```

        case ESP_SUBSCRIBER_EVENT_DISCONNECTED:
cleanupaExit(); break;
        case ESP_SUBSCRIBER_EVENT_ERROR:
handleError(event,error); break;
    }
} //end subscriber_callback

```

handleData

Sample C/C++ code for the handleData method.

```

EspMessageReader * reader = esp_subscriber_event_get_reader(event,
error);
    EspStream * stream = esp_message_reader_get_stream(reader,
error);
    const EspSchema * schema = esp_stream_get_schema(stream, error);
    EspRowReader * row_reader;
    int32_t int_value; int64_t long_value; time_t date_value;
double double_value;
    int numcolumns, numRows, type;
    rc = esp_schema_get_numcolumns(schema, &numcolumns, error);
    while ((row_reader = esp_message_reader_next_row(reader,
error)) != NULL) {
        for (int i = 0; i < numcolumns; ++i) {
            rc = esp_schema_get_column_type(schema, i, &type,
error);
                switch(type){
                    case ESP_DATATYPE_INTEGER:
                        rc = esp_row_reader_get_integer(row_reader, i,
&int_value, error);
                            break;
                    case ESP_DATATYPE_LONG:
                        rc = esp_row_reader_get_long(row_reader, i,
&long_value, error);
                            break;
                    case ESP_DATATYPE_DATE:
                        rc = esp_row_reader_get_date(row_reader, i,
&date_value, error);
                            break;
                }
        }
    }
}

```

.Net External Adapters

Use the .Net SDK to build a custom .Net external adapter.

Connecting to the Event Stream Processor Server

Set credentials and .Net server options when you connect to the ESP Server.

1. Create an error message store for setting credentials and .Net server options:

```

NetEspError error = new NetEspError();

```

CHAPTER 3: Creating Custom External Adapters Using SDKs

2. Set a new URI:

```
NetEspUri uri = new NetEspUri();  
uri.set_uri("esp://cepsun64amd.mycompany.com:19011", error);
```

3. Create your credentials:

```
NetEspCredentials creds = new  
NetEspCredentials(NetEspCredentials.NET_ESP_CREDENTIALS_T.NET_ESP  
_CREDENTIALS_SERVER_RSA);  
creds.set_user("pengg");  
creds.set_password("1234");  
creds.set_keyfile("../test_data\\keys\\client.pem");
```

4. Set options:

```
NetEspServerOptions options = new NetEspServerOptions();  
options.set_mode(NetEspServerOptions.NET_ESP_ACCESS_MODE_T.NET_CA  
LLBACK_ACCESS);  
server = new NetEspServer(uri, creds, options);  
int rc = server.connect(error);
```

Connecting to a Project

Use sample .Net code to connect to a project.

1. Get the project:

```
NetEspProject project = server.get_project("test", "test",  
error);
```

2. Connect to the project:

```
project.connect(error);
```

Publishing

Create a publisher, add rows, and complete the publishing process.

1. Create a publisher:

```
NetEspPublisher publisher = project.create_publisher(null,  
error);
```

2. Connect to the publisher:

```
Publisher.connect(error);
```

3. Get a stream:

```
NetEspStream stream = project.get_stream("WIN2", error);
```

4. Get the message writer:

```
NetEspMessageWriter writer = publisher.get_message_writer(stream,  
error);
```

5. Get and start the row writer, and set an opcode to insert one row:

```
NetEspRelativeRowWriter rowwriter =  
writer.get_relative_row_writer(error);  
rowwriter.start_row(error);  
rowwriter.set_opcode(1, error);
```

(Optional) If publishing in transaction mode, use these arguments to add multiple rows:


```

NetEspRelativeRowWriter rowwriter =
writer.get_relative_row_writer(error);
for(int i=0; i<100; i++){
    rowwriter.start_row(error);
//add row columns' values
rowwriter.end_row(error);
}

```

6. Publish data:

```
rc = publisher.publish(writer, error);
```

Connecting to a Subscriber

Create and connect to a new subscriber.

1. Create a subscriber:

```

NetEspSubscriberOptions options = new NetEspSubscriberOptions();
options.set_mode(NetEspSubscriberOptions.NET_ESP_ACCESS_MODE_T.NET_CALLBACK_ACCESS);
NetEspSubscriber subscriber = new NetEspSubscriber(options,
error);

```

2. Connect to the subscriber:

```
Subscriber.connect(error);
```

Subscribing Using Callback Mode

Perform callbacks for new data.

1. Set the subscriber options:

```

NetEspSubscriberOptions options = new NetEspSubscriberOptions();
options.set_mode(NetEspSubscriberOptions.NET_ESP_ACCESS_MODE_T.NET_CALLBACK_ACCESS);
NetEspSubscriber subscriber = new NetEspSubscriber(options,
error);

```

2. Create the callback instance:

```

NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK callbackInstance = new
NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK(subscriber_callback);

```

3. Create the callback registry:

```
subscriber.set_callback(NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_EVENT_ALL, callbackInstance, null, error);
```

4. Connect to the subscriber:

```
subscriber.connect(error);
```

5. Subscribe to a stream:

```
subscriber.subscribe_stream(stream, error);
```

6. Implement the callback:

```

Public static void subscriber_callback(NetEspSubscriberEvent
event, ValueType
data) {

```

```

switch (evt.getType())
{
    case (uint)
        (NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_EV
        ENT_CONNECTED):
        Console.WriteLine("the callback happened:
        connected!");
        break;
    (uint)
        ( NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_EV
        ENT_DATA):

```

7. (Optional) Use `handleData` to complete a separate method to retrieve and use subscribed data.

```

NetEspRowReader row_reader = null;
while ((row_reader = evt.getMessageReader().next_row(error)) !=
null) {
    for (int i = 0; i < schema.get_numcolumns(); ++i) {
        if ( row_reader.is_null(i) == 1) {
            Console.Write("null, ");
            continue;
        }
        switch
            (NetEspStream.getType(schema.get_column_type((uint)i, error)))
        {
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTEGER:
                ivalue = row_reader.get_integer(i, error);
                Console.Write(ivalue + ", ");
                break;
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_LONG:
                lvalue = row_reader.get_long(i, error);
                Console.Write(lvalue + ", ");
                break;
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_FLOAT:
                fvalue = row_reader.get_float(i, error);
                Console.Write(fvalue + ", ");
                break;
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_STRING:
                svalue = row_reader.get_string(i, error);
                Console.Write(svalue);
                break;
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_DATE:
                dvalue = row_reader.get_date(i, error);
                Console.Write(dvalue + ", ");
                break;
            case
                NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_TIMESTAMP:
                tvalue = row_reader.get_timestamp(i,
                error);
                Console.Write(tvalue + ", ");

```

CHAPTER 3: Creating Custom External Adapters Using SDKs

```
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BOOLEAN:
                boolvalue = row_reader.get_boolean(i,
error);
                Console.WriteLine(boolvalue + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BINARY:
                uint buffersize = 256;
                binvalue = row_reader.get_binary(i,
buffersize, error);
                Console.WriteLine(System.Text.Encoding.Default.GetString(binvalue) +
", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTERVAL:
                intervalvalue = row_reader.get_interval(i,
error);
                Console.WriteLine(intervalvalue + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY01:
                mon = row_reader.get_money(i, error);
                Console.WriteLine(mon.get_long(error) + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY02:
                lvalue = row_reader.get_money_as_long(i,
error);
                Console.WriteLine(lvalue + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY03:
                mon = row_reader.get_money(i, error);
                Console.WriteLine(mon.get_long(error) + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY10:
                mon = row_reader.get_money(i, error);
                Console.WriteLine(mon.get_long(error) + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY15:
                mon = row_reader.get_money(i, error);
                Console.WriteLine(mon.get_long(error) + ", ");
                break;
            case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BIGDATETIME:
                bdt2 = row_reader.get_bigdatetime(i,
error);
                long usecs = bdt2.get_microseconds(error);
                Console.WriteLine(usecs + ", ");
                break;
        }
    }
```

```
    }  
}
```

8. Disconnect from the subscriber:

```
    rc = subscriber.disconnect(error);  
}
```

The ESP adapter integration framework provides a mechanism for integrating custom external adapters with the ESP Server and ESP Studio.

A custom external adapter is any application that publishes or subscribes to a stream or a set of streams in Event Stream Processor using the publisher and subscriber APIs from any of the Event Stream Processor SDKs.

To start and stop external adapters together with an ESP project or visually from within Studio, create an instance of the adapter plug-in by creating a specialized cxml configuration file for the adapter. Within this file, specify command lines that the ESP Server can invoke during various stages of adapter runtime and initialization.

See also

- *Adapter Toolkit: Sample cxml File for Output Adapters* on page 93
- *Adapter Toolkit: Sample cxml File for Input Adapters* on page 93
- *Starting an Adapter* on page 86
- *Creating a Cxml File for a Custom Adapter* on page 110
- *Stopping an Adapter* on page 88

Cxml Configuration File

The external adapter framework defines the structure of the cxml file that contains the adapter properties.

The external adapter configuration file is an XML file that contains the properties and commands used by Event Stream Processor to start and stop the external adapter, and to optionally run schema discovery, as well as other information that allows you to configure the adapter from the ESP Studio.

Here is the structure of a cxml file:

```
<Adapter>
  type = Can be input or output
  external = True
  id = Required to be unique to the adapter
  label = The adapter's name in Studio
  descr = Description of adapter functionalities

<Library>
  file = simple_ext (Always use this for custom external adapters)
  type = Binary

<Special>
```

```

<Internal>
  **These values are not configurable from Studio. simple_ext
  has a function that parses this cxml file and looks up the internal
  fields to find particular commands and their default values**
  id = See $ESP_HOME/lib/adapters/simple_ext.cxml.template
  for a sample cxml file that contains all possible internal
  parameters and comment blocks indicating their usage.
  label = Description of function
  type = Datatype of function
  default = This is the command that simple_ext executes. For
  example, if you are using x_unixCmdExec, it calls esp_convert with
  targets you specify followed by an upload.

<Section>
  <Parameter>
  **These parameter values are visible (except for id) and
  configurable in Studio.**
  id = The property id that you can reference to in <Internal>
  command calls with $(id's name). This is what you reference when
  writing an adapter in CCL.
  label = The property name which appears in Studio.
  descr = Description of adapter property.
  type = The property datatype.
  use = Whether the property is required, optional, or advanced.
  In Studio, required properties appear in red and advanced ones appear
  on a separate tab.
  default = Default value to use, if you do not set a value for
  the property.
  
```

Creating a Cxml File for a Custom Adapter

Create a cxml configuration file for your custom external adapter so that you can configure the adapter in the ESP Studio, and start and stop it with an ESP project.

1. Specify attributes for the Adapter section:

Attribute	Description
type	(Required) Specify whether this is an input or output adapter.
external	(Required) Set the value to true as this is an external adapter.
id	(Required) Specify a unique identifier for your adapter. This value is listed in the type parameter within the ATTACH ADAPTER statement.
label	(Required) The name of your adapter. This name appears in ESP Studio if you hover over the adapter icon in the Visual Editor.

Attribute	Description
description	(Required) Specify the purpose of the adapter. This is also visible in ESP Studio.

2. Specify attributes for the `Library` section:

Attribute	Description
file	(Required) Specify "simple_ext" as the value. Always specify this for external adapters.
type	(Required) Specify "binary" as the value.

3. Specify internal parameters for the `Special` section. See the `$ESP_HOME/lib/adapters/simple_ext.cnxml.template` sample cnxml file for a complete list of internal parameters and usage details.
4. Specify adapter parameters for the `Section` section. These parameters are visible and configuration in ESP Studio.

For each parameter, specify:

Parameter	Description
<code>id</code>	(Required) The property ID that you can reference in <code><Internal></code> command calls with <code>\$(id's name)</code> . This is what you reference when specifying adapter properties for an adapter in CCL.
<code>label</code>	(Required) The property name that appears in ESP Studio.
<code>descr</code>	(Required) A description of the adapter property.
<code>type</code>	(Required) The property datatype.
<code>use</code>	(Required) Whether the property is required, optional, or advanced. In ESP Studio, required properties appear in red and advanced ones appear on a separate tab.
<code>default</code>	(Optional) The default value to use, if you do not set a value for the property.

See also

- *Configuring a New Adapter* on page 75
- *Starting an Adapter* on page 86
- *Stopping an Adapter* on page 88
- *Chapter 4, Adapter Integration Framework* on page 109
- *Adapter Toolkit: Sample cnxml File for Output Adapters* on page 93
- *Adapter Toolkit: Sample cnxml File for Input Adapters* on page 93

Example Cnxml Configuration File

Example of a cnxml configuration file that uses some of the utilities shipped with Event Stream Processor (**esp_convert**, **esp_upload**, **esp_client**, and **esp_discxmlfiles**) to fully define a functional external adapter that supports file directory browsing, source stream creation, and data loading.

In this example, long lines of script are split for readability and to avoid formatting issues. If you are using this to create your own external adapter configuration file, ensure that all command properties are on a single line, regardless of length.

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

<Adapter type="input" external="true"
  id="simplified_xml_input_plugin"
  label="Simplified external XML file input plugin Adapter"
  descr="Example of uploading an XML file through a simple external
Adapter"
>
  <Library file="simple_ext" type="binary"/>

  <!--
    The special section contains the special internal parameters
    which are prefixed with "x_". Although these are parameters,
    the framework requires that you define these using the <Internal
    .../> element. They are hidden within ESP Studio.
  -->
  <Special>
    <Internal id="x_initialOnly"
      label="Does Initial Loading Only"
      descr="Do initial loading, or the continuous loading"
      type="boolean"
      default="true"
    />
    <Internal id="x_addParamFile"
      label="Add Parameter File"
      type="boolean"
      default="false"
    />
    <Internal id="x_killRetryPeriod"
      label="Period to repeat the stop command until the process
exits"
      type="int"
      default="1"
    />

    <!--
      Convert a file of xml record to ESP Binary format using
      esp_convert;
      pipe into the esp_upload program, naming the upload
      connection:
      $platformStream.$platformConnection
    -->
```



```

<Internal id="x_unixCmdExec"
  label="Execute Command"
  type="string"
  default="$ESP_HOME/bin/esp_convert -p $platformCommandPort
&lt;&quot;$directory/$filename&quot; | $ESP_HOME/bin/esp_upload -m
$platformStream.$platformConnection -p $platformCommandPort"
/>
<Internal id="x_winCmdExec"
  label="Execute Command"
  type="string"
  default="$+/{$ESP_HOME/bin/esp_convert} -p $platformCommandPort
&lt;&quot;$directory/$filename&quot; | $+/{$ESP_HOME/bin/esp_upload}
-m $platformStream.$platformConnection -p $platformCommandPort"
/>

<!--
  use the esp_client command to stop an existing esp_upload
connection named:
  $platformStream.$platformConnection
-->
<Internal id="x_unixCmdStop"
  label="Stop Command"
  type="string"
  default="$ESP_HOME/bin/esp_client -p $platformCommandPort 'kill
every {$platformStream.$platformConnection}' &lt;/dev/null"
/>
<Internal id="x_winCmdStop"
  label="Stop Command"
  type="string"
  default="$+/{$ESP_HOME/bin/esp_client} -p $platformCommandPort
&quot;kill every {$platformStream.$platformConnection}&quot;
&lt;nul"
/>

<!--
  Use the esp_discxmlfiles command to do schema discovery.
  The command below will have '-o "<temp file>"' added to it. It
  writes the discovered schema in this file.
-->
<Internal id="x_unixCmdDisc"
  label="Discovery Command"
  type="string"
  default="$ESP_HOME/bin/esp_discxmlfiles -d &quot;
$directory&quot;;"
/>
<Internal id="x_winCmdDisc"
  label="Discovery Command"
  type="string"
  default="$+/{$ESP_HOME/bin/esp_discxmlfiles} -d &quot;$+/{
$directory}&quot;;"
/>
</Special>

<Section>

<!--

```

CHAPTER 4: Adapter Integration Framework

```
Any parameter defined here is visible in the ESP Studio and
you can
configure it at runtime in the data location explorer.
These are defined according to the $ESP_HOME/etc/Adapter.xsd
schema.
-->

<Parameter id="filename"
  label="File"
  descr="File to upload"
  type="tables"
  use="required"
/>
<Parameter id="directory"
  label="path to file"
  descr="directory to search"
  type="directory"
  use="required"
/>
<Parameter id="propertyset"
  label="propertyset"
  descr="to look up properties in project configuration"
  type="string"
  use="advanced"
  default=""/>
</Section>
</Adapter>
```

External Adapter Properties

See `$ESP_HOME/lib/adapters/simple_ext.cnxml.template` for a sample cnxml file that you can copy and customize. It includes all possible internal parameters, along with comment blocks that include usage notes for each parameter.

Property ID	Type	Description
x_addParamFile	boolean	Determines if the parameter file name is automatically appended to all x_cmd* strings. For example, if you specify the command as cmd -f , and this parameter is set to true, the actual command is executed as cmd -f <value of x_paramFile> .
x_initialOnly	boolean	If true, performs only initial loading. Set to false for continuous loading. Initial loading is useful for adapters that start, load some static data and then finish; therefore, allowing another adapter group to startup in a staged loading scenario.

Property ID	Type	Description
x_killRetryPeriod	integer	If this parameter is greater than 0, the x_{unix,win}CmdStop command is retried every x_killRetry seconds, until the framework detects that the x_{unix,win}CmdExec command has returned. If the parameter is zero, run the x_{unix,win}CmdStop command once and assume it has stopped the x_{unix,win}CmdExec command.
x_paramFile	string	Specifies the file name of the adapter framework, which writes all internal and user-defined parameters. It may use other internal parameters to specify the file name. For example, <code>/tmp/mymodel.\$platformStream. \$platformConnection.\$platformCom- mandPort.cfg</code>
x_paramFormat	string	Stops the adapter, and runs in a separate thread, which returns and stops processes created with x_{unix,win}CmdExec .
x_unixCmdClean/ x_winCmdClean	string	Specifies the command executed to perform cleanup after the adapter stops (x_{unix,win}CmdExec returns).
x_unixCmdConfig/ x_winCmdConfig	string	Specifies the command that preprocess (for example, parsing and checking parameters) the parameter file before the adapter starts. such as parsing and checking parameters. This command might convert the parameters to real format (by reading, parsing, and rewriting the parameter file), which is required by the execution command. If the configure command fails (non-zero return), the adapter fails to start.
x_unixCmdDisc/ x_winCmdDisc	string	Specifies the command that performs discovery. This command adds the <code>-o "<temporary disc filename>"</code> argument, which writes the discovery XML to a file, before it executes.
x_unixCmdExec/ x_winCmdExec	string	Specifies the command that starts the adapter. When the command returns, it indicates that the adapter is done running.

Property ID	Type	Description
x_unixCmdStop/ x_winCmdStop	string	Specifies the command that stops the adapter. The stop command runs from a separate thread and stops all processes created with the x_{unix,win}CmdExec command, which returns the x_{unix,win}CmdExec command.

Note: Some of the commands have both Windows or Unix formats because their paths, environment variables, and script files differ. At runtime, ESP determines which command to use based on the operating system it is running on. All commands have substituted parameters and environment variables. When a command is passed to shell or `cmd.exe`, it runs its own round of substitution.

External Adapter Commands

External adapter commands fall into two categories: those that run on the same host as Studio, and those that run on the same host as the Server.

The discovery commands, **x_unixDiscCmd** and **x_winDiscCmd** always run on the Studio host. All other commands run on the Server host.

The Studio and Server are frequently run on the same host, so the development of all command and driving scripts for the custom adapter are straightforward. The configuration becomes more complex during remote execution when Studio and the Server are running on different hosts.

For example, if the Studio is running on a Windows host, and the Server is set up through Studio to execute on a remote Linux host, it implies that the discovery command and the discovery file name that the framework generates are running and are generated in a Windows environment. The path to the discovery file is a Windows-specific path with drive letters and forward-slash (/) characters used as path separators. In this case, the developer of the connector should write the discovery command to run in a Windows environment while coding all other commands to remotely execute on the Linux box using a user-configured **ssh** or **rsh** command.

Command	Description
x_unixCmdConfig x_winCmdConfig	The configure command should perform any required parsing and parameter checking. It can also convert the parameters into the real format expected by the execution command by reading, parsing, and rewriting the parameter file. If the configure command fails (non-zero return), it is reported as a <code>reset()</code> error, and the adapter fails to start.

Command	Description
x_unixCmdExec x_winCmdExec	When the Server starts the adapter, it executes this command with its ending indicating that the connector has finished.
x_unixCmdStop x_winCmdStop	The stop command runs from a separate thread; it should stop all processes created with the x_{unix,win}CmdExec command, thus causing the x_{unix,win}CmdExec to return.
x_unixCmdClean x_winCmdClean	The clean command runs after the Server has stopped the connection, that is, when x_{unix,win}CmdExec returns.
x_winDiscCmd	<p>This command is for schema discovery. It should write a discovery file into the file name passed to it. Append the parameter -o <temporary disc filename> command before executing it.</p> <pre> <discover> <table name="table_name_1" /> <column name="col_name_1" type="col_type_1"/> . . . <column name="col_name_k" type="col_type_k"/> </table> . . . <table name="table_name_n" /> <column name="col_name_1" type="col_type_1"/> . . . <column name="col_name_1" type="col_type_1"/> </table> </discover> </pre>

User-Defined Parameters and Parameter Substitution

You can create internal parameters and any number of user-defined parameters in the cnxml file.

All system and user-defined parameters can be referenced in the command or script arguments. These parameters behave in a similar way to shell substitution variables. This simple example includes long lines that have been split for readability:

```

<Internal id="x_unixCmdExec"
label="Execute Command"

```

CHAPTER 4: Adapter Integration Framework

```
type="string"
default="$ESP_HOME/bin/esp_convert
        -p $platformCommandPort &lt;&quot;$directory/
$filename&quot;; | $ESP_HOME/bin/esp_upload
        -m $platformStream.$platformConnection
        -p $platformCommandPort"
/>
```

External environment variables, such as `ESP_HOME`, may be expanded, as well as internal system parameters (**platformCommandPort**) and user-defined parameters (filename). The full semantics for parameter expansion are:

```
$name
${name}
${name=value?substitution[:substitution]}
${name<>value?substitution[:substitution]}
${name!=value?substitution[:substitution]}
${name==value?substitution[:substitution]}
${name<value?substitution[:substitution]}
${name<=value?substitution[:substitution]}
${name>value?substitution[:substitution]}
${name>=value?substitution[:substitution]}
```

All forms with `{ }` may have a `+` added after `$` (for example, `${name}`). The presence of `+` means that the result of the substitution is parsed again, and any values in it are substituted. The `\` symbol escapes the next character and prevents any special interpretation.

The conditional expression compares the value of a parameter with a constant value and uses either the first substitution on success or the second substitution on failure. The comparisons `==` and `!=` try to compare the values as numbers. The `=` comparisons and `<>` try to compare values as strings. Any characters like `?`, `:` and `}` in the values must be shielded with `\`. The characters `{` and `}` in the substitutions must be balanced, all unbalanced braces must be shielded with `\`. The quote characters are not treated as special.

This form of substitution, `${...}`, may contain references to other variables. This is implemented by passing the result of a substitution through one more round of substitution. The consequence is that extra layers of `\` may be needed for shielding. For example, the string `${name=?\\ \\ }` produces one `\` if the parameter **name** is empty. On the first pass, each pair of backslashes is turned into one backslash, and on the second pass, `\\` turns into a single backslash.

Special substitution syntax for Windows convenience:

<code>/\${value}</code> <code>\$/+{value}</code>	Replaces all the forward-slashes in the value with backslashes, for convenience of specifying the Windows paths that would otherwise require all slashes to be escaped.
<code> \${value}</code> <code> \${+}{value}</code>	Replaces all the <code>%</code> with <code>%%</code> as escaping for Windows.

If the resulting string is passed to shell or `cmd.exe` for execution, shell or `cmd.exe` performs its own substitution as well.

This example uses some of the more powerful substitution features to define the execution command as in the simple example. However, you may make use of the conditional features to support optional authentication and encryption, and an optional user-defined date format.

```
<Internal id="x_unixCmdExec"
  label="Execute Command"
  type="string"
  default="$ESP_HOME/bin/esp_convert
    ${platformSsl==1?-e}
    ${dateFormat}<>?-m '$dateFormat'}
    -c '${user=?user:$user}:$password'
    -p $platformCommandPort
    <"$directory/$filename" |
      $ESP_HOME/bin/esp_upload
      ${platformSsl==1?-e}
  -m $platformStream.$platformConnection
    -c '$user:$password'
  -p $platformCommandPort"
/>
```

Autogenerated Parameter Files

The basic external adapter framework, when started, writes its set of parameters (system and user-defined) to a parameter file.

This file is written in one of the following:

- Java properties
- Shell assignments
- Simple XML format

Commands then have full access to the parameter file.

There is an example of parameters in the `simplified_xml_input_plugin.cnxml` file.

```

    <Internal id="x_paramFile"
      label="Parameter File"
      type="string"
      default="/tmp/PARAMETER_FILE.txt"
    />
  <Internal id="x_paramFormat"
    label="Parameter Format"
    type="string"
    default="prop"
  />
  <Internal id="x_addParamFile"
    label="Add Parameter File"
    type="boolean"
    default="false"
  />
```

CHAPTER 4: Adapter Integration Framework

The parameter file is written to `/tmp/PARAMETER_FILE.txt`.

```
directory=/home/sjk/work/aleri/cimarron
/branches/3.1/examples/input/xml_tables
filename=trades.xml
platformAuth=none
platformCommandPort=31415
platformConnection=Connection1
platformHost=sjk-laptop
platformSqlPort=22200
platformSsl=0
platformStream=Trades
```

or a full list of parameters, in the Java properties format. You can specify the format as `shell` for shell assignments, or as `xml` for a simple XML format.

When `x_addParamFile` is specified as true, as shown here:

```
<Internal id="x_addParamFile"
  label="Add Parameter File"
  type="boolean"
  default="true"
/>
```

the argument `/tmp/PARAMETER_FILE.txt` is added to all commands before they execute.

configFilename Parameter

The **configFilename** parameter enables you to specify user-editable configuration files in the Studio.

If you create a user-defined **configFilename** parameter, clicking in the value portion of this field in Studio produces a file selector dialog, allowing you to choose a file on the local file system. Right-clicking the read-only name shows you a different dialog, allowing you to modify file contents. This lets you specify user-editable configuration files.

A comprehensive list of datatypes you can use with adapters supplied by Event Stream Processor, or any custom internal or external adapters you create.

Some exceptions for custom external adapters are noted in the datatype descriptions.

Note: This table includes all the adapter-related datatypes that are supported by Event Stream Processor. For more information about the specific datatypes that are supported by each adapter, as well as for its datatype mapping descriptions, see the section on that adapter.

Datatype	Description
<code>bigdatetime</code>	Stores the time of day (in microsecond precision) and the date. The default format is YYYY-MM-DD HH:MM:SS.SSSSS.
<code>bigint</code>	An alias for <code>long</code> .
<code>boolean</code>	Value is true or false. The format for values outside of the allowed range for <code>boolean</code> is 0/1/false/true/y/n/on/off/yes/no, which is case-insensitive.
<code>choice</code>	A list of custom values from which a user selects one value.
<code>configFilename</code>	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes.
<code>date</code>	Stores the date.
<code>decimal</code>	Numbers that contain decimal points. Accepts two mandatory parameters, <code>precision</code> and <code>scale</code> , which determine the range of values that can be stored in a decimal field. <code>precision</code> specifies the total number (from 1 to 34) of digits that can be stored. <code>scale</code> specifies the number of digits (from 0 to <code>precision</code>) that can be stored to the right of the decimal point.
<code>directory</code>	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes.
<code>double</code>	An alias for <code>float</code> .

APPENDIX A: Adapter Parameter Datatypes

Datatype	Description
filename	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes.
float	Floating point value. The range of allowed values is 2.22507e-308 to 1.79769e+308
int	A signed 32-bit integer value. The range of allowed values is -2147483648 to +2147483647 (-2^{31} to $2^{31}-1$). Constant values that fall outside of this range are automatically processed as long datatypes. To initialize a variable, parameter, or column with the lowest negative value, specify (-2...7) -1 instead to avoid CCL compiler errors. For example, specify (-2147483647) -1 to initialize a variable, parameter, or column with a value of -2147483648.
long	A signed 64-bit integer. The range of allowed values is -9223372036854775808 to +9223372036854775807 (-2^{63} to $2^{63}-1$). To initialize a variable, parameter, or column with a value of -9223372036854775808, specify (-9223372036854775807) -1 to avoid CCL compiler errors.
password	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes. While entering a value for this field, user sees '*' for every character.
permutation	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes. This datatype is not supported for custom external adapters.
query	A string value that Studio creates from the table name. This datatype is not supported for custom external adapters.
range	An integer value for which user can define lower and upper limits. For example, <pre><Parameter id="port" label="KDB Port" descr="IP port of the database listener" type="range" rangeLow="0" rangeHigh="65535" default="5001" use="required" /></pre>

APPENDIX A: Adapter Parameter Datatypes

Datatype	Description
<code>runtimeDirectory</code>	<p>Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but no less than 65535 bytes.</p> <p>This datatype is not supported for custom external adapters.</p>
<code>runtimeFilename</code>	<p>Runtime file name, if different from discovery time file name.</p> <p>Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but no more than 65535 bytes.</p> <p>This datatype is not supported for custom external adapters.</p>
<code>string</code>	<p>Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but no more than 65535 bytes.</p>
<code>tables</code>	<p>List of choices returned by the getTables() that are defined in the adapter.</p>
<code>text</code>	<p>A value capable of storing multiline text.</p> <p>This datatype is not supported for custom external adapters.</p>
<code>time</code>	<p>Stores the time of day as a two-byte field having a range of 00:00:00 to 23:59:59. The default format is HH24:MM:SS.</p>
<code>timestamp</code>	<p>Stores the date and the time of day (in millisecond precision). The default format is YYYY-MM-DD HH:MM:SS.SSS.</p>
<code>uint</code>	<p>Positive integer value. The range of allowed values is 0 to 0xffffffff.</p>

Date and Timestamp Formats for Input Adapters

SAP supports numerous formats for date and timestamp datatypes.

Use the info below to create a custom format for your date and timestamp datatypes.

Character	Description
%a	The day of the week, using the locale's weekday names. You can specify either the abbreviated or full name.
%A	Equivalent to %a.
%b	The month, using the locale's month names. You can specify either the abbreviated or full name.
%B	Equivalent to %b.
%c	The locale's appropriate date and time representation.
%C	The century number [00,99]. Leading zeros are permitted but not required.
%d	The day of the month [01,31]. Leading zeros are permitted but not required.
%D	The date as %m / %d / %y.
%e	Equivalent to %d.
%h	Equivalent to %b.
%H	The 24-hour clock [00,23]. Leading zeros are permitted but not required.
%I	The 12-hour clock [01,12]. Leading zeros are permitted but not required.
%j	The day number of the year [001,366]. Leading zeros are permitted but not required.
%m	The month number [01,12]. Leading zeros are permitted but not required.
%M	The minute [00,59]. Leading zeros are permitted but not required.
%n	Any white space.
%p	The locale's equivalent of a.m. or p.m.
%r	12-hour clock time using AM/PM notation.

APPENDIX B: Date and Timestamp Formats for Input Adapters

Character	Description
%R	The time as %H : %M.
%S	The seconds [00,60]. Leading zeros are permitted but not required.
%t	Any white space.
%T	The time as %H : %M : %S.
%U	The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. Leading zeros are permitted but not required.
%w	The weekday as a decimal number [0,6], with 0 representing Sunday. Leading zeros are permitted but not required.
%W	The week number of the year (Monday as the first day of the week) as a decimal number [00,53]. Leading zeros are permitted but not required.
%x	The date, using the locale's date format.
%X	The time, using the locale's time format.
%y	The year within the century. When a century is not otherwise specified, values in the range [69,99] refer to years 1969 to 1999 inclusive, and values in the range [00,68] refer to years 2000 to 2068 inclusive. Leading zeros are permitted but not required.
%Y	The year, including the century. For example, 1988.
%%	Replaced by %.

APPENDIX C **Date and Timestamp Formats for Output Adapters**

SAP supports numerous formats for date and timestamp datatypes.

Use the info below to create a custom format for your date and timestamp datatypes.

Character	Description
%a	The locale's abbreviated weekday name.
%A	The locale's full weekday name.
%b	The locale's abbreviated month name.
%B	The locale's full month name.
%c	The locale's appropriate date and time representation.
%C	The year divided by 100, and truncated to an integer as a decimal number [00,99].
%d	The day of the month as a decimal number [01,31].
%D	Equivalent to %m / %d / %y.
%e	The day of the month as a decimal number [1,31]. A single digit is preceded by a space.
%F	Equivalent to %Y - %m - %d. This is the ISO 8601:2000 standard date format.
%g	The last 2 digits of the week-based year, as a decimal number [00,99].
%G	The week-based year as a decimal number. For example, 1977.
%h	Equivalent to %b.
%H	The 24-hour clock as a decimal number [00,23].
%I	The 12-hour clock as a decimal number [01,12].
%j	The day of the year as a decimal number [001,366].
%m	The month as a decimal number [01,12].
%M	The minute as a decimal number [00,59].
%n	A newline character.

APPENDIX C: Date and Timestamp Formats for Output Adapters

Character	Description
%p	The locale's equivalent of either a.m. or p.m.
%r	The time in a.m. and p.m. notation.
%R	The time in 24-hour notation (%H : %M).
%S	The second as a decimal number [00,60].
%t	A tab character.
%T	The time in the format %H : %M : %S.
%u	The weekday as a decimal number [1,7], with 1 representing Monday.
%U	The week number of the year as a decimal number [00,53]. The first Sunday of January is the first day of week 1, and days in the new year before this are in week 0.
%V	The week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1.
%w	The weekday as a decimal number [0,6], with 0 representing Sunday.
%W	The week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1, and days in the new year before this are in week 0.
%x	The locale's appropriate date representation.
%X	The locale's appropriate time representation.
%y	The last two digits of the year as a decimal number [00,99].
%Y	The year as a decimal number. For example, 1997.
%z	The offset from UTC in the ISO 8601:2000 standard format (+hhmm or -hhmm), or by no characters if no time zone is determinable. For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich).
%Z	The time zone name or abbreviation, or by no bytes if no time zone information exists.
%%	Replaced by %.

APPENDIX D **Internal Adapter API**

If the adapters that are available from SAP do not meet your needs, Event Stream Processor provides an internal adapter API that you can use to build internal adapters. However, with the addition of the ESP adapter toolkit, the internal API is being deprecated and may be removed from future versions. SAP strongly recommends that you use the ESP adapter toolkit instead.

As part of creating a custom internal adapter, you must also create a custom adapter library. This library uses the Event Stream Processor shared utility library to help convert external data to ESP Server format.

You can implement life cycle and information management functions for your custom adapter using the C APIs in the adapter shared utility library. You can use the C interface to implement your custom adapter in C or C++ without any compiler restrictions.

The header file, `GenericAdapterInterface.h`, contains the import declarations that are required to call utility functions in the adapter shared utility library.

The Adapter Shared Utility Library

The adapter shared utility library includes the utility functions required for a custom adapter implementation: data conversion, data manipulation, and data management.

The header file, `GenericAdapterInterface.h`, contains the import declarations required to call utility functions in the adapter shared utility library.

When calling functions, each data utility requires a unique handle. The adapter shared utility library is labeled `esp_adapter_util_lib.dll` for Windows installations, and `libesp_adapter_util_lib.so` for Linux installations.

When calling functions in the adapter shared utility, each data utility requires a unique handle. For example, you can use the **ConnectionRow** function by calling **CreateConnectionRow**. This call returns a unique handle in the form of a void pointer. You can then pass this pointer back when making calls to any other APIs under **ConnectionRow**.

Sample Model File

Sample syntax you can use to build a basic model file.

This model represents a schema with two columns of string data. The model also defines an input connection that references a sample custom adapter implementation.

APPENDIX D: Internal Adapter API

```
CREATE MEMORY STORE "memory" PROPERTIES INDEXTYPE ='tree',
INDEXSIZEHINT =8;

CREATE INPUT WINDOW Custom
SCHEMA (column1 STRING, column2 STRING)
PRIMARY KEY (column1)
STORE "memory";
ATTACH INPUT ADAPTER Connection1
TYPE custom_in
TO Custom
PROPERTIES RowCount=15;
```

The Adapter Configuration File

The internal adapter configuration file is an XML file (.cnxml) that contains the properties and commands used by ESP to start and stop the internal adapter, as well as other information that allows you to configure the internal adapter from the ESP Studio.

The adapter configuration file also constructs the name for your custom adapter DLL file. The library name is referenced when you load your adapter.

This is sample code for naming the custom adapter DLL file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Adapter type="input"
  id="custom_in"
  label="Custom Input"
  descr="Dummy Custom Input Adapter"
>
<Library file="custom_in" type="binary"/>
<Special>
</Special>
<Section>
  <Parameter id="RowCount"
    label="RowCount"
    descr="How many rows to generate"
    type="uint"
    use="required"
    default="10"/>
</Section>
</Adapter>
```

Note: The .cnxml file adheres to the Adapter.xsd file.

Once the .cnxml and custom adapter DLL files are ready, copy them to the ESP_HOME/lib/adapters folder.

DLL Export Functions

The DLL export functions define how a custom internal adapter controls its life cycle and communicates with the ESP Server. Custom adapter implementation is bundled in a separate DLL. When building a custom internal adapter, an adapter developer must implement the DLL Export functions found in the DLL.

Adapter Setup Functions

The ESP Server uses information management functions to complete the adapter implementation process.

These functions are used for both input and output adapters.

API	Description
void* createAdapter();	Returns a unique handle that the ESP Server uses to make subsequent calls to the adapter. This is the first call the ESP Server makes when creating an adapter.
void* deleteAdapter(void* adapter);	Deletes an adapter using the unique handle returned by the <code>createAdapter</code> API. This is the last call the ESP Server makes concerning a given adapter.
void setCallbackReference(void* adapter, void* connectionCallbackReference);	Gives the adapter implementation a unique handle that corresponds to the <code>connectionCallback</code> object on the ESP Server side. Use this handle as a parameter when making callbacks to the ESP Server.
void setConnectionRowType(void* adapter, void* connectionRowType);	Provides the adapter implementation with information related to schema.
void setConnectionParams(void* adapter, void* connectionParams);	Provides the adapter implementation with information related to connection parameters.

Adapter Life Cycle Functions

All adapters follow a set of adapter life cycle events.

API	Description
bool reset(void* adapters);	Initializes both input and output adapters.

API	Description
void start(void* adapter);	Processes data that is specific to the adapter implementation. Call this API immediately after resetting both input and output adapters.
void* getNext(void* adapter);	Reads data and returns a pointer to the data in a format the ESP Server understands. The adapter shared utility library provides data conversion functions. Call this API for input adapters only.
void putNext(void* adapter,void* stream);	Converts presented data into a format the ESP Server understands, and writes it according to the specified adapter implementation. The adapter shared utility library provides data conversion functions. Call this API for output adapters only.
void stopRequested(void* adapter);	Sets a flag to indicate that a request has been sent to stop the adapter. Call this API for both input and output adapters.
void stop(void* adapter);	Stops an adapter. Call this API for both input and output adapters.
void cleanup(void* adapter);	Performs cleanup activities after an adapter is stopped.
void commitTransaction(void* adapter);	Notifies an output adapter when a transaction ends and pushes any buffered data into the target. Call this API for output adapters only.
void putStartSync(void* adapter);	Notifies the adapter implementation that the base data is being sent. Call this API for output adapters only.
void putEndSync(void* adapter);	Notifies the adapter implementation that the base data has been sent. Call this API for output adapters only.
void purgePending(void* adapter);	Instructs an output adapter to purge pending data. Call this API for output adapters only.
bool isOutBase(void* adapter);	Determines whether the adapter expects to receive the output for the base contents of the stream. Call this API for output adapters only.

Miscellaneous Functions

Descriptions of miscellaneous APIs supported for Event Stream Processor adapters.

API	Description
int64_t getNumberOfGoodRows(void* adapter);	Retrieves information from the adapter implementation about the number of good rows processed by the adapter.
int64_t getNumberOfBadRows(void* adapter);	Retrieves information from the adapter implementation about the number of bad rows processed by the adapter.
int64_t getTotalRowsProcessed(void* adapter);	Retrieves information from the adapter implementation about the total number of rows processed by the adapter.
bool canDiscover(void* adapter);	Retrieves information from the adapter implementation about whether it supports schema discovery functionality.
bool hasError(void* adapter);	Retrieves information from the adapter implementation about whether there were any errors during the processing of data.
void getError(void* adapter, char**errorString);	Retrieves error information from the adapter implementation.
void getStatistics(void* adapter, AdapterStatistics* adapterStatistics)	Retrieves custom statistics information from an adapter. The Server uses this to periodically update the <code>_esp_adapter_statistics</code> metadata stream. Enable the <code>time-granularity</code> project option to update the <code>_esp_adapter_statistics</code> metadata stream. The adapter stores its statistics in key value format within the <code>AdapterStatistics</code> object. The <code>AdapterStatistics</code> object is populated using the void addAdapterStatistics(void* adapterStatistics, const char* key, const char* value) API, which is available in the adapter utility library.

API	Description
<code>int64_t getLatency(void* adapter)</code>	Retrieves latency information from the adapter, in microseconds, and uses this information to periodically update the latency column in the <code>_esp_connectors</code> metadata stream. Enable the time-granularity project option to update the <code>_esp_connectors</code> metadata stream.

Schema Discovery for Internal Custom Adapters

Use extern C functions to enable your custom internal adapter for schema discovery.

Method	Description
<code>extern "C" DLLEXPORT bool canDiscover(void* adapter)</code>	Checks whether an adapter supports discovery. This is the first API called when an adapter is running schema discovery. Adapters that support discovery return a value of true.
<code>extern "C" DLLEXPORT void setDiscovery(void* adapter)</code>	Tells the adapter that it is running in discovery mode.
<code>extern "C" DLLEXPORT int getTableNames(void* adapter, char*** tables)</code>	Points to an array of strings that populates tables. The function returns table names when discovering tables in a database, or file names when discovering particular types of files in a directory.
<code>extern "C" DLLEXPORT int getFieldNames(void* adapter, char*** names, const char* tableName)</code>	Retrieves field names.
<code>extern "C" DLLEXPORT int getFieldTypes(void* adapter, char*** types, const char* tableName)</code>	Retrieves field types.
<code>extern "C" DLLEXPORT int getSampleRow(void* adapter, char*** row, const char* tableName, int pos)</code>	Retrieves sample rows.

DLL Import Functions

A custom internal adapter can import DLL import functions, which provide useful functionalities for implementing a custom adapter.

The DLL import functions are various API import declarations listed in the `GenericAdapterInterface.h` file. These APIs are all optional.

Error Handler Functions

Describes various error handler utility APIs used by custom internal adapters.

These functions are used for both input and output adapters.

API	Description
void* createConnectionErrors();	Creates a ConnectionsErrors object and its related ConnectionErrorsWrapper object.
bool empty(void* connectionErrors);	Checks whether two parameters pair without errors.
size_t sizeConnectionErrors(void* connectionErrors);	Measures the size of a parameter pair and its error message and returns a measurement sum.
void clearConnectionErrors(void* connectionErrors);	Clears the parameter pair created by the ConnectionErrors object and any error messages generated by the pair.
void addParam(void* connectionErrors, const char *param, const char *error);	Adds a parameter-value pair into the ConnectionErrors object.
void addGeneral(void* connectionErrors, const char* error);	Adds a general error message into the ConnectionErrors object.
void getAdapterError(void* connectionErrors, char** errorString);	Retrieves the stored parameter-value pair and the general error message created by the addParam and addGeneral APIs and places them into the errorString object.

Parameter Handler Functions

Describes various parameter handler utility APIs used by custom internal adapters.

These functions are used for both input and output adapters.

API	Description
void addConnectionParam(void* connectionParam, const char* key, const char* val);	Populates the ConnectionParam object with a key-value pair.
int getConnectionParamInt64_t(void* connectionParams, const char* key);	Retrieves a parameter value as a long datatype.

API	Description
<code>const char* getConnectionParamString(void* connectionParams, const char* key);</code>	Retrieves a parameter value as a <code>string</code> data-type.
<code>const char* substitute(void* connectionParam, const char *where, const char *paramPrefix="", int depth=0);</code>	Substitutes the specified environment variables and parameters in <code>string</code> format.
<code>DLLIMPORT int getConfiguredSections(void* connectionParams);</code>	Determines the number of existing configuration sections.

Data Conversion Functions

The ESP Server uses these APIs to convert data within the custom internal adapter implementation.

These functions are used for both input and output adapters.

API	Valid Row-Type	Description
<code>void* createConnectionRow(const char *type);</code>	All	Creates a <code>ConnectionRow</code> object of a given type and its related <code>ConnectionRowWrapper</code> object. The valid values for <code>type</code> are: <ul style="list-style-type: none"> • RowByName – stores data internally as name-value pairs. • RowByOrder – stores data internally as an indexed vector.
<code>void deleteConnectionRow(void* connectionRow);</code>	All	Deletes a <code>ConnectionRow</code> object and its related <code>ConnectionRowWrapper</code> object.
<code>void clear(void* connectionRow);</code>	All	Clears the contents of a <code>ConnectionRow</code> object and prepares the object for reuse. If the <code>ConnectionRowType</code> is owned by the given <code>ConnectionRow</code> , all data, including the type, is cleared. If the <code>ConnectionRowType</code> is not owned by the given <code>ConnectionRow</code> , only the internal data store is cleared and the type remains the same. The ownership of a type is decided by the parameter <code>own</code> , located in the API <code>setStreamType</code> .

API	Valid Row-Type	Description
<code>void clearData(void* connectionRow);</code>	All	Clears all data from the internal data store of a <code>ConnectionRow</code> object, leaving only the stream information.
<code>void* createNew(void* connectionRow);</code>	All	Creates a new <code>ConnectionRow</code> object of the same subtype as the one specified.
<code>void setDateFormat(void* connectionRow, const char *fmt);</code>	All	Sets the date format using a representative string.
<code>void setTimestampFormat(void* connectionRow, const char *fmt);</code>	All	Sets the timestamp format using a representative string.
<code>void setStreamName(void* connectionRow, const char *sname);</code>	All	Specifies a unique name for the stream.
<code>const char* getStreamName(void* connectionRow);</code>	All	Retrieves the name of the given stream.
<code>void setStreamType(void* connectionRow, void *connectionRowType, bool own);</code>	All	Specifies a parsed stream type and sends stream row definition information contained in the <code>ConnectionRowType</code> object to the <code>ConnectionRow</code> object. To apply the row schema, the ESP Server calls this API after the <code>ConnectionRow</code> object is created. The own parameter specifies ownership of a type object and therefore deletion responsibility, as seen in the <code>clear</code> API.
<code>void *getStreamType(void* connectionRow);</code>	All	Retrieves stream row definition information contained in the <code>ConnectionRowType</code> object.
<code>bool getFieldAsStringWithKey(void *connectionRow, const char *fname, const char **val);</code>	Row-By-Name	Retrieves a column value from the <code>ConnectionRow</code> object using the given column name.
<code>bool setFieldAsStringWithKey(void *connectionRow, const char *fname, const char* val);</code>	Row-By-Name	Updates the value of the column with the specified name, or creates a new column with the specified name and value.

API	Valid Row-Type	Description
bool getFieldAsStringWithIndex(void* connectionRow, int fn, const char **val);	Row-ByOrder	Retrieves a column value from the ConnectionRow object using the column index.
bool setFieldAsStringWithIndex(void* connectionRow, int fn, const char *val);	Row-ByOrder	References a column index and updates or appends the relevant column.
bool appendFieldAsString(void* connectionRow, const char *val);	Row-ByOrder	Appends a new column to the given ConnectionRow object and populates it with the given value.
char getOp(void* connectionRow);	Row-ByOrder	Retrieves operation type information.
void setOp(void* connectionRow, char op);	Row-ByOrder	Sets the operation type of the ConnectionRow object. Valid values are: <ul style="list-style-type: none"> • i or I (insert) • d or D (delete) • u or U (update) • p or P (upsert)
const char *getFlagsAsString(void* connectionRow);	Row-ByOrder	Retrieves flag information in string format.
bool setFlagsAsString(void* connectionRow, const char * flags);	Row-ByOrder	Sets up the flag information in string format and subscribes with shine through, if possible. Therefore, when an update contains no new data for previously received fields, information for those fields is retained.
int size(void* connectionRow);	Row-ByOrder	Determines the number of existing fields.
void* toRow(void* connectionRow, size_t rowNo, void* connectionErrorMsgs);	All	Converts an existing ConnectionRow into an ESP StreamRow.

API	Valid Row-Type	Description
bool fromRow(void* connectionRow, void* streamRow, size_t rowNo, void* connectionErrorMsg);	All	Converts an existing ESP StreamRow into a ConnectionRow.

Callback Functionality

An adapter implementation can use callback functions to log on to the ESP Server, retrieve information related to schemas, and convey state information to the ESP Server.

Callback functions are contained in the `esp_server_lib.lib` file for Windows installations, and in the `libesp_server_lib.so` file for Linux and Solaris installations.

These functions are used for both input and output adapters, unless otherwise noted.

API	Description
void postEndSync(void* callBackReference);	Notifies the adapter implementation that the base data flow has ended. Call this API for output adapters only.
void logMessage(void* callBackReference,int level,const char* message);	Logs a message with the adapter implementation. The valid values for the the severity level are: <ul style="list-style-type: none"> • L_EMERG • L_ALERT • L_CRIT • L_ERR • L_WARNING • L_NOTICE • L_INFO • L_DEBUG
void postStartSync(void* callBackReference);	Notifies the adapter implementation that the base data is about to be sent. Call this API for output adapters only.

API	Description
void notifyConnState(void* callBackReference, int oldSt, int st);	Notifies the ESP Server that the adapter state has changed. The valid values for adapter state are: <ul style="list-style-type: none"> • RS_READY - ready to start. • RS_INITIAL - performing start-up activities. • RS_CONTINUOUS - ready to continuously receive data. • RS_IDLE - timeout. • RS_DONE - no more incoming data, ready to exit. • RS_DEAD - stopped receiving data, is inactive.
int getColumnCount(void* connectionRowType);	Retrieves the number of existing columns of a given row type from the adapter implementation.
const char* getColumnName(void* connectionRowType, int pos);	Retrieves a column name using the column's position in the adapter implementation.
void* getTimeContext(void* callBackReference);	Retrieves information contained in the Time-Context object from ESP Server.
const char* getStreamName(void* connectionRowType);	Retrieves the name of the stream specified by the given ConnectionRowType.
void deactivateOutput(void* callBackReference);	Tells a stream to temporarily suspend output data flow.
void activateOutput(void* callBackReference, bool sendBase);	Requests the stream to start sending output data. This is indicated by the sendBase flag.
void setAdapterState(void* callBackReference, int state);	Sets up a running state for the adapter implementation.
int getAdapterState(void* callBackReference);	Retrieves running state information from the adapter implementation.
int getColumnDatatype(void* callBackReference, int pos);	Retrieves the type of a field using the field's position in the CallBackReference object.
void addAdapterStatistics(void* adapterStatistics, const char* key, const char* value);	Populates the AdapterStatistics object with a custom statistics item in key-value format.

Adapter Run States

Adapters progress through a set of run states (RS) as they interact with Event Stream Processor.

- **RS_READY** – indicates that the adapter is ready to be started.
- **RS_INITIAL** – indicates that the adapter is performing start-up and initial loading. An adapter enters the **RS_INITIAL** state when the reset function is called.
- **RS_CONTINUOUS** – indicates that the adapter is continuously waiting for additional data. If the **RS_CONTINUOUS** state is set in the reset method, input adapters return from reset with data to process, and output adapters return from reset prepared to accept data.
- **RS_IDLE** – indicates that the adapter has timed out or is attempting to restore a broken socket.
- **RS_DONE** – indicates when the adapter no longer returns data and can no longer retrieve data following the poll period.
- **RS_DEAD** – indicates that the adapter has entered the exited state. The adapter does not operate until you call the restart function.

When polling is enabled, an input adapter may change states from **RS_CONTINUOUS** to **RS_IDLE**. Change the adapter state back to **RS_CONTINUOUS** to retry data retrieval after a certain amount of time.

Sample Custom Internal Adapter Implementation

Event Stream Processor ships with sample syntax you can use to build your custom internal adapter implementation. This implementation incorporates extern C methods that enable schema discovery in a custom adapter.

See the sample custom internal adapter for UNIX:

```
% cd $ESP_HOME/examples/cpp/adapter

% make -f Makefile.unix
g++ -fPIC -m64 -I $ESP_HOME/include/adapter -c -o
adapter_custom_in.o adapter_custom_in.cpp
g++ adapter_custom_in.o -m64 -shared -L $ESP_HOME/lib -L $ESP_HOME/
lib/adapters -lesp_adapter_util_lib -lesp_server_lib -o
libesp_adapter_custom_in_lib.so
g++ -fPIC -m64 -I $ESP_HOME/include/adapter -c -o
adapter_custom_out.o adapter_custom_out.cpp
g++ adapter_custom_out.o -m64 -shared -L $ESP_HOME/lib -L $ESP_HOME/
lib/adapters -lesp_adapter_util_lib -lesp_server_lib -o
libesp_adapter_custom_out_lib.so
```

See the sample custom internal adapter for Windows:

APPENDIX D: Internal Adapter API

```
%ESP_HOME%\examples\cpp\adapter>cd %ESP_HOME%\examples\cpp\adapter  
%ESP_HOME%\examples\cpp\adapter>nmake -f Makefile.win
```

Index

A

- adapter
 - log4j API 5
 - logging 5
- adapter configuration 130
- adapter configuration file 75
- adapter examples 91
 - schema discovery 92
- adapter integration framework 109
- adapter property sets
 - creating 4
 - editing 4
- adapter toolkit 9
 - adapter example 91
 - adapter example, schema discovery 92
 - API reference 13
 - build custom formatter 58
 - build custom transporter 37
 - CSV String to ESP formatter 43
 - debugging an adapter 94
 - ESP Publisher 69
 - ESP Subscriber 66
 - ESP to CSV String formatter 44
 - ESP to JSON Stream formatter 45
 - ESP to Object List formatter 47
 - ESP to String List formatter 47
 - ESP to XML String formatter 48
 - ESP to XMLDoc String formatter 48
 - EspConnector modules 66
 - examples 90
 - File Input transporter 17
 - File Output transporter 20
 - formatters 40
 - FTP Input transporter 21
 - FTP Output transporter 23
 - HTTP Output transporter 24
 - JDBC Input transporter 27
 - JDBC Output transporter 29
 - JMS Input transporter 31
 - JMS Output transporter 33
 - JSON Stream to JSON String formatter 51
 - JSON String to ESP formatter 49
 - Multistream Publisher 71
 - Multistream Subscriber 67
 - Object List to ESP formatter 51
 - preconfigured adapters 10
 - sample cnxml file, input 93
 - sample cnxml file, output 93
 - Socket Input transporter 34
 - Socket Output transporter 36
 - standard formatters 40
 - standard transporters 14
 - Stream to Stream formatter 53
 - Stream to String formatter 52
 - String List to ESP formatter 54
 - transporters 14
 - XML String to ESP formatter 54
 - XMLDoc Stream to ESP formatter 54
- adapters
 - adapter shared utility library 129
 - adapter utilities 129
 - adding a new property set 4
 - ATTACH ADAPTER statement 3
 - attaching an adapter 3
 - callback functions 139
 - configuring property sets 4
 - creating a custom adapter 12
 - custom internal 129
 - data conversion functions 136
 - debugging a custom adapter 94
 - DLL export functions 131
 - information management functions 131
 - life cycle functions 131
 - managed 2
 - miscellaneous functions 133
 - overview 1
 - parameter datatypes 121
 - parameter handler functions 135
 - preconfigured adapters 10
 - publishing data 3
 - run states 141
 - schema discovery 62
 - starting 86
 - stopping 88
 - subscribing to data 3
 - unmanaged 2
- Adapters
 - error handler functions 135
- API reference
 - adapter toolkit 13

Index

ATTACH ADAPTER statement 3

B

batch processing 61

C

CCL statements

ATTACH ADAPTER statement 3

cnxml configuration file

example 112

cnxml file

custom internal adapters 130

cnxml files

sample, input adapter 93

sample, output adapter 93

configuration

adapter configuration file 75

configuring a new adapters 75

CSV String to ESP formatter 43

ESP Publisher 69

ESP Subscriber 66

ESP to CSV String formatter 44

ESP to JSON Stream formatter 45

ESP to Object List formatter 47

ESP to String List formatter 47

ESP to XML Stream formatter 48

ESP to XMLDoc String formatter 48

File Input transporter 17

File Output transporter 20

FTP Input transporter 21

FTP Output transporter 23

HTTP Output transporter 24

JDBC Input transporter 27

JDBC Output transporter 29

JMS Input transporter 31

JMS Output transporter 33

JSON Stream to JSON String formatter 51

JSON String to ESP formatter 49

Multistream Publisher 71

Multistream Subscriber 67

Object List to ESP formatter 51

property sets 4

Socket Input transporter 34

Socket Output transporter 36

Stream to Stream formatter 53

Stream to String formatter 52

String List to ESP formatter 54

XML String to ESP formatter 54

XMLDoc Stream to ESP formatter 54

configuration parameters

ESP connection parameters 73

creating a custom external adapter, SDK
guidelines 97

custom .Net external adapters

connecting to a subscriber 105

connecting to projects 104

connecting to the server 103

publishing data 104

subscribing using callback 105

custom adapters 109

build custom formatter module 58

build custom transporter module 37

creating a configuration file 75

debugging 94

custom adapters, creating 12

preconfigured adapters 10

custom C/C++ external adapters

creating authentication credentials 101

getting a project 101

publishing and subscribing 101

sample code for handleData 103

subscribing using callback 101

custom external adapters

.Net adapters 103

auto-generated parameter files 119

configFilename parameter 120

external adapter commands 116

external adapter configuration file 109

external adapter properties 114

parameter substitution 117

publish using callback 100

user-defined parameters 117

custom internal adapters 129

sample implementation 141

schema discovery 134

custom Java external adapters

connecting to projects 97

creating publishers 97

sample code for adding rows 98

subscribe using Direct Access mode 100

subscribing using callback 98

D

datatype formats for input adapters

date format 125

timestamp format 125

- datatype formats for output adapters
 - date format 127
 - timestamp format 127
- datatype mappings
 - ESP to Java 56
 - Java to ESP 56
- datatypes
 - adapter parameter datatypes 121
- date format 125, 127
- debugging an adapter 94
- DLL Import Functions 134

E

- envelopes 61
- ESP adapter toolkit 9
- EspConnector 66
 - ESP Publisher 69
 - ESP Subscriber 66
 - Multistream Publisher 71
 - Multistream Subscriber 67
- examples
 - adapter toolkit 90
 - adapters 91
 - schema discovery 92
- external data
 - input and output adapters 1

F

- formats for input adapters
 - date format 125
 - timestamp format 125
- formats for output adapters
 - date format 127
 - timestamp format 127
- formatters 40
 - CSV String to ESP formatter 43
 - ESP to CSV String formatter 44
 - ESP to JSON Stream formatter 45
 - ESP to Object List formatter 47
 - ESP to String List formatter 47
 - ESP to XML String formatter 48
 - ESP to XMLDoc String formatter 48
 - implementing schema discovery 63
 - JSON Stream to JSON String formatter 51
 - JSON String to ESP formatter 49
 - Object List to ESP formatter 51
 - Stream to Stream formatter 53

- Stream to String formatter 52
- String List to ESP formatter 54
- XML String to ESP formatter 54
- XMLDoc Stream to ESP formatter 54

G

- guaranteed delivery 64
 - input transporter, enabling 64

I

- input adapters
 - overview 1

J

- Java adapters
 - debugging 94

L

- life cycle functions 131
- logging
 - adapter 5
 - log4j API 5

M

- managed adapters 2
- mappings, datatypes
 - ESP to Java 56
 - Java to ESP 56
- model files 129
- modules
 - build custom formatter 58
 - build custom transporter 37
 - EspConnector 66
 - formatters 40
 - transporters 14

O

- output adapters
 - overview 1

Index

P

- property sets
 - creating 4
 - editing 4

S

- sample files
 - cnxml, input adapter 93
 - cnxml, output adapter 93
- schema
 - adapters 62
 - discovery 62
- schema discovery
 - adapter example 92
 - custom adapters, implementing 63
 - overview 62

SDK

- creating a custom external adapter 97

- standard formatters 40
- standard transporters 14
- starting an adapter 86
- stopping an adapter 88
- streams

- schema discovery 62

T

- timestamp format 125, 127

- transactions 61

transporters 14

- build custom formatter 58
- build custom transporter 37
- enabling guaranteed delivery 64
- File Input transporter 17
- File Output transporter 20
- FTP Input transporter 21
- FTP Output transporter 23
- guaranteed delivery 64
- HTTP Output transporter 24
- implementing schema discovery 63
- JDBC Input transporter 27
- JDBC Output transporter 29
- JMS Input transporter 31
- JMS Output transporter 33
- Socket Input transporter 34
- Socket Output transporter 36

U

- unmanaged adapters 2

W

- windows
 - schema discovery 62