HP Connect-It

Software version: 3.91

SDK

Document Release Date: 16 February 2009
Software Release Date: February 2009

Legal Notices

Table of Contents

Introduction

The Connect-It Development Kit enables you to develop and implement your own connectors. This development kit uses a Java interface based on the J2EE Connector Architecture (1.5) standard. The JCA standard defines a set of Java interfaces used to simplify the integration of enterprise applications (ERP, database applications, etc).

Who is this guide intended for?

This guide is destined for developers who have sufficient expertise in Java and the JCA standard. For more information about this standard, consult the following Web site: J2EE Connector Architecture [http://java.sun.com/j2ee/connector].

Terminology

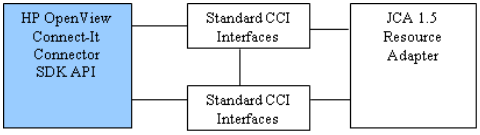
The following acronyms are used throughout this guide:

- JCA: J2EE Connector Architecture
- RA: Resource Adapter
- EIS: Enterprise Information System
- CCI: Common Client Interface

- SPI: Service Provider Interface
- JDBC: Java Database Connectivity

General information

The API defines an extension to the JCA 1.5 API which enables the connector to be integrated into the application. The following diagram shows how this works:



The communication mode with the connector depends on the information system (EIS) to which it is connected.

Two communication possibilities exist:

- Outbound communications (synchronous)

  The client initiates the data exchange. This occurs, for example, when a query is sent to a database.
- Inbound communications (asynchronous)

  The EIS initiates the data exchange. The connector is in listening mode. This is what takes place for messaging.

SPI Extensions

The SDK supplies an extension to the SPI classes to enable the support of metadata descriptions.

CCI Extensions

The SDK supplies a client layer that can manage access to a system, whether it be a relational database or not. This extension groups functions from the standard CCI API and the JDBC API.

1 Data exchange

The goal of a connector designed using the SDK is to standardize data exchange with information systems. Data exchanges include sending and receiving data.

Data and data types

Before data can be exchanged, the structure of the data must be known. This structure is what is called metadata. The SDK requires that the structure of the data be known before any operations are done using the data. This is done via two interfaces:

com.hp.ov.cit.connector.cci.ObjectRecord - represents a specific piece of data.

And

com.hp.ov.cit.connector.cci.ObjectType - represents the structure that a set of related data must have.

Since it is required to describe each piece of data that is sent or received, an *ObjectRecord* instance is linked to its *ObjectType* description.

Data supplied via a connector are generally organized within a hierarchy or graph. Their metadata is also hierarchical. Metadata is said to be 'complex' when it contains other metadata. The 'child' metadata make up the fields of the data. Metadata is said to be 'simple' when it does not contain other metadata. This metadata contains no fields.

An *ObjectRecord* graph is composed of:

- A single *ObjectRecord* root data item.
- Each of the *ObjectRecords* can be accessed by traversing the fields recursively.

Models

The SDK provides two distinct data models described by the *ObjectType*, *ObjectRecord* pair. These models are the *Class/Instance* model and the *XMLSchema/XML* model. Only one model is possible per connector.

Class/Instance model

This model is an object representation of a data structure. This model is based on the Java notions of class and instance.

Class

A class has a name and belongs to a package which forms its namespace. It is made up of fields that are associated with classes.

Within this model, a class makes up the metadata. It can be accessed via the *ObjectType* interface and has the following methods:

```
public String getName();
public String getNamespace();

public Class getObjectClass();
public boolean isSimple();
public Field getField(String fieldName);
public Field[] getFields();
```

A field accessed by the *com.hp.ov.cit.connector.cci.Field* interface contains its own information and the information that is related to its class. A class has the following characteristics:

- It can be modified
- It can have a default value
- It can appear several times and when it appears in a list it is described as being indexed
- It can be required to have a value

This is done through the *Field* interface via the following methods:

```
public String getName();
public ObjectType getType();
public Object getDefault();
public boolean isIndexed();
```

```
public boolean isReadOnly();
public boolean isRequired();
```

Instance

An instance is associated with a class and contains values for one or more of its fields.

Within this model, an instance forms a piece of data. It is represented via the *ObjectRecord* interface and has the following methods:

```
public Object get(String fieldName);
public Object get(String fieldName, int fieldIndex);
public void set(String fieldName, Object value);
public void set(String fieldName, int fieldIndex, Object value);
public void remove(String fieldName);
public void remove(String fieldName, int fieldIndex);
```

Simple types

The following table provides the list of Java simple types that are supported by the SDK.

java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
java.lang.String
java.util.Date
byte[]
char[]

Example

Consider the data model below:

```
A
|- String
|- int
|- B
|- String
|- C*
|- boolean
```

Business classes are thus represented as:

```
public class A
{
private String stringField = "This is a string";
private int intField;
```

```
private B bField;
}

public class B
{
private String stringField;
private List<C> listOfCField;
}

public class C
{
private boolean booleanField;
}
```

Operations on types can be done as follows:

```
ObjectType objectTypeA = ...;
Field field = objectTypeA.getField("stringField");
boolean isSimple = field.getType().isSimple(); // true
Object defaultValue = field.getDefault(); // "This is a string"
...
field = objectTypeA.getField("bField");
isSimple = field.isSimple(); // false
ObjectType objectTypeB = field.Type();
field = objectTypeB.getField("listOfCField");
boolean isIndexed = field.isIndexed(); //true;
```

Data can be stored in the following manner:

```
ObjectRecord objectA = ...;
ObjectRecord objectB = ...;
ObjectRecord objectC = ...;

objectA.set("intField", 5);
objectA.set("bField", objectB);

java.util.List<C> list = new java.util.ArrayList<C>();
list.add(objectC);
objectB.set("listOfCField", list);
```

XMLSchema/XML model

This representation model is adapted to systems handling XML data. Metadata is formed from a set of independent XML schemas. This model limits the use of the interfaces described above. In this case, the only pertinent methods of the *ObjectType* interface are:

```
public String getName();
public String getNamespace();

public boolean isXSD();
public org.w3c.dom.ls.LSInput[] getXSD();
```

This model also supposes that metadata identified by its name and namespace is always simple. This means that it cannot contain any fields, and may only contain one or more XML schemas.

Data itself can be accessed in the *ObjectRecord* interface via the following methods:

```
public void readXML(org.w3c.dom.ls.LSInput input);
public void writeXML(org.w3c.dom.ls.LSOutput output);
```

These methods enable the XML representation to be imported or exported:

- org.w3c.dom.ls.LSInput - represents an input source for the XML data.
- org.w3c.dom.ls.LSOutput - represents an output source for the XML data.

2 Design-Time

This section describes elements that are used by a connector to connect to an EIS and discover its metadata.

DesignTimeFactory interface

The *com.hp.ov.cit.connector.spi.designtime.DesignTimeFactory* interface centralizes all the information required to:

- Obtain a connection
- Describe the structure of the data exchanges with the EIS

Communication mode

The methods

public boolean supportsOutbound()

And

public boolean supportsInbound()

are used to determine the communication mode used by the EIS. Within Connect-It these two modes are exclusive. A connector implementation can only support one mode at a time.

Design-time connection

Data exchange types must be described regardless of the communication mode. To do this a connection is used, whether it be a real one or not. For outbound communications it is also possible that this connection be different from the connection that is used for the data exchange itself. For example, in the case of a web service, metadata is described using a WSDL file that can be accessed via an FTP connection whereas communication with the web service is done using the http protocol.

The API of the *DesignTimeFactory* class provides the following methods:

- **public boolean requiresSeparateMetaDataConnection()**

  Determines if the EIS distinguishes between the two connection types. This method is not used for inbound communications.

- **public javax.resource.cci.ConnectionSpec createMetaDataConnectionSpec()**

  This method returns a JavaBean implementation of the *ConnectionSpec* interface. The object contains client-specific information such as "user" and "password" that are used to connect during the design-time phase. The method involved for outbound communications that do not differentiate design-time connections from run-time connections is:

- **public javax.resource.cci.ConnectionSpec createConnectionSpec()**

For example, the url must be known if metadata is accessed via an http connection:

```
package com.myeis;

import java.net.URL;
import javax.resource.cci.ConnectionSpec;

public class MyEISConnectionSpec implements ConnectionSpec
{
private URL url;

public URL getUrl()
{
return url;
}

public void setUrl(String url)
{
this.url = url;
}
}
```

Once the connection information is retrieved, the metadata can be described.

Retrieving metadata

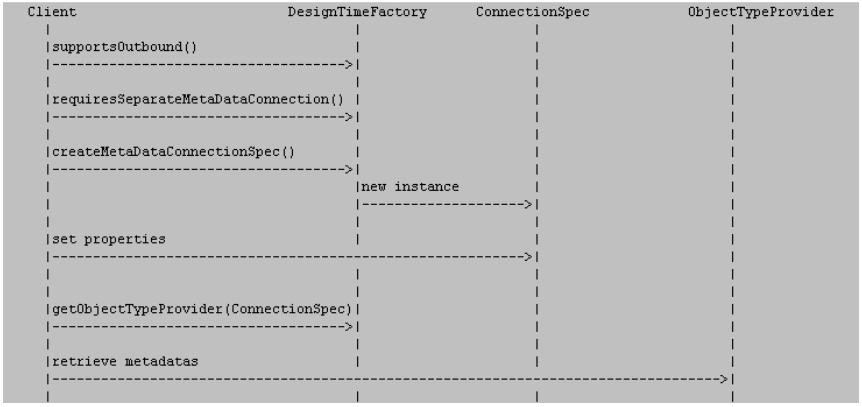
The method

public ObjectTypeProvider getObjectTypeProvider(javax.resource.cci.ConnectionSpec metaDataConnSpec)

returns an object which is used to obtain the description of the data that is exchanged with the EIS. The required connection information makes up its parameters.

Example

The operations described above are shown in the following diagram. The diagram shows an outbound communication which requiring specific connection to access the metadata:



A client queries the **DesignTimeFactory** to see if the connector supports the outbound communication. If it does, it queries to see if the connection to the metadata is distinct from the connection used to exchange data. Depending on the response, the client will either call the *createMetaDataConnectionSpec* method or the *createConnectionSpec* method in order to retrieve a connection's description. The client then sets the properties of the method and calls the *DesignTimeFactory* to retrieve the *ObjectTypeProvider* which is used to describe the metadata.

ObjectTypeProvider interface

The *com.hp.ov.cit.connector.spi.designtime.ObjectTypeProvider* interface is used to describe EIS data types. This description may be infinite. For example, an A data type may contain a B data type itself containing an A data type. To avoid recursion problems, instead of describing data types in one block, the interface is based on a navigable model. This makes it possible to find first level data first. Then, as subsequent calls are made to the interface, the other levels of data can be described. As these types are retrieved via a connection, calling the method:

public void close()

closes the connection.

The following methods are used to describe first-level metadata:

```
public java.util.List<ObjectType> getReceivedTypes();
public java.util.List<ObjectType> getRequestTypes();
public java.util.List<ObjectType> getResponseTypes();
```

Depending on the EIS type and communication mode (inbound or outbound), these methods will need to be supported or not supported. Supported methods are implemented as follows:

```
public java.util.List<ObjectType> getXXXTypes()
{
java.util.List<ObjectType> types = new java.util.ArrayList<ObjectType>();
types.add(new MyEISObjectType());
...
return types;
}
```

For unsupported methods:

```
public java.util.List<ObjectType> getXXXTypes() throws javax.resource.NotS
upportedException
{
throw new javax.resource.NotSupportedException();
}
```

Inbound communications

Only the following method is supported in this mode:

```
public java.util.List<ObjectType> getReceivedTypes()
```

This method must return the list of events that could be received from the EIS.

Outbound communications

Two types of data exchange modes are supported:

- Request/response (such as an HTTP request)
- Query (such as an SQL SELECT query)

Data types from queries are retrieved via:

```
public java.util.List<ObjectType> getRequestTypes()
```

This method must return the list of query types that could be sent to the EIS.

Once a query produces a response, such as when the **getPurchaseOrder(int id)** function returns a "PurchaseOrder" object, the following method must be used to describe the expected response type:

```
public java.util.List<ObjectType> getResponseTypes()
```

Once a query leads to its response, such as when the "getPurchaseOrders(PurchaseOrderType)" function returns a "PurchaseOrder" object, the following method:

```
public java.util.List<ObjectType> getReceivedTypes()
```

is used. Instead of sending a query containing data, the EIS is queried to find elements via their metadata.

Note that the **getResponseTypes()** method is not supported separately. An EIS response cannot be received if a query has not been sent to it.

Navigation

Once the first level types have been retrieved, the following method is called to return the sub-types of the other levels:

```
public ObjectType getType(String namespace, String name)
```

Using the information from the *namespace, name* couple sent as parameter by the caller, it is possible to know the level that is to be described. Once a level is terminal, the method must return *null*.

3 Runtime

This section describes elements that are used by a connector to connect to an EIS and exchange data.

Outbound communications

Configuration

The key class of this communication mode is the one that implements the *javax.resource.spi.ManagedConnectionFactory* interface. This class must also implement the *javax.resource.spi.ResourceAdapter* interface. As outlined by the JCA specifications, this class must be a JavaBean. The fields of this JavaBean object represent information that is required by the connection regardless of the client. For example, for a database accessed via an ODBC connection, the name of this database is required regardless of the client.

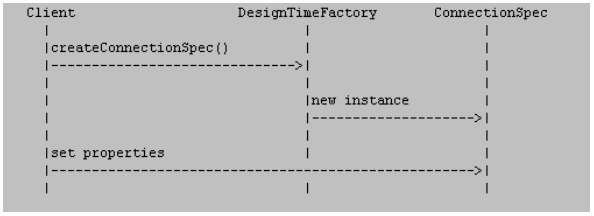
```
package com.mycompany.myeis;

import javax.resource.spi.ManagedConnectionFactory;
import javax.resource.spi.javax.resource.spi.ResourceAdapter

public class MyEISManagedConnectionFactory implements ManagedConnectionFac
tory, ResourceAdapter
{
private String dataSourceName;
```

```
public String getDataSourceName()
{
return dataSourceName;
}

public void setDataSourceName(String dataSourceName)
{
this.dataSourceName = dataSourceName;
}

...
}
```

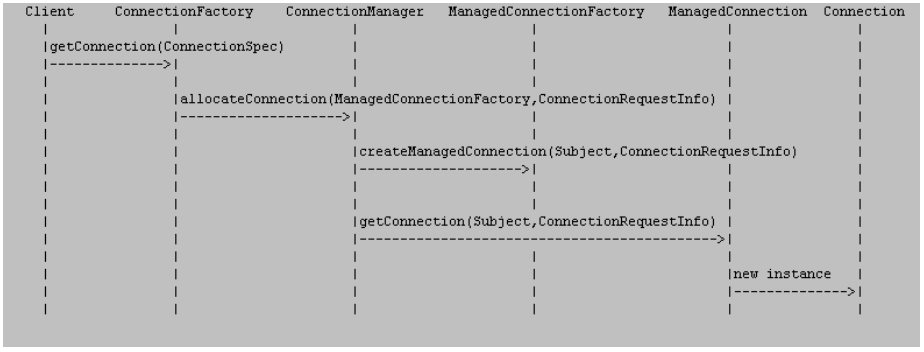
Connection

Client connection's obtained from a connector built using the SDK complies with the JCA standard.

A *javax.resource.cci.ConnectionSpec* object representing the connection information must be retrieved first. This is done as shown in the following schema:



The client accesses EIS via the *javax.resource.cci.ConnectionFactory* interface to create a connection from the information that is supplied. To simplify the example, certain details have been omitted (connection pooling, connection listener).



All implementations must return a *com.hp.ov.cit.connector.cci.Connection* type connection object.

Exchange

Once the connection has been established, the client application (Connect-It) is capable of exchanging data with the external system. At this stage, two exchange modes are possible in accordance with design-time information:

- Request with or without a response
- Query

Request/Response mode

Most exchanges with an EIS can be grouped into this category. For example, inserting a record into a relational database. Accessing this feature is done via the method:

public Interaction createInteraction()

The following *com.hp.ov.cit.connector.cci.Interaction* interface is used:

```
public interface Interaction
{
...
public ObjectRecord execute(ObjectRecord request) throws ResourceException
;
}
}
```

Data is supplied as input and a response or no response is returned.

Query mode

A prototype of expected data is sent to the EIS via a query. By analogy, an SQL SELECT query specifies in the input which columns are expected in the records that are retrieved.

Accessing this feature is done via the method:

public Statement createStatement()

The following *com.hp.ov.cit.connector.cci.Statement* interface is used:

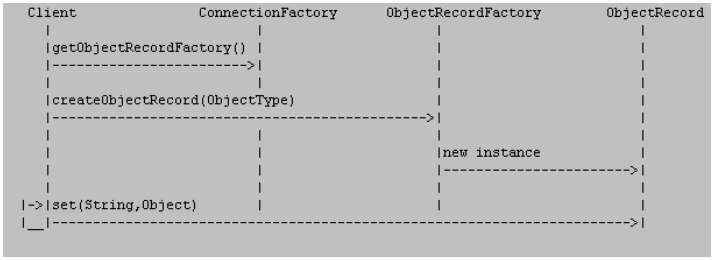
```
public interface Statement
{
...
public ObjectResultSet executeQuery(ObjectRecord prototype) throws Resourc
eException;
}
}
```

The **next()** and **getObjectRecord()** methods are used to iterate through the result set to retrieve the data.

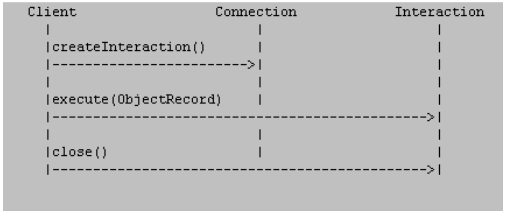
```
public interface ObjectResultSet
{
public boolean next();
public ObjectRecord getObjectRecord();
public void close() throws ResourceException;
}
```

Schemas

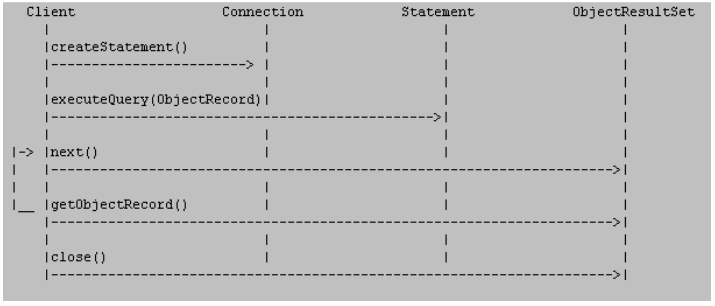
Creating a piece of data from design-time metadata:



Creating an interaction with data that was retrieved:



Querying from a data prototype that was retrieved:



Inbound communications

Configuration

The key class of this communication mode is the one that implements the *javax.resource.spi.ResourceAdapter* interface. As outlined by the JCA specifications, this class must be a JavaBean. The fields of this JavaBean object represent information that is required by the connection regardless of the client.

Connection

The class that implements the *javax.resource.spi.ActivationSpec* interface represents the information required to establish a client connection. As for the *javax.resource.spi.ResourceAdapter* class, it must be a JavaBean object.

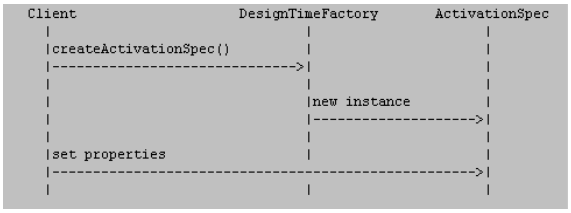
Exchange

The EIS initiates the exchange. The connector acts as an event listener. When events are received, the connector notifies the client via the *javax.resource.spi.endpoint.MessageEndPointFactory* object passed as parameter when it was started. This allows it to create a *ConnectionListener* object whose interface is:

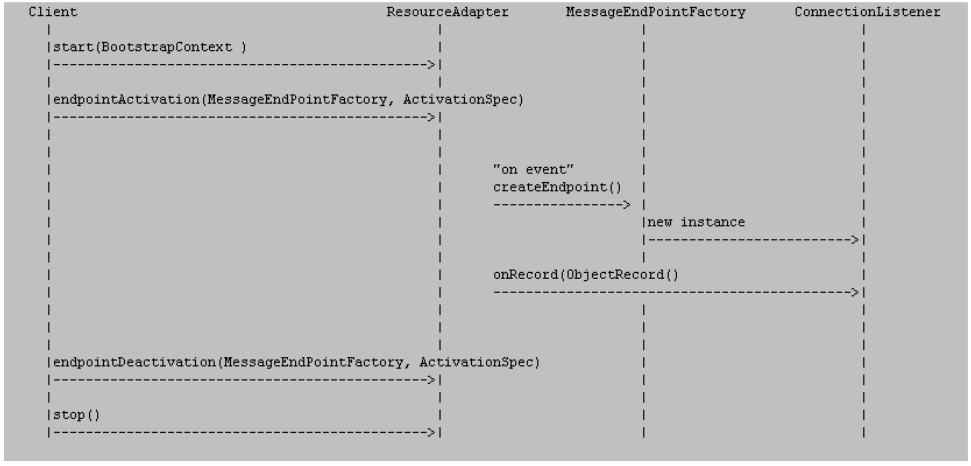
```
public interface ConnectionListener extends MessageListener
{
public void onException(Exception exception);
public ObjectRecord onRecord(ObjectRecord record);
}
```

Schemas

Retrieving connection information (design-time):



Life cycle of the ResourceAdapter class:



4 Deployment

To use the connector with Connect-It you must first create a deployment file. The SDK uses its own deployment descriptor file and not the `ra.xml` descriptor from the JCA standard. This XML file is based on the context notion introduced by the *Spring* framework. It must be named `designtime-beans.xml` and saved to the root of the connector's JAR archive.

The following information is included:

- Complete name of the *com.hp.ov.cit.connector.spi.designtime.DesignTimeFactory* class.
- Complete name of the *javax.resource.spi.ResourceAdapter* class. For outbound communications, the *javax.resource.spi.ManagedConnectionFactory* class is implemented.

  An example is given below:

```
<beans>

<bean id="designTimeFactory" class="com.mycompany.myeis.MyEisDesignTime
Factory">
<property name="resourceAdapter">
<ref bean="resourceAdapter"/>
</property>
</bean>

<bean id="resourceAdapter" class="com.mycompany.myeis.MyEisManagedConne
ctionFactory"/>

</beans>
```

5 Configuration

A certain number of configuration files are required by *Connect-It* in order to use the connector. This name must be unique among all existing *Connect-It* connectors. We recommend that you follow "Java" package naming conventions. In this example we will use the name *com.mycompany.myeis* for our connector.

Description file

This is main file for the actual description of the connector. It groups all the properties related to the connector such as its unique name, the references to file names described after and its activation key. The extension of this file must be `.dsc`. We recommend that you name the file `myeis.dsc`.

Example:

```
{CONNECTORDESC
InternalName=com.mycompany.myeis
ParentInternalName=Application_connectors
Name=My EIS
HTMLHelp=This is a description of my connector
Key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXX
IconFile=myeis.bmp
Sched.CanUsePointer=0
Cnx.HasCnx=1
Wizard.File=myeis-wizard.xml
Java.Class=com.hp.ov.cit.container.RAContainer
Java.Configuration.File=myeis-config.xml
```

```
Java.JVMConfiguration.File=myeis-jvmconf.xml
Java.HasOptions=1
}
```

Icon file

You must supply a 16x16 bitmap to view an icon in the connectors navigation tree. This file can be named `myeis.bmp`.

Configuration file

This file contains the set of JavaBeans properties that must be configured by the user. This file is also used to specify which properties will be included in the scenario configuration that is exported via this command line:

```
conitsvc -export[:<property file>] <scenario>
```

Example of a *myeis-config.xml* file:

```
<configuration>
<property name="ra_url" type="String" export="true">
<definition>
<default/>
</definition>
<export>
<description>URL</description>
</export>
</property>
<property name="cs_userName" type="String" export="true">
<definition>
<default/>
</definition>
<export>
<description>User</description>
</export>
</property>
</configuration>
```

Wizard file

An XML-format wizard definition file for the connector.

It is used to describe the pages that are used to configure the connector in *Connect-It*. It contains a connection definition page. Interface controls are also described in terms of notions (text, checkbox, button), labels, position, etc.

All JavaBeans properties that must be configured by the user must be in this file. The following naming convention must be used:

- The prefix *ra_* must be added to each property that is related to the implementation of the *javax.resource.spi.ResourceAdapter* interface.
- The prefix *mdcs_* must be added to each property that is related to the implementation of the designtime (metadata) *javax.resource.cci.ConnectionSpec* interface.
- The prefix *cs_* must be added to each property that is related to the implementation of the *javax.resource.cci.ConnectionSpec* interface.
- The prefix *as_* must be added to each property that is related to the implementation of the *javax.resource.spi.ActivationSpec* interface.

Example of a `myeis-wizard.xml` file:

```
<wizard>

<page name="pgConnector">
<title>Connection</title>

<description>Configure connection to MyEIS</description>

<description>Enter the URL</description>
<control type="Textbox" name="ra_url">
<Value>$(GetValue[ra_url])</Value>
<label>URL</label>
<XOffset>2500</XOffset>
<labelLeft>1</labelLeft>
<Mandatory>1</Mandatory>
<MandatoryMsg>You must specify an URL value</MandatoryMsg>
<bind>Value</bind>
</control>

<description>Enter the user name</description>
<control type="Textbox" name="cs_userName">
<Value>$(GetValue[cs_userName])</Value>
<label>User</label>
<XOffset>2500</XOffset>
<labelLeft>1</labelLeft>
<bind>Value</bind>
</control>

<Transition>
<To script="true">{trConnector}</To>
</Transition>
</page>

</wizard>
```

JVM configuration file

You must provide the application with the classpath configuration file in order to start the JVM.

Connect-It requires a minimum configuration regardless of the connector built using the SDK. This configuration is described in the file located at *CONNECT-IT_HOME*/*config*/*shared*/*jca-container-jvmconf.xml*. It must be included in your own JVM configuration file.

Example of a `myeis-jvmconf.xml` file:

```
<jvmConfiguration id="com.mycompany.myeis">
<jarLocation>./com.mycompany.myeis</jarLocation>
<jars>
<jar groupId="com.mycompany.myeis" optional="false" provided="true"
version="1.00" versionNeeded="true">myeis</jar>
</jars>
<import>../shared/jca-container-jvmconf.xml</import>
</jvmConfiguration>
```

6 Packaging

The connector must be packaged with the Connect-It installation in the following manner:

```
Connect-It/
|
|- lib/
|   |- com.mycompany.myeis/
|       |- myeis-1.00.jar
|       |- myeis-3rdparty1.jar
|       |- myeis-3rdparty2.jar
|       |- ...
|
|- config/
|- com.mycompany.myeis/
|- myeis.bmp
|- myeis-jvmconf.xml
|- myeis.dsc
|- myeis-wizard.xml
|- myeis-config.xml
```

Note:

To ensure that names are unique, the connector's configuration and archive directories must follow the "Java" package naming conventions. The name *com.mycompany.myeis* in the example above follows these conventions.

Java archive

The following structure must be used for the `myeis-1.00.jar` archive:

```
myeis-1.00.jar
|
|- designtime-beans.xml
|
|- com/
|  |- mycompany/
|     |- myeis/
|        |- MyEisDesignTimeFactory.class
|        |- MyEisManagedConnectionFactory.class
|        |- MyEisConnectionManager.class
|        |- ...
|
|- META-INF/
|  |- Manifest.mf
```

7 Extension

Interface com.hp.ov.cit.connector.spi.ContainerContext

When the connector is instantiated via Connect-It, the application provides the implementation with a specialization of the <javax.resource.spi.BootstrapContext> class used to access specific features of the container. This context class provides the following possibilities:

Event listener

It is possible to receive notifications of execution events concerning the Connect-It scenario. This is done via a listening class using these methods:

```
public void addContainerListener(ContainerListener listener);
public void removeContainerListener(ContainerListener listener);
```

The SDK has introduced 2 listening class types:

- com.hp.ov.cit.connector.spi.ExecutionListener: Listens for notifications when a scenario starts or stops.
- com.hp.ov.cit.connector.spi.SessionListener: Listens for session opening and closing notifications for a scenario that is executing.

Access to the scenario path

It is possible to obtain the full path of the scenario executed via the call:

```
public String getScenarioAbsolutePath();
```

If you are interested by these features, you will need to enter the following code in your <javax.resource.spi.ResourceAdapter> implementation:

```
public void start(BootstrapContext bootstrapContext) throws ResourceAdapte
rInternalException
{
if (bootstrapContext instanceof ContainerContext)
{
//store this CIT context for use
}
else
{
//who is my container?
throw new ResourceAdapterInternalException();
}
}
```

com.hp.ov.cit.connector.spi.designtime.ObjectTypeProviderEx class

This class is a specific implementation of the <com.hp.ov.cit.connector.spi.designtime.ObjectTypeProvider> interface. It enables any implementation that uses it to supply additional information concerning the supported types to the container.

The <com.hp.ov.cit.connector.cci.ObjectType> interface provides the Java class type to contain simple data (whole, Boolean values, etc). However, for some types, notably dates, a Java class may be insufficient to describe the semantics of a type (for example, date, date/time or time). This special class addresses this issue by giving the container additional information about a simple type that is taken into consideration via the method:

```
public String getXSDBuiltinDatatype(ObjectType simpleType)
```

This method returns the name of a "built-in" type from the XML Schema [http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#built-in-datatypes] specification in order to complete the description of a simple type.

By default the basic class does not provide any additional information about the manipulated simple types.

Typical usage is as follows:

```
public class MyObjectTypeProvider extends ObjectTypeProviderEx
{
....

@Override
public String getXSDBuiltinDatatype(ObjectType simpleType)
{
```

```
if( simpleType instanceof MyDateObjectType)
{
return "date";
}
else if( simpleType instanceof MyDatetimeObjectType)
{
return "dateTime";
}
else if( simpleType instanceof MyTimeObjectType)
{
return "time";
}
else
{
//sorry ...no additionnal info along the Java class type
return null;
}
}
}
```

8 Use

Implementing a connector developed using the SDK, is linked to:

- An SDK access declaration in the Connect-It authorization certificate.
- A key that has been generated for the connector created using the SDK.

Authorization certificate

The authorization certificate activates:

- The runtime that enables the connector created using the SDK to be used.
- The menu used to generate a key for the newly created connector (key used by the runtime).

▶ Connect-It - User's guide, Installation chapter, Entering the authorization certificate.

Generate a key

A key allows the connector to be used.

To generate a key:

1 Launch the Connect-It scenario builder

2 Select **Java/ Generate SDK activation key**

3 In the window that is displayed, enter:
   - The name of the connector.
   - Its mode (production, consumption).
4 The key that is generated must be copied to the description file
   ▶ Connect-It Guide - SDK, section Database description file [page 65].

This key is linked to the authorization certification which enables the connector to be activated and used.

I Appendix

9 Wizard file

This section provides information about the syntax used for the connector's configuration wizard's XML file.

General structure

A wizard is made up of pages. Each pages can have input fields, labels and descriptions. Each page defines a transition to the next page.

```
<wizard>

<include/>
<property/>
<page>
<transition/>
</page>

</wizard>
```

Wizard element

The root element must be *wizard*.

Possible sub-elements are:

Element	Optional	Description
include	Yes	Used to include definitions from external files.
property	Yes	Used to define scripted properties.
page	Yes	Defines pages that make up the wizard.

Include element

Used to include a file. The syntax is as follows:

```
<include type="..." [basedir="..."]>the file name</include>
```

Attribute	Optional	Description
type	No	Defines the inclusion type
basedir	Yes	Defines the directory of the file to include
included	Yes	Used to ignore or not to ignore the element

The inclusion types are:

- string
- wizard

String inclusion type

Used to include a resources file (localization strings). By default, the path of the file to include is relative to the current file.

Strings that are defined in this file are accessed using the following syntax: $(IDS_*NAME_OF_THE STRING*).

For example:

Let's examine the `myeisstrings.str` file

```
EIS_TITLE, "Title for the EIS"
EIS_DESCRIPTION, "Description of the EIS"
....
```

The resources are used in the wizard file by including the string IDs:

```
<wizard>
<include type="string">eisstrings.str</include>
<title>$(IDS_EIS_TITLE)</title>
...
</wizard>
```

Note:

- Access to the resources is only effective for elements defined after the inclusion.
- The inclusion is taken into account when the wizard is generated. Its value cannot be scripted.

Wizard inclusion type

Used to include another wizard file. The elements that can specify this type of inclusion are *wizard* and *page*.

Parameters can be sent to the included wizard and can be accessed using this syntax:

```
$(GetValue[NAME_OF_THE PARAMETER])
```

For example, to send the parameter *myParameter* whose value is *myValue* to the `myIncludedWizard.xml` wizard, the following syntax is required:

```
<include type="wizard" myParameter="myValue>myIncludedWizard.xml</include>
```

Page element

A wizard is made up of pages. Possible attributes are:

Attribute	Optional	Description
name	No	Defines the name of the page. Each page name is unique.
included	Yes	Used to ignore or not to ignore the element.

Possible sub-elements are:

Element	Optional	Description
Transition	No	Defines the transition to the next page.
Description	Yes	Use to add a description to the page, a page section or a control.
Title	Yes	Defines the title of the page.
property	Yes	Used to define scripted properties.

Element	Optional	Description
control	Yes	Defines the controls on the page.
linebreak	Yes	Defines line breaks.
separator	Yes	Defines a horizontal separator.

```
<page name="..." included="...">


<title/>
<image/>
<description/>
<property/>
<control/>
<linebreak/>

<separator/>
<transition/>

</page>
```

Note:

The first page of a connector's wizard must be named **pgConnector**.

Property element

A property is a basic value type such as *string* or *long*. Possible attributes are:

Attribute	Optional	Description
name	No	Defines the name of the property.
included	Yes	Used to ignore or not to ignore the element.
script	Yes	Used to specify scripted content.

Example:

```
<page name="myPage">
<property name="IsVisible" type="Long" script="true">RetVal = 1</property>
</page>

<property name="DelimString" script="true">RetVal = ""</property>
```

A property is used via the *property full path* syntax which references the complete path (without the root) of the property in the XML tree structure.

Example:

```
<visible script="true">{myPage.IsVisible} &lt;&gt; 1</visible>
<value script="true">{DelimString}</value>
```

Control element

Used to define a graphical control. Possible attributes are:

Attribute	Optional	Description
name	No	Defines the name of the property.
type	No	Defines the control type. It must be unique for the page.
included	Yes	Used to ignore or not to ignore the element.
script	Yes	Used to specify scripted content.

Possible sub-elements, regardless of the control type, are:

Element	Optional	Type	Description
visible	Yes	boolean	Specifies whether or not the control is visible.
enabled	Yes	boolean	Specifies whether or not the control is grayed out.
readonly	Yes	boolean	Specifies whether or not the control can be edited.
mandatory	Yes	boolean	Specifies whether or not the control requires a value.
mandatorymsg	Yes	string	Specifies the error message if no value is provided when the mandatory attribute is present and is equal to 1.
label	Yes	string	Defines text above the control.
labelleft	Yes	boolean	If '1' or 'true', positions the label to the left.

Element	Optional	Type	Description
xoffset	Yes	long	Defines the space to the left of the control.
bind	Yes	string	Specifies the control elements whose values were taken into account when its associated page was validated. Example: <bind>value</bind> used to take into account the value of the element <value>.
property	Yes	string	Used to define scripted properties.

Other sub-elements are available depending on the type of control that is involved. The main controls and their sub-elements are:

Control type	Sub-element	Type	Description
textbox	value	string	Value of the input text.
	multiline	long	0 = single line otherwise percentage of the control size
	password	boolean	Value specifying whether or not the field is encrypted. 1 = encrypted field
checkbox	value	boolean	Specifies whether or not the control is checked.
	caption	string	Control label
combobox	value	string	Value of the selected item.
	values	string	List of possible items (label=value) separated by commas. Example: <values>English=en,French=fr<values>
numbox	value	long	Numerical value for the control.
	minvalue	long	Specifies the minimum value.

Control type	Sub-element	Type	Description
	maxvalue	long	Specifies the maximum value.
label	caption	string	Control label
fileedit	value	string	Path of the selected file.
	openmode	long	Defines the editing type: ■ 1 = OPEN ■ 2 = SAVE ■ 4 = OPEN_DIR ■ 8 = SAVE_DIR ■ 16 = APPEND
	filters	string	Defines a file filter. Example: <filters>XML files (*.xml)|*.xml|XMLSchema files (*.xsd)|*.xsd|</filters>
	defext	string	Default extension to use. Example: <defext>txt</defext>
	serializationId	string	Defines the id of the file selection control. Several controls can use the same id. This id is used to save the path of the last selected file.
optionbuttons	value	sting	Value of the selected item.
	values	string	List of possible items (label=value) separated by commas. Example: <values>ISO-8859-1=0,UTF-8=1,Shift-JIS=2<values>
	border	boolean	Specifies whether or not the control has a frame.

Example:

```
<control type="TextBox" name="Server">
<value>$(GetValue[Server])</value>
```

```
<caption>$(IDS_SERVER_LABEL)</caption>
<xoffset>2500</xoffset>
<bind>value</bind>
</control>
```

Bind attribute

The *bind* attribute is used to link a control to a configuration property of a connector. Currently, only the value *value* is supported by the SDK. When it is specified for a control named 'cs_myprop', the value of the control's <value> element is sent to the connector as the value for the 'cs_myprop' configuration property (as the value of the 'myprop' property of the connector's **ConnectionSpec** property).

Password management

Managing configuration properties such as passwords requires specific handling in the wizards. If the property containing the password is 'cs_password', the name of the wizard control for this property must be 'clearcs_password'.

Example:

```
<control type="TextBox" name="clearcs_password">
<value>$(GetValue[cs_password])</value>
<password>1</password>
<label>$(IDS_PASSWORD_LABEL)</label>
<xoffset>2500</xoffset>
<labelleft>1</labelleft>
<bind>value</bind>
</control>
```

Linebreak and separator elements

These elements are used to format the wizard page. Possible attributes are:

Attribute	Optional	Description
included	Yes	Used to ignore or not to ignore the element.

Transition element

Every page must have a transition element. This element specifies what the next page is. Possible attributes are:

Attribute	Optional	Description
script	Yes	Used to specify scripted content.

Examples:

```
<transition><to>nextPage</to></transition>

<transition>
<to script="true">
if( $(GetValue[ShowAdvancedWiz]) = 1 ) then
RetVal = "pgAdvanced"
else
RetVal = {trConnector}
end if
</to>
</transition>
```

Note:

The transition of the last page of a connector's wizard must be equal to the scripted value *{trConnector}*.

Script attribute

Wizards support simple scripts written using Basic syntax. These scripts are evaluated when the wizard is executed.

The *script* attribute is available for all elements containing a value. It is used to specify the value of the element as a scripted expression which is evaluated when the value of the attribute is *true*.

Example:

```
<... script="true">
if {Protocol.Value} = "ftp" or {Protocol.Value} = "http" then
```

```
RetVal = 1
else
RetVal = 0
end if
</...>
```

In Basic scripts used in the wizards, the syntax {...} references the value of a wizard control or property. These values are referenced using the complete path (without the root) of the property in the XML tree structure.

Included attribute

This attribute is available for most elements. It is optional. It contains a boolean value which specifies if the element in question is to be ignored or not.

The different values that this attribute can have are:

- *0* or *1* (or any other that is not *0*)
- *false* or *true*
- An expression that uses the *and*, *or* and *not* operators.

When the value of this attribute is *false*, the contents of the element to which it belongs will be ignored.

 Note:

The value of this attribute is evaluated when the wizard is generated and not when it is executed. Therefore, including an element cannot depend on the value of a control or any other scripted expression. The value of this attribute is generally evaluated using the **GetValue** function.

Functions

The functions defined below are not Basic script functions. They are functions that are evaluated when the wizard is generated and not when it is executed.

Format of the functions:

```
$(FunctionName[param1,param2<,optionalparam>,...])
```

GetValue function

This function is used to dynamically retrieve a value from the wizard. This function is the most used wizard function since it allows the current value of a connector's configuration property to be retrieved.

The syntax is as follows:

```
$(GetValue[name,default])
```

The *name* parameter specifies the name of the value to find. The *default* parameter defines a default value if the current value is not found.

Several existing values have predefined names:

- OSUnix: Returns *1* if the platform is Unix and *0* otherwise.
- OSWindows: Returns *1* if the platform is Windows and *0* otherwise
- WizardDir: Returns the complete path of the installation wizard directory (CONNECT-IT_HOME/config/wiz)
- NameID: Returns the name of the connector
- ShowAdvancedWiz: Returns *1* if the wizard is in advanced mode and *0* otherwise
- ConfigDir: Returns the complete path of the connector's configuration directory

When the *GetValue* function is called, the search for the value is done on:

1. Specific values defined in the description file.
2. The connector's configuration properties.
3. Predefined values.

Example:

```
<value>$(GetValue[mylogin])</value>

<property name="trConnector" script="true">
if( $(GetValue[Cnx.HasCnx, 1]) = 1 then
RetVal = "pgConnection"
else
...
</property>

<control type="checkbox" name="UseWindowsRegistry" included="$(GetValue[OS
Windows])">
<value>$(GetValue[UseWindowsRegistry])</value>
<caption>$(IDS_SERVER_LABEL)</caption>
<xoffset>2500</xoffset>
<bind>value</bind>
</control>
```

Dump function

This function is used to format a string for use in a script. The string is enclosed by quotation marks and quotation marks in the string are escaped. This function is very useful in scripts that retrieve strings using the **GetValue** function or with strings from an `.str` file. The syntax is as follows:

```
$(Dump[string])
```

Example:

```
<value script="true">RetVal = $(Dump[$(GetValue[theValue])])</value>
```

EspaceCommas function

This function is used to escape commas in a string. The function can be used when the string is a sub-element of a string that uses a comma as character separator (for example, the values element of the *optionbuttons* control). The syntax is as follows:

```
$(EscapeCommas[string])
```

File function

This function is used to retrieve the full path of a file. The syntax is as follows:

```
$(File[name,basedir])
```

The *name* parameter specifies the file's name. The *basedir* parameter defines the file's directory. The default directory is the wizard's directory.

Example:

```
<image>$(File[myfile.bmp])</image>
```

10 Configuration file

This section provides information about the syntax used for the configuration file.

The file is structured in the following manner:

```
configuration>

<property>
<definition>
<default/>
</definition>
<export>
<description/>
</export>
</class>
</property>

<property>
<definition>
<default/>
</definition>
<export>
<description/>
</export>
</class>
</property>

</configuration>
```

Configuration element

The root element must be *configuration*. Possible sub-elements are:

Element	Optional	Description
property	Yes	Defines one or more properties required by the connector's Java code.

Property element

Specifies a Java configuration property.

Possible attributes are:

Attribute	Optional	Type	Description
name	No	string	Defines the name of the property.
type	No	string	Specified the property type.
export	Yes	Boolean	Specifies whether or not the property needs to be taken into account during export (-export option).

Possible sub-elements are:

Element	Optional	Description
Definition	Yes	Property definition.
export	Yes	Definition of the export.
class	Yes	Definition of the corresponding Java class.

Definition element

Has the following sub-elements:

Element	Optional	Type	Description
default	Yes	string	Specifies the default value that is used when initializing the wizard.

Export element

Has the following sub-elements:

Element	Optional	Type	Description
Description	Yes	string	Specifies the description used when the property is exported. Appears as a comment in the exported properties file.

Class element

A Java class is implicitly associated with each property type. This element lets you overload the implicit class of the property type.

In the example below, a *String* property type is declared and corresponds to a JavaBean property in the *java.net.URI* class.

```
<property name="myURIProperty" type="String" export="true">
<class>java.net.URI</class>
</property>
```

Property types

The following table lists the supported property types and their default JavaBean property type.

Type	JavaBean Type
Boolean	java.lang.Boolean
Byte	java.lang.Byte
Short	java.lang.Short
Long	java.lang.Integer

Type	JavaBean Type
LingInt	java.lang.Long
Float	java.lang.Float
Double	java.lang.Double
String	java.lang.String
Memo	java.lang.String
Date	java.util.Date
Time	java.sql.Time
Timestamp	java.sql.Timestamp
Password	java.lang.String
File	java.io.File
Url	java.net.URL

Please consult the JavaBeans documentation for the complete list of supported JavaBean types.

11 JVM configuration file

This section provides information about the syntax used for the JVM configuration file.

The file is structured in the following manner:

```
<jvmConfiguration>

<jarLocation/>
<jarLocation/>

<jars>
<jar/>
<jar/>
<jar/>
</jars>

<jvmOptions>
<jvmOption/>
<jvmOption/>
</jvmOptions>

<import/>
<import/>

</jvmConfiguration>
```

jvmConfiguration element

The root element must be *jvmConfiguration*.

Possible attributes are:

Attribute	Optional	Type	Description
id	No	string	Defines a unique identifier for the configuration.

Possible sub-elements are:

Element	Optional	Type	Description
jarLocation	Yes	string	Defines the paths of the classpath used by the connector.
jars	Yes		Defines the archives used by the connector.
import	Yes	string	Used to include a classpath from an external file.
jvmOptions	Yes		Used to define JVM options.

jarLocation element

The connector's classpath comprises one or more paths which reference the different archives (.jar or .zip files) required for code execution. For each connector it is possible to define the paths to search for the archives. The path value is either relative to the Connect-It installation lib directory or an absolute path. The archives are searched in the order that the paths are declared.

Example:

```
<jarLocation>./com.mycompany.myeis</jarLocation>
<jarLocation>c:/myEIS/myEISPath</jarLocation>
```

By default, if no *jarLocation* element is specified, the path used is the Connect-It installation **lib** directory.

Jars element

Possible sub-elements are:

Element	Optional	Type	Description
jar	Yes	string	Defines an archive entry for the classpath.

Jar element

Possible attributes are:

Attribute	Optional	Type	Default	Description
groupId	No	string		Defines a group identifier for the archive.
provided	Yes	boolean	true	Specifies if the archive in question is from one of the classpath search paths or if the path needs to be provided by the user. The value 'true' indicates that it is supplied by the installation (search paths). In this case, the 'optional' attribute is ignored.

Attribute	Optional	Type	Default	Description
optional	Yes	boolean	false	Specifies if the archive is optional. The value 'false' indicates that the archive must be present in one of the classpath search paths, or in the additional classpath defined in the application ('Java/ Configure the JVM' menu) or in the connector (on the wizard's 'Configure the JVM' page).
version	Yes	string		Used to append archive version to the archive. The full name of the archive becomes `name-version.jar`. If this extension is not found, a new search is done using the fullname `name-version.zip`.
versionNeeded	Yes	boolean	true	Indicates if archive must be searched using its name and version. ■ The value 'true' indicates that the search is on the name and the version. ■ The value 'false' indicates that the search is on the name and the version, then just the name for each path.

The value must reference the name of the archive to be added.

Sample classpath entry for the `xercesImpl-2.6.2.jar` library that is provided with the application:

```
<jar groupId="xerces" optional="false" provided="true" version="2.6.2" ver
sionNeeded="true">xercesImpl</jar>
```

jvmOptions element

This element is used to define additional JVM options

Possible sub-elements are:

Element	Optional	Type	Description
jvmOption	Yes	string	Defines a JVM option.

Example:

```
<jvmOptions>
<jvmOption>-Xmx125m</jvmOption>
<jvmOption>-Dcom.sun.management.jmxremote</jvmOption>
</jvmOptions>
```

Import element

In addition to the connector's configuration, it is possible to provide additional JVM configuration elements. Theses elements are declared in one or more files which use the same syntax. Depending on where the import declaration is made, the declarations can come before or after the current definitions. The value must reference the relative path of the file to import.

Example:

```
<import>../shared/jca-container-javaconf.xml</import>
```

12 Database description file

This section provides information about the syntax used for the description file.

File structure

The file is structured in the following manner:

```
{CONNECTORDESC
//property list
//property name=property value
Name=
InternalName=
...
}
```

Properties

The following table list the connectors' properties:

Property	Type	Optional	Default value	Description
Common properties				

Property	Type	Optional	Default value	Description
Name	string	No		Connector name that is displayed.
InternalName	string	No		Internal name of the connector (unique).
ParentInternal-Name	string	Yes		Name of the parent node that it belongs to.
HTMLHelp string yes Description in html format.				
Key	string	No		Activation key
Icon				
IconFile	string	No		Relative path to the connector's icon (.bmp)
Schedulers				
Sched.CanUsePointer	boolean	Yes	true	The SDK does not provide support for schedule pointers. This value must be set to 0. *Sched.CanUsePointer=0*
Cache				
Cache.Support-Cache	boolean	Yes	true	The SDK does not provide support for the metadata cache. This value must be set to 0. *Cache.Support-Cache=0*
Timezone				
Tmz.HandleServer-Delay	boolean	Yes	true	The SDK does not provide support for server time differences. This value must be set to 0. *Tmz.HandleServer-Delay=0*
External formats				

Property	Type	Optional	Default value	Description
ExtFmt.Use	boolean	Yes	true	The SDK does not provide support for extended formats. This value must be set to 0. *ExtFmt.Use=0*
Wizard				
Wizard.File	string	Yes		Relative path of the Wizard file.
Java				
Java.Class	string	No		Specifies the connector's Java class. Must be: `Java.Class=com.hp.ov.cit.container.RAContainer`
Java.Configuration.File	string	Yes		Relative path of the configuration file.
Java.JVMConfiguration.File	string	Yes		Relative path of the JVM configuration file.
Java.HasOptions	string	Yes	false	This value should be set to 1. Java.HasOptions=1
Java.SupportProxy	Boolean	Yes	false	Is a proxy server configuration supported?
Java.PriorToJdk15ProxyRegistration	Boolean	Yes	false	Does the record of the proxy server use the function introduced by the <java.net.ProxySelector> class?
Miscellaneous				

Property	Type	Optional	Default value	Description
RedeployOn-Change	string	Yes		If the properties in this list are modified, the connector must be redeployed. Example: `RedeployOnChange=cs_CacertsFile cs_KeystoreFile as_KeystorePassword`

Example

Below is a sample configuration file for an *outbound* type connector built using the SDK.

```
{CONNECTORDESC
Name=MyEIS
InternalName=com.mycompany.myeis
ParentInternalName=com.mycompany
HTMLHelp=Connector to interact with my eis
Key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXX
IconFile=myeis.bmp

EventDriven=0
SupportParallelization=1
Transac.CanSupportTransactions=1
Cnx.HasCnx=1
Cnx.CanDisableReconnection=0
Wizard.File=myeis-wizard.xml
Java.Class=com.hp.ov.cit.container.RAContainer
Java.Configuration.File=myeis-config.xml
Java.JVMConfiguration.File=myeis-jvmconf.xml
Java.HasOptions=1

// The following properties must always have these values
Sched.CanUsePointer=0
Cache.SupportCache=0
Tmz.HandleServerDelay=0
ExtFmt.Use=0
}
```

Additional information

Multiple descriptions

A description file can contain several descriptions each of which corresponds to a *CONNECTORDESC* section. Although it is recommended to write a single description file for each connector, including several descriptions in the same file can be useful when defining connector categories or when managing different versions of the same EIS.

Connector hierarchy

The *ParentInternalName* property is used to specify the internal name of the parent node, or category, in the connector hierarchy. Categories are also defined in description files in a more simplified format:

```
{CONNECTORDESC
InternalName=...
ParentInternalName=...
Name=...
HTMLHelp=...
IconFile=...
}
```

If no *ParentInternalName* property is specified, the category (or the connector) will be located at the root of the hierarchy.

Connect-It has a certain number of predefined categories:

Category	Internal name
Application connectors	Application_connectors
Protocol connectors	Protocol_connectors
ERP connectors	ERP_connectors
Inventory connectors	Gateways

13 Java code

JavaBeans

Supported types

A certain number of interfaces from the JCA specifications must be implemented as JavaBeans. The following interfaces are used by the SDK:

```
javax.resource.spi.ManagedConnectionFactory
javax.resource.spi.ResourceAdapter
javax.resource.cci.ConnectionSpec
javax.resource.spi.ActivationSpec
```

The following table lists the value types that are authorized for their properties:

java.lang.Boolean
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Byte
java.lang.Short
java.lang.Long
java.lang.Float
java.lang.Character

The SDK extends this list to other frequently used types. The following types are supported:

java.util.Date
java.sql.Time
java.sql.Timestamp
java.io.File
java.net.URL
java.net.URI

Validation

In some cases, the value that a JavaBean object property can have depends on another property. Since the object does not control the order in which the properties are updated, the SDK provides an alternative to this problem via the interface.

```
public interface ValidatingBean
{
public void validate() throws InvalidPropertyException;
}
```

This interface is used to manage a validation or initialization phase on the JavaBean that implements it once all of its properties have been updated.

Logging

The SDK uses the *Jakarta Commons Logging (JCL)* framework to log messages in the Connect-It log. Include the following code to use this function from a Java class:

```
package com.mycompany.myeis;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyEISClass
{
private static final Log log = LogFactory.getLog(MyEISClass.class);

...
}
```

JCL defines a priority level for each message. The following levels are used by Connect-It:

- error - Error messages
- info - Information
- warn - Warning messages
- debug - Debug messages Logged only when 'debug' mode is activated.

To log messages to the Connect-It log, use these *org.apache.commons.logging.Log* interface methods:

```
log.error(Object message);
log.error(Object message, Throwable t);
log.warn(Object message);
log.warn(Object message, Throwable t);
log.info(Object message);
log.info(Object message, Throwable t);
log.debug(Object message);
log.debug(Object message, Throwable t);
```

Log4J support

The JCL framework is used to unify access to an implemented logging system: Log4J, JDK Logging, etc.

By default, Connect-It uses a configuration of the Log4J library. All messages logged by the Log4J layer, whether called directly or via the JCL API, will be taken into account by Connect-It.

JDK logging support

Connect-It adds support for the logging framework supplied by the JDK thanks to a static configuration. The default static configuration is described by the `<JRE_HOME>\lib\logging.properties` file.

When the connector is instantiated, the logging level of the JDK's root logger is modified to make it correspond to the one configured for the Connect-It application. All log events, obtained via a direct call to the JDK logging framework, are redirected to the application.

Internationalization

The SDK uses Java's standard internationalization mechanism. To implement this mechanism with your code, you will need to create one or more *properties* files that will contain the strings required for internationalization.

Example

`com/mycompany/myeis/i18n/mymessages.properties` file

```
connection.error = Connection error.
execution.failed = Execution failed.
```

`com/mycompany/myeis/i18n/mymessages_fr.properties` file

```
connection.error = Erreur de connexion.
execution.failed = Echec de l'exécution.
```

com/mycompany/myeis/MyEISClass.java file

```
package com.mycompany.myeis;

import java.util.ResourceBundle;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyEISClass
{
private static final ResourceBundle bundle = ResourceBundle.getBundle("com
.mycompany.myeis.i18n.mymessages");
private static final Log log = LogFactory.getLog(MyEISClass.class);

public void execute()
{
try
{
...
}
catch(Exception e)
{
log.error(bundle.getString("execution.failed"), e);
}
}
}
```