IBM WebSphere Studio Application Developer
Integration Edition V4.1

The J2EE Connector Architecture (JCA) Tool Plug-in

Note!
Before using this information and the product it supports, be sure to read the general information under **Notices**.

First Edition (February 2002)

IBM welcomes your comments. You can send your comments by any one of the following methods:

1. Electronically to either the following e-mail address. Be sure to include your entire network address if you wish a reply.

   torrcf@ca.ibm.com

2. By mail to the following address:

   IBM Canada Ltd. Laboratory
   B3/KB7/8200/MKM
   8200 Warden Avenue
   Markham, Ontario, Canada L6G 1C7

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Chapter 1. Introduction

Written by Michael Beisiegel, Piotr Przybylski and Hesham Fahmy - IBM Toronto Lab

This document describes an extension to the J2EE Connector Architecture (JCA) that allows EISs to provide plug-in components for tool environments.

JCA version 1.0 provides a connector run-time architecture that receives a lot of support by the industry. It makes the J2EE platform a very attractive environment for doing business and enterprise application integration.

One of the most important extensions for the next version of the connector architecture is to make EISs pluggable into tool environments.

This document defines the form of metadata that EISs need to provide, explains how a tool environment interacts with an EIS to get this information, and shows how the EIS provides code generation contributions.

It also covers CCI extensions that take the metadata support into account for doing EIS service invocations.

The goal of these extensions is to be independent of a particular vendor IDE. They do not provide a user interface, nor do they mandate a certain tool task flow. They are provided as plug-ins, so that a particular vendor IDE can render them in the most natural way.

Chapter 2. Metadata Support

Metadata support is important from two perspectives. First, tools want to be able to discover meta information about the functions offered by an EIS. The tools then aid a developer in building components (for example, Java(TM) bean, EJB, flows, and others) that use these functions. Run times either drive a connector through code generated by the tools from the metadata, or they are engines that drive a connector by interpreting the metadata.

Most developers today would agree that XML should be used for capturing the metadata about the functions offered by an EIS. The specific *ML dialect should also be selected with the right view of the future, that is, provide EISs with a growth path into the world of Web services.

The following sections describe how to use the Web Services Description Language (WSDL) with extensions as the perfect solution for EIS metadata support.

WSDL

Web Services Description Language (WSDL) is a W3C note in the moment (http://www.w3.org/TR/2001/NOTE-wsdl-20010315). WSDL has received a lot of momentum since its publication, and is already supported by various tool vendors.

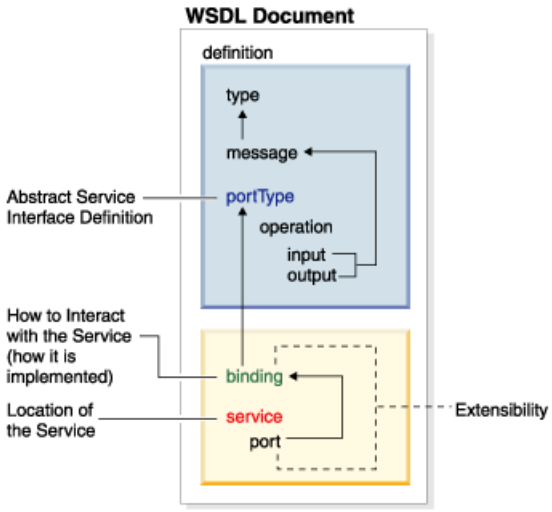
WSDL Document Architecture

The following picture shows the WSDL document architecture. Shown at the top are the sections that allow you to describe a service interface in an abstract way. WSDL prefers to use XML Schema as its canonical type system.

The sections at the bottom allow you describe how and where to access the concrete service that implements the abstract interface.

Looking at the W in WSDL may let you think that the language is for describing Web Services only, but this is not true. The inventors equipped the language with a smart extensibility mechanism, which allows you to describe any kind of service, be it a Web Service or be it some legacy EIS service (function).

The important thing to recognize is that describing the abstract interface in XML, especially the types in XML Schema, doe not mean that you actually use XML in your interaction with a service; the bindings are what make the difference.



WSDL and J2EE Connector Architecture - A Natural Fit

WSDL provides a standard way for describing which services are offered by a specific EIS instance, and how you access them. The J2EE Connector Architecture provides a standard client programming model for accessing EIS services.

If you look at the WSDL information that is relevant for the single execution of an operation you end up with a very natural fit between the two:

```
port                              Connection
operation                         Interaction
        operation binding         InteractionSpec
        input msg                 Record
                input msg binding         Streamable.write
        output msg                Record
                output msg binding        Streamable.read
```

A Java API for WSDL Documents - JSR 110

Runtimes and Tools need an easy way of dealing with WSDL documents. This is what JSR 110 (http://www.jcp.org/jsr/detail/110.jsp) is about. It provides a standard set of APIs for representing and manipulating services described by WSDL documents. These APIs define a way to construct and manipulate models of service descriptions (http://oss.software.ibm.com/developerworks/projects/wsdl4j).

Connector Binding

To invoke an operation via a connector we have to be able to capture meta information about the following aspects:

- The connection properties to be set on a ManagedConnectionFactory
- The interaction properties to be set on an InteractionSpec
- The Record, i.e. their structure and the specific way that they have to be formatted
- The operation, i.e. which InteractionSpec, input Record, and output Record belong together
- That the operation is offered by a specific endpoint

ManagedConnectionFactory is about location of the operation, i.e. which endpoint provides it. The requirement for the Connector Binding is to provide a port extension to capture this information.

The InteractionSpec is about specifying the operation in the way, as it is understood by the endpoint. The requirement for the Connector Binding is to provide an operation binding extension to capture this information.

For Records you need to know their structure and the way that they have to be formatted so that an endpoint is able to interpret them. The structure is taken care of by the XML Schema from which you can derive an in Java space representation (see later). It is the format aspect that imposes a requirement on the Connector Binding. It has to provide a format binding extension to capture the specific formatting information (see format binding section).

WSDL already takes care of the last two bullets.

A Connector WSDL Sample

```
<?xml version="1.0"?>
<definitionsname="CustomerInfo"
 <types>
  <xsd schema targetNamespace=
     "http://www.customercommandservice.com/CustomerCommand"
     xmlns="http://www.w3.org/1999/XMLSchema/"
    <xsd:complexType name="Customer">
     <xsd:element name="Num"type="xsd:string"/>
     ...
    </xsd:complexType>
  </xsd schema>
 </types>
 <message name="GetCustomerInfoInput">
    <part name="Customer"type="Customer"/>
 </message>
  ...
 <portType name="CustomerInfoPortType">
   <operation name="GetCustomerInfo">
     <input message="GetCustomerInfoInput"/>
     <output message="GetCustomerInfoOutput"/>
   </operation>
 </portType>

 <binding name="CustomerInfoConnectorBinding"type="CustomerInfoPortType">
     <format:typemapping style="COBOL"encoding="COBOL">
       <format:typemap typename="Customer"formattype="/CustomerInfo.ccp:CUSTINF"/>
     </format:typemapping>
    <operation name="GetCustomerInfo">
     <cics:operation functionName="GETCUST"/>
     <input>
       ...
     </input>
     <output>
       ...
     </output>
   </operation>
 </binding>
 <service name="CustomerServices">
    <port name='CICS_A'binding='CustomreInfoConnectorBinding'>
     <cics:address connectionURL="...."serverName="CICS_A"/>
    </port>
 </service>
</definition>
```

```
CustomerInfoTypes.cbl
...
    01  CUSTINF.
        02 Num          PIC X(8).
        02 FirstName    PIC X(20).
        02 LastName     PIC X(20).
...
```

How the Connector Binding Extends WSDL

```
<definitions .... >
   <binding ...>
      <connector:binding />
         format:typeMapping encoding="..." style="...">
            <format:typeMap
               typeName="..."
               formatType="..." /> *
```

```
        </format:typeMapping>
      <operation .... >
         <connector:operation functionName="name"...
           interaction attributes ... />
          <input>
             ...
          </input>
          <output>
             ...
          </output>
          <fault>*
             ...
          </fault>
      </operation>
   </binding>
   <port .... >
      <connector:address hostName="uri" portNumber="..."
         ...connection attributes ... />
   </port>
</definitions>
```

connector:binding

The purpose of the connector binding is to signify that the binding is bound to a J2EE Connector Architecture based connector.

connector is the short name for the namespace that identifies the particular connector, e.g.

```
<cics:binding />
```

connector:operation

The connector operation contains the InteractionSpec attributes that are necessary to execute the operation on the EIS side, e.g.

```
<cics:operation functionName="GETCUST" />
```

connector:address

The connector address contains the ManagedConnectionFactory attributes that are necessary to configure the connection factory, e.g.

```
<cics:address connectionURL="..." serverName="..." />
```

Format Binding

format:typeMapping

The format typemapping identifies the style and encoding of the native types. The typemapping contains format typemaps which associate the logical format (i.e. XML Schema) with the native format.

The native format is identified by a format type identifier.

Here a sample where the native format type is described by COBOL:

```
<format:typeMapping encoding="COBOL" style="COBOL" >
   <format:typeMap typename="Customer"
         formatType="/CustomerInfo.ccp:CUSTINF"/>
</format:typeMapping>
```

Create an EIS-specific Connector Binding

To create an EIS-specific connector binding using the WSDL extensions implementation, you would follow the general directions discussed previously and see the sample at the end of this article.

Chapter 3. Web Service Invocation Framework (WSIF)

Having a standard way for describing the services that reside in an EIS using WSDL allows to simplify the client programming model from where we are today with JCA CCI.

The Web Service Invocation Framework (WSIF) represents this simplification. It is a WSDL based service invocation runtime, that allows you to invoke any kind of service, e.g. SOAP, JCA, ...

For more information on WSIF see:

- http://www-106.ibm.com/developerworks/library/ws-wsif.html
- http://www-106.ibm.com/developerworks/webservices/library/ws-wsif2/index.html

Client Programming Model

The client programming model for WSIF follows the pattern that was established for the usage of resources in J2EE. Note that the setup of the port factory is shown inline, it would normally be looked up in JNDI to which it would have been configured at deployment time.

```
WSIFPortFactory portFactory = new WSIFDynamicPortFactory(
                                  "...CustomerInfo1.wsdl",
                                  <service ns>,  "CustomerInfoService",
                                  <portType ns>, "CustomerInfoPortType");

WSIFPort port = portFactory.getPort();
WSIFOperation operation = port.createOperation("getCustomerInfo");

WSIFMessage inputMessage = operation.createInputMessage();
WSIFMessage outputMessage = operation.createOutputMessage();

CustomerInfo input = new CustomerInfo();
input.setCustomerNumber("44444");
inputMessage. setObjectPart ("InputRecordPart", input);

operation.execute(inputMessage, outputMessage, null);

CustomerInfo output =
        CustomerInfo)outputMessage.getObjectPart("OutputRecordPart");
System.out.println(output.getCustomerNumber());
System.out.println(output.getFirstName());
System.out.println(output.getLastName());

port.close();
```

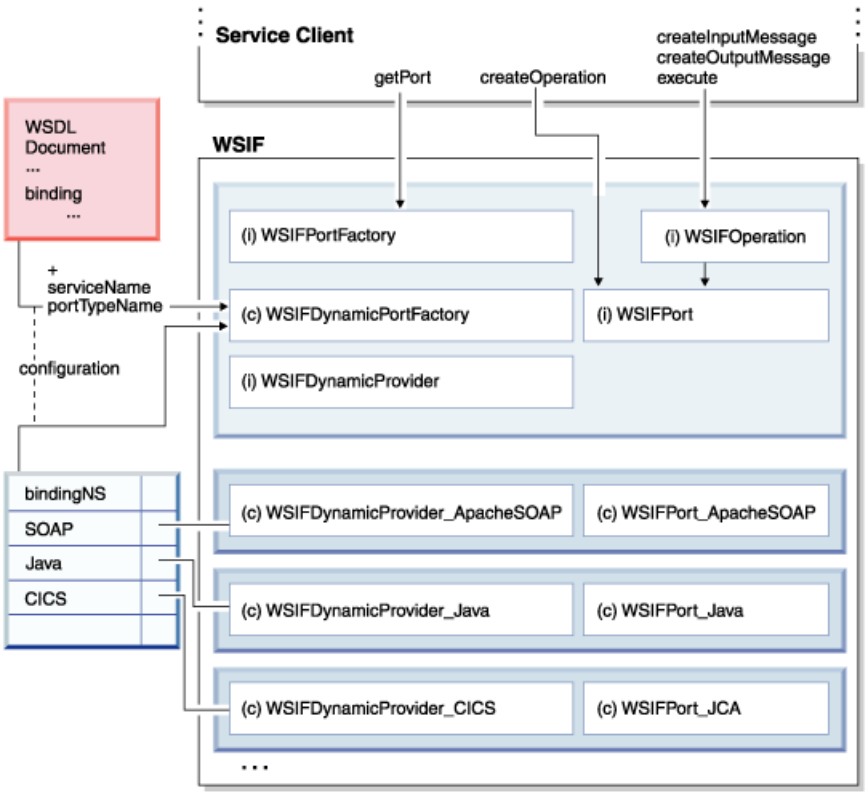
In order to get an intuitive user experience the model terminology is also carried though in the runtime interfaces. The nice thing is that the client programming model doesn't contain any binding specifics anymore. So you can change the binding of the service without impacting the service client.

Architecture

The first thing that you need in setting up for a service invocation is a port. Ports get factored from port factories (i.e. implementations of the WSIFPortFactory interface). WSIF ships with a base port factory implementation called WSIFDynamicPortFactory. On creation this factory gets configured with the WSDL document, the service name, and the portType name from which you want to access an operation.

There is another aspect that the factory gets configured with which are binding specific dynamic providers (i.e. implementations of the WSIFDynamicProvider interface). These dynamic providers are the actual factories of the binding specific ports (i.e. implementations of the WSIFPort interface)

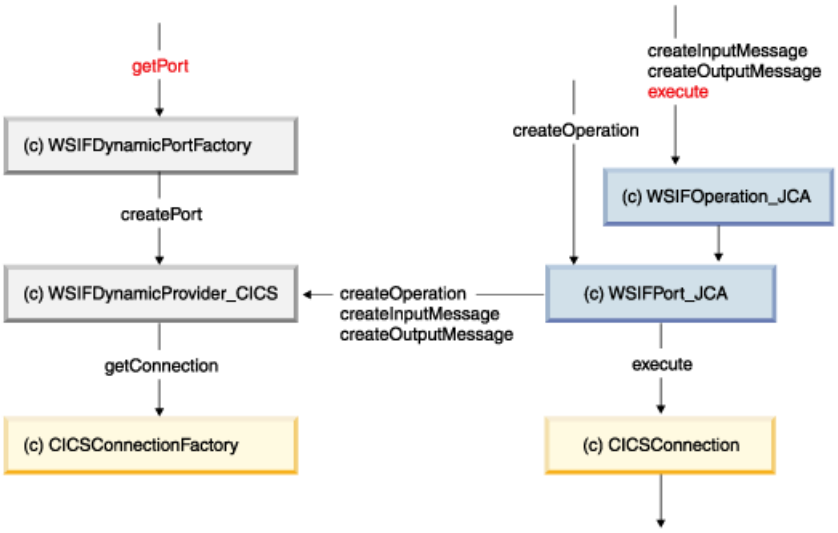
As a client you don't know with which actual port implementation you are dealing with, all you know is that it is a WSIFPort . From it you create the specific operation that you want to invoke, and drive the execution of it.



WSIF and JCA

Enabling a JCA connector for WSIF is very straightforward. The element that you have to implement is a dynamic provider, and depending on your native format a specific message implementation.

In general you don't have to provide a specific port and operation implementation, we provide a generic one that is build with the assumption that the connector implements the JCA CCI.



If your connectors native format is stream based then you can use the provided JCAStreamableMessage as your message implementation. This class extends a class named JCAMessage. If you have to create a specific message implementation you should extend JCAMessage.

Create WSIF extensions for your JCA connector

To create WSIF extentions for your JCA connector, follow these steps:

1. Create a dynamic provider.
2. Optionally, create your own message implementation extending JCAMessage.
3. Optionally, create your own port and operation implementation.

See the sample at the end of the article to add these extensions.

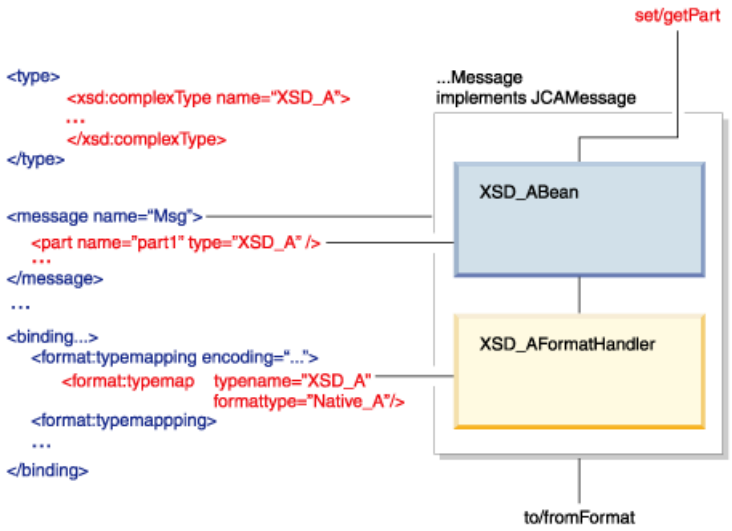
Chapter 4. Format Handling

Format handling is about marshalling the Java representation of a data structure described by XML Schema to/from its binding dependent native format. Separating Java representation and format handling enables late binding, and prevents the service client logic from using objects that are binding specific.

FormatHandler

On service invocation messages (input, output, fault) get exchanged with the provider of the service. Messages consist out of parts that are typed. In order for the provider to understand them the invocation runtime has to transform them into the providers native format.

Looking at the WSDL you see that a binding section defines type mappings that map the XML Schema types to respective native types. So given the meta information from the WSDL you can generate two runtime elements. One is a bean as the Java representation of the structure described by XML Schema. The other is a FormatHandler which gets generated based on the defined format typemapping.



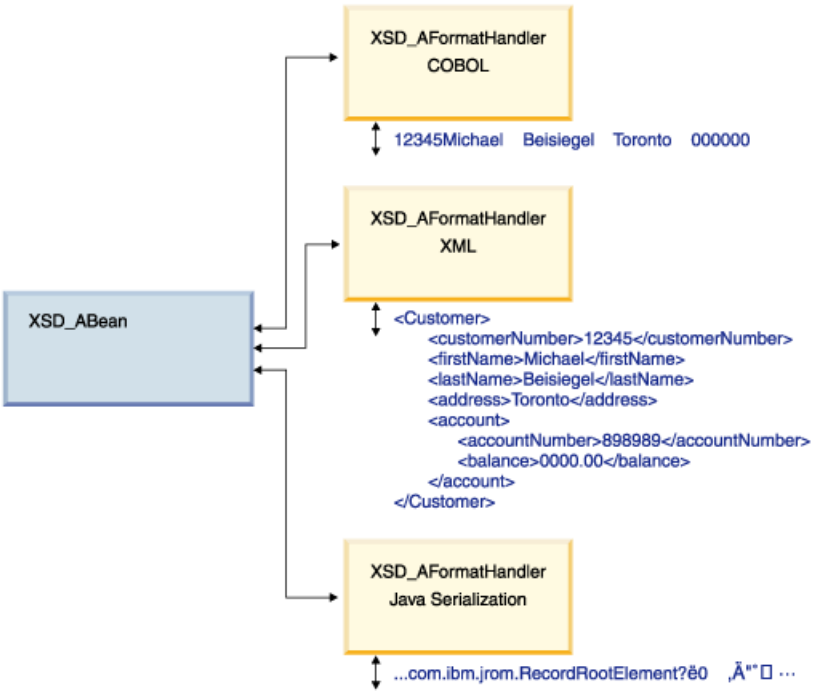
In order for a message implementation to produce its native format it uses the format handlers for its respective part types. The message is a generic implementation for a particular provider. Now how does the message know which format handler to use for its parts. A runtime message knows about the meta information of the concrete message it was factored for. So it knows what type its parts have. These types have qualified names (i.e. namespace and localname). We defined a rule on how to construct the name of the required format handler.

The rule for constructing the name of the format handler looks as follows:

<fliped xsd typenamespace>.<binding shortname>.<format encoding+style>.<xsd typelocalname>"FormatHandler"

The JCAUtil class provides you with a set of utility methods that you can use in your message implementation as well as in the implementation of the format handler generator.

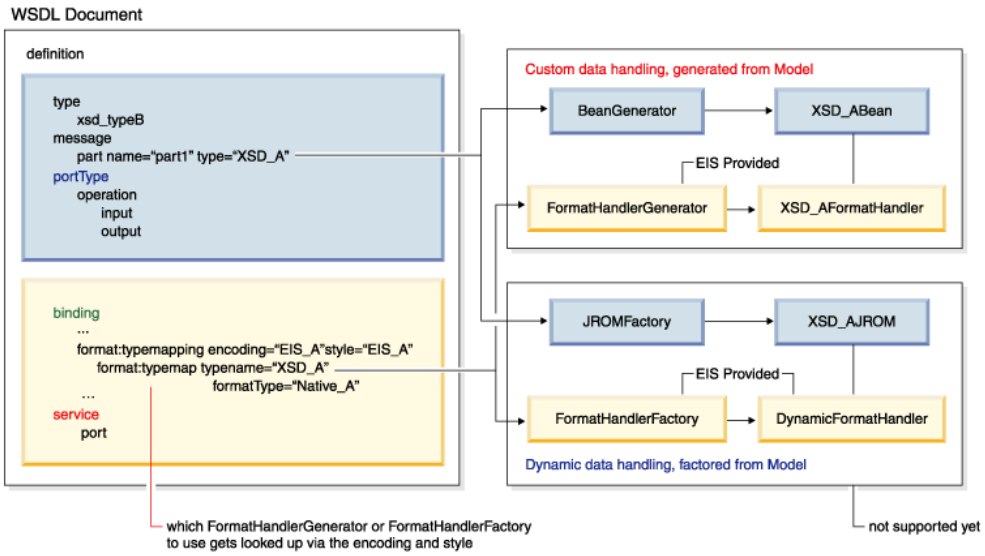
To illustrate the power of separating Java representation from marshalling (in contrast to what we have today with Records in JCA), the following picture shows the same bean bound in three different ways, namely COBOL, XML, and Java serialization.



Besides being able to handle a bean derived from the XSD Schema type, the format handler also has to support instance of the XSD Schema type in form of a JDOMSource or SAXSource. This allows for direct usage of XML in your service invocation if required.

FormatHandler Generator

To support your native formats you have to provide format handler generators with your plug-in. Each format handler generator generates format handlers for a specific encoding and style.



Note that the architecture also allows for dynamic format handling which will be supported at a later point in time.

Create the FormatHandler Generator for your EIS

To create the FormatHandler generator for your EIS, follow these steps:

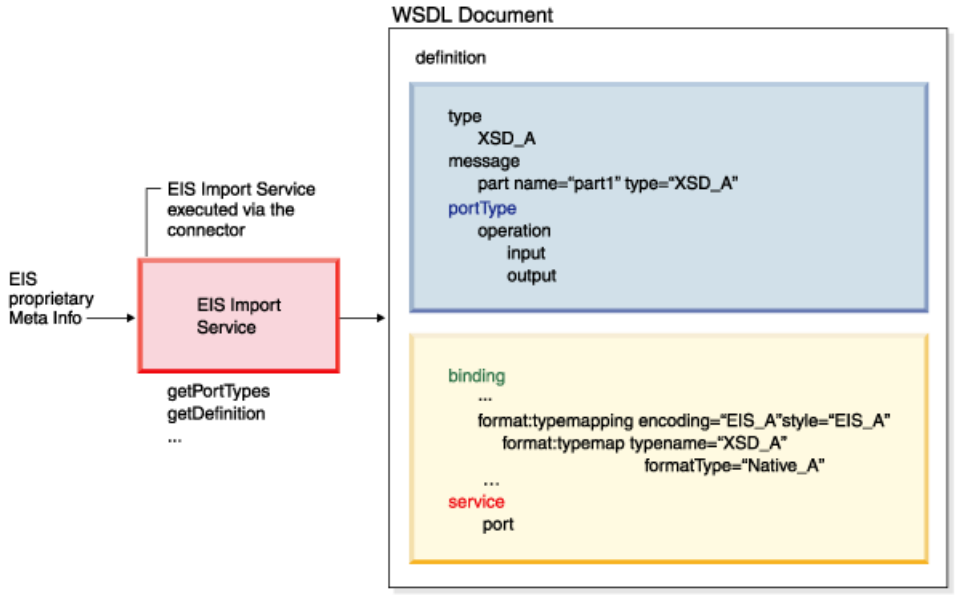
1. Create a format handler generator.
2. It will generate format handlers accordingly that will implement the WSIFFormatHandler interface.
3. The name must be based on the rule defined previously.
4. The generator handles the following parts: Bean, JDOMSource, SAXSource.

See the sample to learn how to create the FormatHandler generator.

Chapter 5. Import Service

Many EIS's have very rich metadata support describing the services they offer. EIS's provide programmatic access to this meta information. Not only is the form that the metadata is in proprietary, also the access to it is different for each EIS.

The EIS import service is the solution to this problem, it provides a standard interface for accessing the meta information, and it delivers it in form of WSDL. You implement the import service with your EIS connector.



Interface

The Import interface consists of two operations *getPortTypes*, and *getDefinition*.

The *getPortTypes* operation allows you to get an overview about the interfaces and operations the EIS offers. The operation returns an array of portTypes, the number of portTypes returned can be controlled via the queryString input argument (supporting the queryString is optional).

Note, if your EIS doesn't have the notion of interfaces you can just return one portType containing all the operations your EIS offers.

The *getDefinition* operation allows you to retrieve the complete service definition for a chosen portType selection. Besides selecting the portType, the portType selection allows to subset the portType by identifying the operations that you are interested in. The operation returns the WSDL definition, and an array of XML Schema sources for the case the portType uses XML Schema complex types.

Import.wsdl file

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ImportRemoteInterface"
    targetNamespace="http://importservice.jca.ibm.com/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://importservice.jca.ibm.com/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <import location="Import.xsd"
                  namespace="http://importservice.jca.ibm.com/"/>
    <message name="getDefinitionRequest">
         <part name="portTypeSelection" type="tns:PortTypeSelection"/>
    </message>
    <message name="getDefinitionResponse">
         <part name="result" type="tns:ImportDefinition"/>
    </message>
    <message name="getPortTypesRequest">
         <part name="queryString" type="xsd:string"/>
    </message>
    <message name="getPortTypesResponse">
         <part name="result" type="tns:PortTypeArray"/>
    </message>
    <portType name="Import">
         <operation name="getDefinition"
                    parameterOrder="portTypeSelection">
              <input message="tns:getDefinitionRequest"
                          name="getDefinitionRequest"/>
              <output message="tns:getDefinitionResponse"
                          name="getDefinitionResponse"/>
         </operation>
         <operation name="getPortTypes" parameterOrder="queryString">
              <input message="tns:getPortTypesRequest"
                          name="getPortTypesRequest"/>
              <output message="tns:getPortTypesResponse"
                          name="getPortTypesResponse"/>
         </operation>
    </portType>
</definitions>
```

Import.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema attributeFormDefault="qualified" elementFormDefault="qualified"
    targetNamespace="http://importservice.jca.ibm.com/"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd1="http://importservice.jca.ibm.com/"
                        xmlns:xsd2="http://xml.apache.org/xml-soap">
    <import namespace="http://xml.apache.org/xml-soap"
            schemaLocation="http://xml.apache.org/xml-soap"/>
    <import namespace="http://schemas.xmlsoap.org/wsdl/"
            schemaLocation="http://schemas.xmlsoap.org/wsdl/"/>
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"
            schemaLocation="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="ImportDefinition">
        <all>
```

```
        <element name="importXSDs" type="xsd1:ArrayOfImportXSD"/>
        <element name="definition" type="xsd1:javax.wsdl.Definition"/>
    </all>
</complexType>
<complexType name="ImportXSD">
     <all>
         <element name="location" type="string"/>
         <element name="namespace" type="string"/>
         <element name="source" type="string"/>
     </all>
</complexType>
<complexType name="ArrayOfImportXSD">
    <complexContent>
        <restriction base="soapenc:Array">
             <sequence/>
             <attribute ref="soapenc:arrayType"
                           wsdl:arrayType="xsd1:ImportXSD[]"/>
        </restriction>
    </complexContent>
</complexType>
<complexType name="javax.wsdl.Definition">
     <all>
         <element name="imports" type="xsd2:Map"/>
         <element name="bindings" type="xsd2:Map"/>
         <element name="namespaces" type="xsd2:Map"/>
         <element name="documentBase" type="xsd1:java.net.URL"/>
         <element name="documentationElement" type="xsd1:anyElement"/>
         <element name="portTypes" type="xsd2:Map"/>
         <element name="QName" type="xsd1:javax.wsdl.QName"/>
         <element name="messages" type="xsd2:Map"/>
         <element name="extensionRegistry"
                type="xsd1:javax.wsdl.extensions.ExtensionRegistry"/>
         <element name="targetNamespace" type="string"/>
         <element name="services" type="xsd2:Map"/>
         <element name="extensibilityElements"
                type="xsd1:java.util.List"/>
         <element name="typesElement" type="xsd1:anyElement"/>
     </all>
</complexType>
<complexType name="java.net.URL">
    <all>
        <element name="ref" type="string"/>
        <element name="port" type="int"/>
        <element name="file" type="string"/>
        <element name="authority" type="string"/>
        <element name="content"/>
        <element name="query" type="string"/>
        <element name="URLStreamHandlerFactory"
               type="xsd1:java.net.URLStreamHandlerFactory"/>
        <element name="userInfo" type="string"/>
        <element name="protocol" type="string"/>
        <element name="host" type="string"/>
        <element name="path" type="string"/>
        </all>
</complexType>
<complexType name="java.net.URLStreamHandlerFactory">
```

```
    <all/>
</complexType>
<complexType name="anyElement">
    <sequence>
        <any processContents="skip"/>
    </sequence>
</complexType>
<complexType name="javax.wsdl.QName">
    <all>
        <element name="localPart" type="string"/>
        <element name="namespaceURI" type="string"/>
    </all>
</complexType>
<complexType name="javax.wsdl.extensions.ExtensionRegistry">
    <all/>
</complexType>
<complexType name="java.util.List">
    <all>
        <element name="empty" type="boolean"/>
    </all>
</complexType>
<complexType name="PortTypeSelection">
    <all>
        <element name="operationSelection"
               type="xsd1:ArrayOfOperationSelection"/>
        <element name="portTypeQName"
               type="xsd1:javax.wsdl.QName"/>
    </all>
</complexType>
<complexType name="OperationSelection">
    <all>
        <element name="operationName" type="string"/>
        <element name="inputName" type="string"/>
        <element name="outputName" type="string"/>
    </all>
</complexType>
<complexType name="ArrayOfOperationSelection">
    <complexContent>
        <restriction base="soapenc:Array">
            <sequence/>
            <attribute ref="soapenc:arrayType"
                  wsdl:arrayType="xsd1:OperationSelection[]"/>
        </restriction>
    </complexContent>
</complexType>
<complexType name="PortTypeArray">
    <all>
        <element name="portTypes"
               type="xsd1:ArrayOfJavax.wsdl.PortType"/>
    </all>
</complexType>
<complexType name="javax.wsdl.PortType">
    <all>
        <element name="QName" type="xsd1:javax.wsdl.QName"/>
        <element name="operations"
               type="xsd1:java.util.List"/>
```

```
            <element name="documentationElement"
                   type="xsd1:anyElement"/>
            <element name="undefined" type="boolean"/>
        </all>
    </complexType>
    <complexType name="ArrayOfJavax.wsdl.PortType">
        <complexContent>
            <restriction base="soapenc:Array">
                <sequence/>
                <attribute ref="soapenc:arrayType"
                     wsdl:arrayType="xsd1:javax.wsdl.PortType[]"/>
            </restriction>
        </complexContent>
    </complexType>
</schema>
```

Implementation

You implement the Import Service with your EIS connector; this includes providing your connectors respective import service WSDL binding definition.

The way that this service is implemented is dependent on your EIS. In the case of a tailorable EIS the implementation would do a live access to the EIS's metadata repository. In the case of a non-tailorable EIS the definitions could be shipped as part of the RAR and your import service implementation would access them from there.

Create the Import Service for your EIS

To create the import service for your EIS, follow these steps:

1. Create the import service WSDL binding definition.
2. Implement the service with your connector.

See the sample to learn how to create the import service.

Chapter 6. Packaging - Putting it all together

This chapter shows you how you package up your tool extensions in your connector RAR. You declare the things provided to the tool environment by means of a XML file.

jca_tool_plugin.xml

```
<jca_tool_plugin tns = "http://schemas.xmlsoap.org/wsdl/myeis/" name = "MyEIS">
   <Description>MyEIS</Description>
   <version>0.0.1</version>
   <wsdl_extensions>
      <address classname = "com.ibm.wsdl.extensions.jca.myeis.MyEISAddress" />
      <binding classname = "com.ibm.wsdl.extensions.jca.myeis.MyEISBinding"  />
      <operation classname = "com.ibm.wsdl.extensions.jca.myeis.MyEISOperation" />
      <extension_registry
         classname = "com.ibm.wsdl.extensions.jca.myeis.MyEISExtensionRegistry"/>
   </wsdl_extensions>
   <wsif_extensions
      classname = "com.ibm.wsif.providers.jca.myeis.WSIFDynamicProvider_MyEIS" />
   <import>
      <service wsdlfile = "com/ibm/jca/importservice/myeis/ImportMyEIS.wsdl"
          servicename = "ImportService" />
   </import>
   <formathandler>
      < generator encoding ="myeis" style= "myeis"
        classname = "com.ibm.jca.myeis.formathandler.MyEISFormatHandlerGenerator" />
   </formathandler>
</jca_tool_plugin >
```

Putting it into the RAR

The XML file along with a jar of the tool extenision implementation classes has to be packaged into the resource adapters archive file (RAR).

The following shows where to place them in the RAR:

- /META-INF/ra.xml
- /META-INF/jca_tool_plugin.xml
- /howto.html
- /images/icon.jpg
- /ra.jar
- /cci.jar
- /jca_tool_plugin.jar
- /win.dll
- /solaris.so

Chapter 7. Rendering the Tool Plug-in

This section describes how a tool environment can use connectors that support the tool plug-in. First we layout a very generic description, and then we show how the connectors supporting the plug-in can be used in WebSphere(R) Studio Application Developer Integration Edition.

Generic Description

The following is a generic description of the tool task flow for a tool plug-in with and without import service support.

With Import Service

- Ask for connection properties.
- Call the getPortTypes method of the importing service.
- Present portTypes, and allow for selection.
- Call getDefinition method of importing service passing the selection.
- Add returned WSDL definition to your work environment.

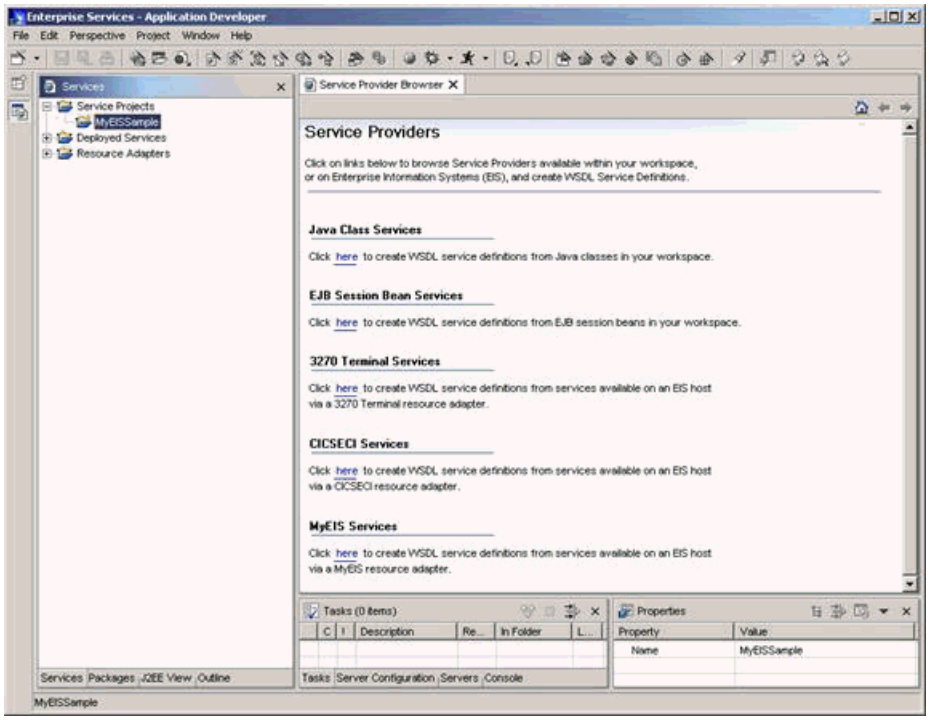
Without Import Service

- Ask for connection properties.
- Ask for portType name.
- Create WSDL definition in your work environment.
- Open the WSDL definition with an editing tool.
- Add operations with their binding to the portType:
  - use existing messages
  - import new messages by importing existing language structures
- Add format typemapping to the binding:
  - specify encoding and style
  - specify format typemaps

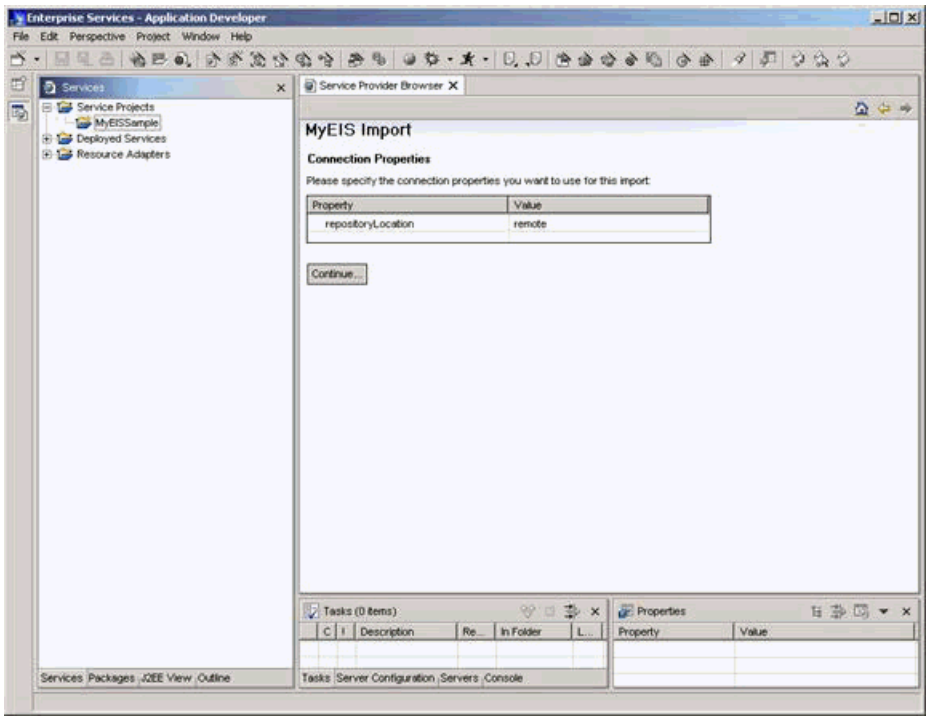
Sample: How a user works with it

With Import Service

MyEIS is a connector that supports the importing service. In the service provider browser you select MyEIS if you want to create a service definition for it.



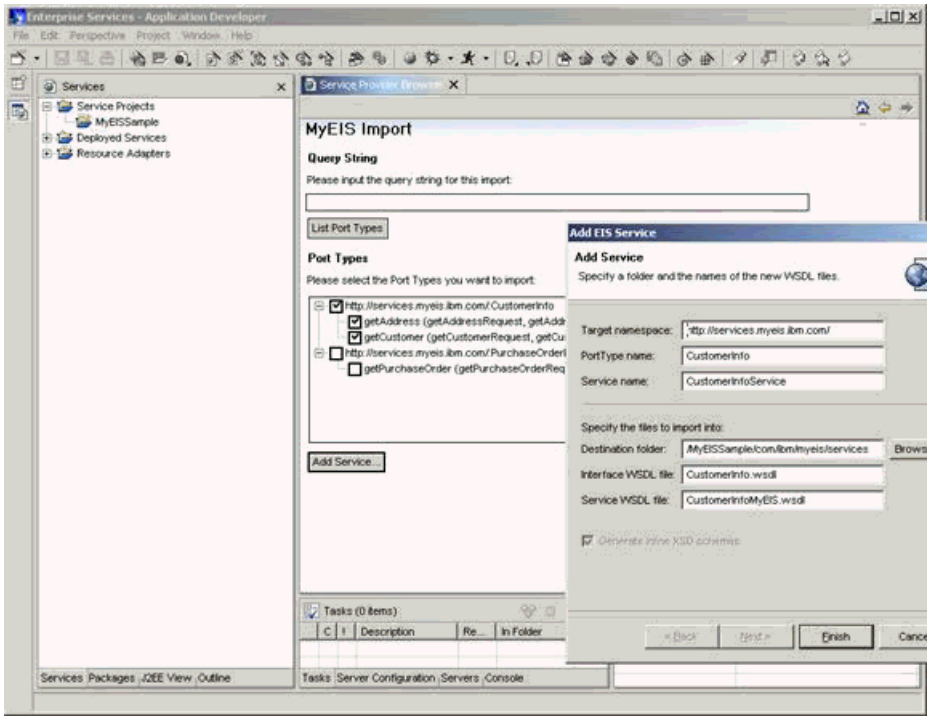
Next you are asked for the connection properties to access the EIS. After setting the properties press **Continue** ....

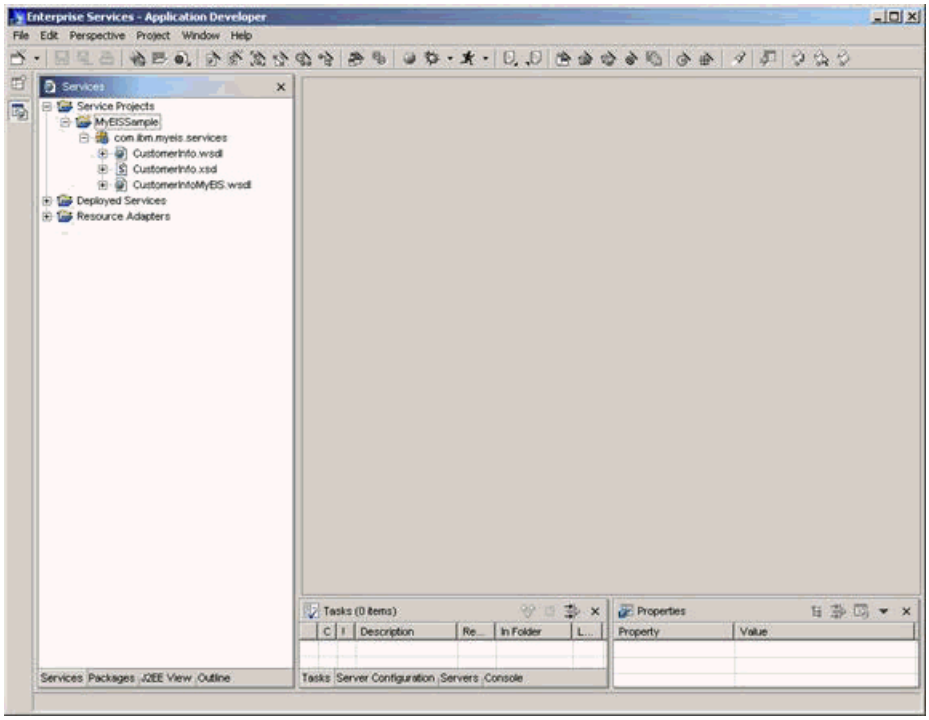


Next you click the **List Port Types** button to query for available portTypes offered by the EIS system. You can use the entry field to specify a query string to restrict the list of portTypes returned. The **List Port Types** button invokes the getPortTypes method of the MyEIS import service.

The list of returned portTypes is displayed and you can select the specific portType, or Operations in the same portType, you want to import. After you made the selection, the **Add Service** button will launch the Add Service Wizard. The **Finish** button of this wizard triggers the getDefinition method of the importing service. The returned definition will be added to the workspace.

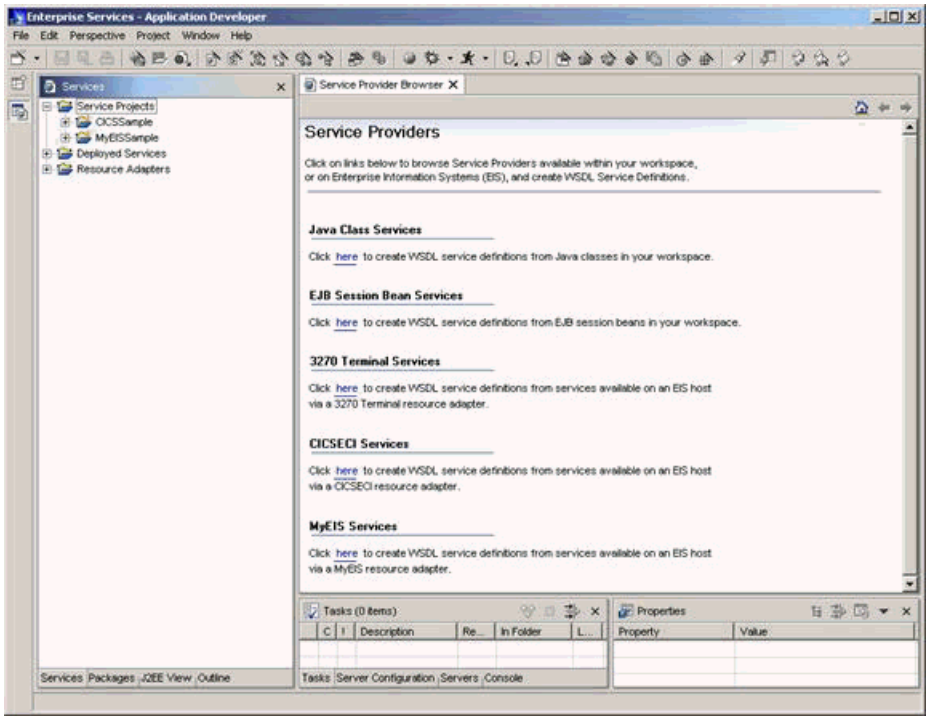
You can repeat the sequence of selecting a portType and adding its definition multiple times.





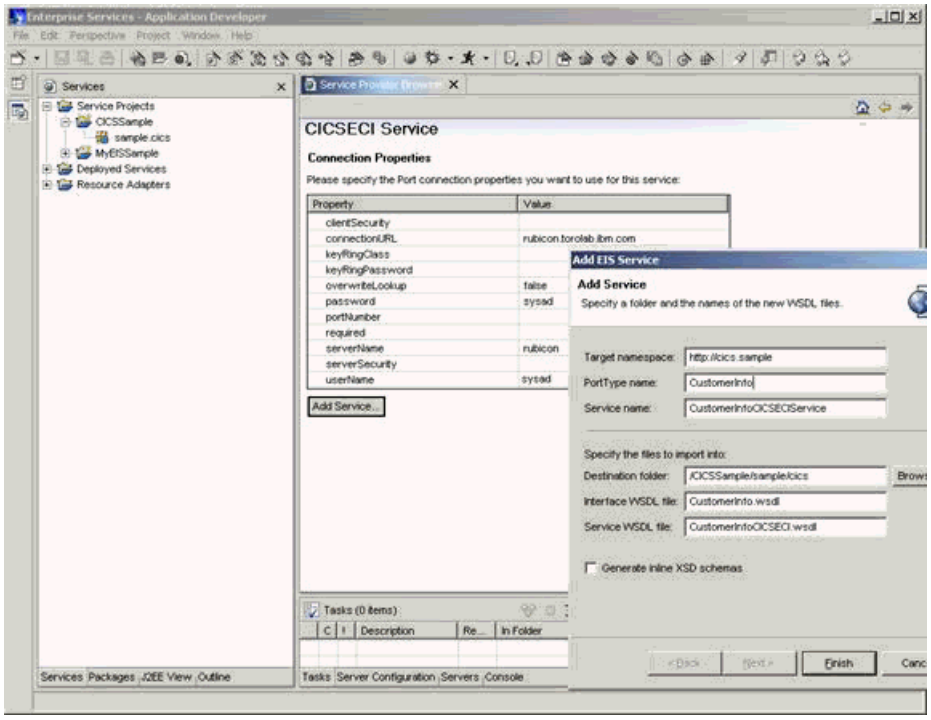
Without Import Service

This is a sample for a connector that doesn't support the importing service, here shown on the CICS(R) ECI connector. Select CICS ECI if you want to create a service definition for a CICS ECI service.

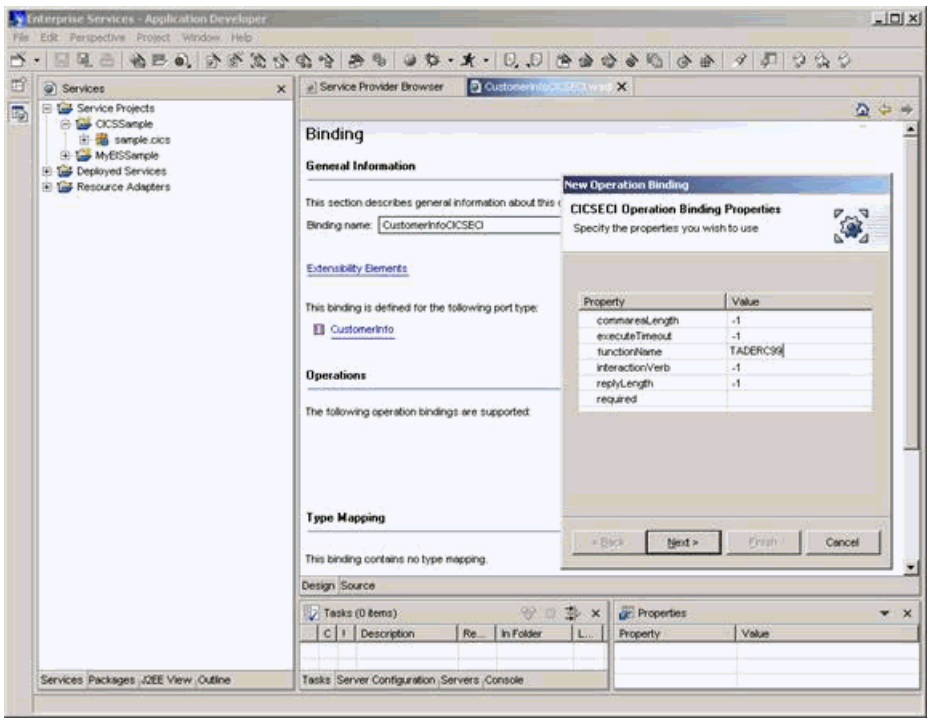


Next you are asked for the connection properties to access CICS. After setting the properties, click **Add Service**. This gets you into the Add Service wizard, where

you specify the namespace and the name of the portType that you want to create. Pressing **Finish** will create a service definition with an empty portType.



For the created service definition the editor will be opened, it allows you to interactively add the operations and their bindings to the portType as well as the format typemapping. Clicking **New** in the operation section takes you to the New Operation Binding wizard.



Chapter 8. Sample: The tool-enabled MyEIS connector, Step by Step...

This sample will walk you through the steps of developing a tool extensions to enable your J2EE Connector Architecture Resource Adapter for the tool environment. The complete source code for the MyEIS connector and its JCA_Tool_Plugin support is included in the "myeis.jar" file, shipped as part of the "myeis.rar" in the following directory:

<install directory>/eclipse/plugins/com.ibm.etools.ctc.binding.eis/runtime

MyEIS Connector

The connector provided as a sample contains the minimal set of classes necessary to execute the interaction with the MyEIS back-end system. It is not meant to provide a sample implementation of the JCA Resource Adapter but only shows the skeleton classes necessary to support Resource Adapter's pluggability into the tools, import and sample services. The Connector consists of the following classes, implementing J2EE Connector Architecture interfaces, in the com.ibm.jca.myeis package:

- MyEISConnection - client view of the managed connection (connection handle)
- MyEISConnectionFactory - factory of connection handles
- MyEISInteraction - object providing methods to execute interactions with EIS
- MyEISInteractionSpec - object encapsulating properties of a specific interaction
- MyEISManagedConnection - physical connection to the backend system. The implementation of this class illustrates two different possibilities of retrieving the list of PortTypes and Definition by the Resource Adapter. The first method is to ship the service (wsdl) files representing available services and definitions with the Resource Adapter. This method would be appropriate for the connector to the EIS with the infrequently changing (for example only between EIS releases) services available. In the sample implementation, the shipped wsdl files are accessed in the com.ibm.myeis.repository.Repository class implementation. The second method to get the services available is by querying the EIS (com.ibm.myeis.MyEIS class in the sample). In that case the Resource Adapter would connect to and query the EIS for available services. The tooling does not make any assumptions about which method is used. The sample connector illustrates support for both methods, switchable through the property on the Managed Connection Factory.
- MyEISManagedConnectionFactory - the factory of physical connections to the EIS system

Create the MyEIS specific Connector Binding

To enable capturing of the meta information characterizing the details of the interactions with the EIS in the wsdl model, the Resource Adapter provides Connector Bindings. These bindings consist of the following:

- extensibility elements for the Binding, Operation and Port
- binding serializer
- binding deserializer

- ExtensionRegistry used to register the serializer and deserializer with the runtime and tools.

The sample connector implements Connector Bindings in the com.ibm.wsdl.extensions.jca.myeis package. The Binding extensibility element is used as a tag (has no properties) and defines the binding type. The extensibility element for the Operation captures the meta information describing a single interaction with the EIS system. The properties of this extensibility element correspond to the properties of the Resource Adapter's InteractionSpec. The Port extensibility element represents the endpoint (address) where the service is available. It contains properties corresponding to the properties of the ManagedConnectionFactory. To implement Extensibility Element, create class implementing the javax.wsdl.extensions.ExtensibilityElement and java.io.Serializable interfaces and add appropriate properties from InteractionSpec to the Operation extensibility element and from Managed Connection Factory to Port extensibility element. Next, you need to select the namespace for your bindings. The sample connector uses the following namespace URI: http://schemas.xmlsoap.org/wsdl/myeis/ with the standard prefix, followed by the connector specific last segment. The element names are left to the implementor, however the ones used in the sample connector (MyEISBinding, MyEISAddress, MyEISOperation) make the wsdl files easier to read. The next step is to provide a serializer and deserializer for your bindings. This is necessary since the wsdl file reader and writer cannot handle arbitrary extensibility elements. The serializer writes the xml format of your extensibility elements, for example Operation, the deserializer reads in and parses xml, creating from it instances of these elements. For example to serialize MyEISOperation object from the sample connector, you need to create the following xml fragment:

```
<myeis:operation functionName="CUSTOMERINFO_getAddress" />
```

with the following code:

```
if (extension instanceof MyEISOperation) {
                MyEISOperation operation = (MyEISOperation) extension;
                pw.print( "      <"  +  TPrefix +  ":operation" );

                if (operation.getFunctionName() !=  null ) {
                        DOMUtils.printAttribute( "functionName" ,
                        operation.getFunctionName(), pw);
                }
```

To deserialize the xml fragment, use the following:

```
if (MyEISBindingConstants.Q_ELEM_OPERATION.equals(elementType)) {
                MyEISOperation operation =  new  MyEISOperation();

                String functionName = DOMUtils.getAttribute(el, "functionName" );
                if (functionName !=  null ) {
                        operation.setFunctionName(functionName);
                }
                return operation;
```

The last component that you need to provide with your bindings is the Extension Registry. Its purpose is to provide convenient way to register serializer and deserializer for your extensibility elements with tools and runtime. The class implements the javax.wsdl.extensions.ExtensionRegistry interface and registers serializer/deserializer for each element in its constructor using the following code:

```
MyEISBindingSerializer ser =  new  MyEISBindingSerializer();
```

```
// operation
this .registerSerializer(javax.wsdl.BindingOperation.class,
   MyEISOperation. class , ser);
this .registerDeserializer(javax.wsdl.BindingOperation.class,
   MyEISBindingConstants.Q_ELEM_OPERATION, ser);
```

Create the WSIF extensions for MyEIS Connector

The WSIF requires that every binding to be executed has to provide the set of supporting classes implementing binding runtime. JCA Tool Plug-in provides elements of the runtime and delegates to the specific connector only these elements that cannot be handled in a generic manner i.e. elements corresponding to the extensibility elements provided by the connector, Address and Operation.

The JCA Tool Plug-in implements JCAMessage, JCAPort and JCAOperation and requires the resource adapter to provide WSIFDynamicProvider and WSIFDynamicProviderJCAExtensions implementations.

The WSIF uses WSIFDynamicProvider to create a dynamic port corresponding to the endpoint at which the service is accessible. This, in turn, corresponds to the creation of the Connection to the EIS. The resource adapter implements the WSIFDynamicProvider and WSIFDynamicProviderJCAExtensions interfaces. In its createDynamicWSIFPort method, it creates the WSIFPort_JCA (part of the JCA Tool Plug-in run time) initialized with the active javax.resource.cci.Connection. The resource adapter creates the connection either using JNDI lookup or directly using Managed Connection Factory. The name used for the lookup is built from the service namespace URI, service name and port name. The JCA Tool Plug-in run time provides a class (com.ibm.wsif.jca.util.JCAUtil) with a set of utility methods that should be used to construct names (for example, JNDI lookup name, Format Handler class name). In the required implementation of the createOperation method the resource adapter creates JCAOperation and initializes is with the instance of Connection and InteractionSpec as shown below:

```
BindingOperation bindingOperationModel =
   aBinding.getBindingOperation(aOperationName, aInputName, aOutputName);

ExtensibilityElement bindingOperationModelExtension =
  (ExtensibilityElement) bindingOperationModel.getExtensibilityElements().get(0);
if
  (bindingOperationModelExtension ==  null ) {
       throw new  WSIFException( "missing bindingOperation extension" );
}
if (!(bindingOperationModelExtension  instanceof  MyEISOperation)){
       throw new  WSIFException( "invalid extensibility element" );
}

MyEISOperation operationModelExtension = (MyEISOperation)
   bindingOperationModelExtension;

MyEISInteractionSpec interactionSpec = new MyEISInteractionSpec();

interactionSpec.setFunctionName(operationModelExtension.getFunctionName());

operation =  new JCAOperation(aDefinition, aBinding, aOperationName, aInputName,
   aOutputName, aConnection, interactionSpec,  this );
```

Optionally, the Resource Adapter can provide implementation of the Message creation methods to be able to create customized messages. The default implementation uses the Streamable interface to exchange input and output data with the Connector.

Create the FormatHandler Generator for MyEIS

The connector provides the format handler generator by implementing com.ibm.jca.formathandler.FormatHandlerGenerator interface. During connector deployment into the tools, the format handler generator is registered, based on the encoding and style it specifies in the xml file. When the service definition is deployed and the helper classes need to be generated, the encoding and style are used to lookup the format handler generator and invoke it from tooling. The generate method is passed the following arguments:

- generationPackage - the fully qualified package that the generator should use to generate its classes in. This name follows the naming convention described earlier in the document.
- beanClass - the fully qualified name of the Java bean class that represents the XSD type (or element) for which a format handler is being generated.
- aDefinition - the WSDL definition from which the XSD type is derived.
- aBinding - the WSDL binding element that contains the type mapping for the XSD type that is being used.
- anEncoding - the encoding attribute off the TypeMapping element associated with the format handler to be generated.
- aStyle - the style attribute off the TypeMapping element associated with the format handler to be generated
- xsdQname - the QName of the XSD element (or type) for which the format handler is being generated.
- elementType - the type of element represented by the xsdQname parameter. This is either CustomFormatHandlerGenerator.XSD_TYPE or CustomFormatHandlerGenerator.XSD_ELEMENT

The format handler generator returns a HashTable that contains a set of java.io.InputStreams. Each input stream contains the generated code for a particular class generated by the format handler generator. The number of input streams returned depends on the number of classes that are generated by the format handler generator. The keys of the HashTable must be the name of each class (unqualified) that is represented by its corresponding InputStream value. The class name keys are unqualified because they must all be in the generationPackage that is passed in as an input parameter.

The arguments to the generate method may contain sufficient information to generate the specific format handler i.e. the xsd type of the part and assumed encoding and style since these were used to locate and invoke the generator. In some cases, additional information may be required. In this case, the TypeMappings from the passed bindings can be used:

```
TypeMapping typeMapping =  null ;
Iterator iterator = aBinding.getExtensibilityElements().iterator();
while (iterator.hasNext()){
         Object element = iterator.next();
         if (element instanceof TypeMapping){
                  typeMapping = (TypeMapping)element;
                   break ;
         }
}
```

When the type mappings are used, the generator should verify that type mapping has an encoding, and optionally a style it supports e.g.

```
if (!typeMapping.getEncoding().equals( "myeis" ))
        return null ;
```

Finally, the generator could retrieve from the format binding the formatType string corresponding to the xsd (type or element) for which the format handler is being generated:

```
String formatType = null ;
iterator = typeMapping.getMaps().iterator();
while (iterator.hasNext()){
         TypeMap typeMap = (TypeMap)iterator.next();
         if (typeMap.getTypeName().equals(xsdQname)){
                  formatType = typeMap.getFormatType();
                  break ;
         }
}
```

The formatType is a string and tools do not make any assumption about it contents. It is a specific value understood by the generator. In the sample generator, the native format for all types is Java serialization and therefore the formatType string is empty.

The sample generator: com.ibm.jca.myeis.formathandler. MyEISFormatHandlerGenerator does not use typeMaps since the native format it generates is a simple Java serialization and therefore formatTypes in the wsdl files are empty.

In addition to handling the associate part as a Java bean, the format handler should have a capability to read and write the native format to and from the SAXSource and JDOMSource. This allows the invoker of the service to pass or retrieve the data as XML. The sample format handler generator shows one of the possible methods to convert a native format, in its case a Java bean, to the source format.

Create the Import Service for MyEIS

The resource adapter supporting the import service is required to provide, in addition to the implementation described in the section “MyEIS Connector” section above, the following: a service file with the bindings for the Import interface from the JCA Tool Plug-in and a set of format handlers for the conversion to native format during execution of the import service. The sample connector binding file is as follows:

```
<binding name="ImportMyEISBinding" type="importservice:Import">
    <myeis:binding/>
   <operation name="getDefinition">
        <myeis:operation functionName="IMPORT_DEFINITION"/>
        <input name="getDefinitionRequest"/>
        <output name="getDefinitionResponse"/>
   </operation>
   <operation name="getPortTypes">
        <myeis:operation functionName="IMPORT_PORTTYPES"/>
        <input name="getPortTypesRequest"/>
        <output name="getPortTypesResponse"/>
   </operation>
</binding>
<service name="ImportService">
    <port binding="tns:ImportMyEISBinding" name="ImportMyEISPort">
        <myeis:address repositoryLocation="remote"/>
    </port>
</service>
```

It contains all the information needed to execute import service, for example function names (in the operation binding) and the repository location (in the port binding). The user will be able to edit these values if necessary, for example if the port binding contains the location of the target EIS system.

The format handlers necessary to execute the import service are no different than any other format handlers generated by the Resource Adapter and the format handler generator can be used to create them. The sample connector provides all the elements needed for the import service in the com.ibm.jca.importservice.myeis package.

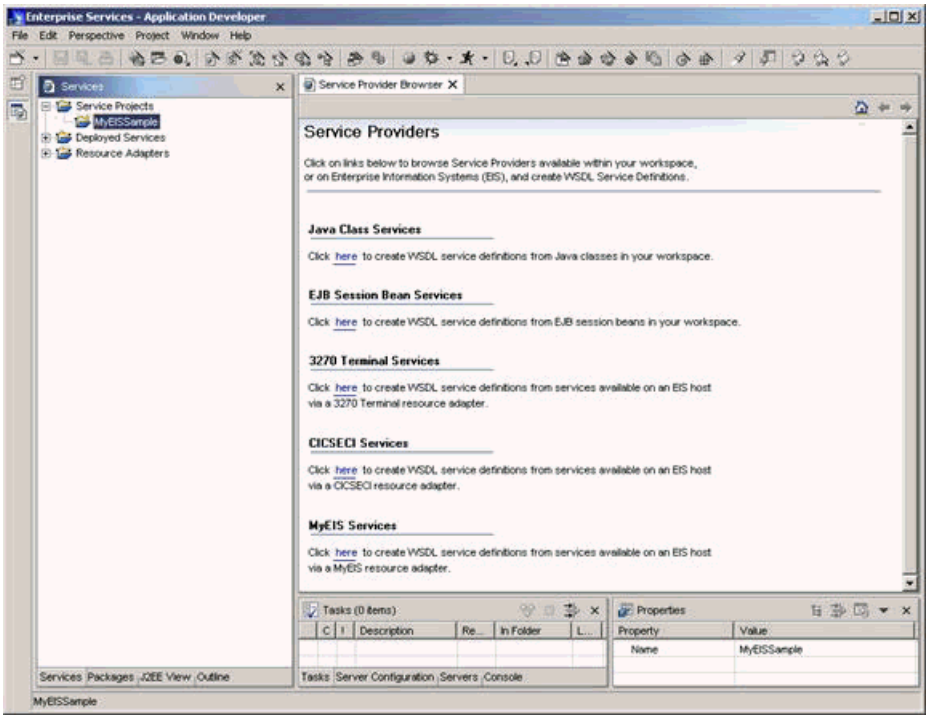
Packaging the MyEIS Connector together with the Tool Extensions

The sample MyEIS Connector is packaged into myeis.rar, a resource adapter archive file. It contains the "META-INF/jca_tool_plugin.xml" file, "ra.xml" and one jar with the connector runtime and tool extensions. The rar file can contain more than one jar file so for example when the tool extensions are added to the existing connector, they could be packaged in the separate jar.

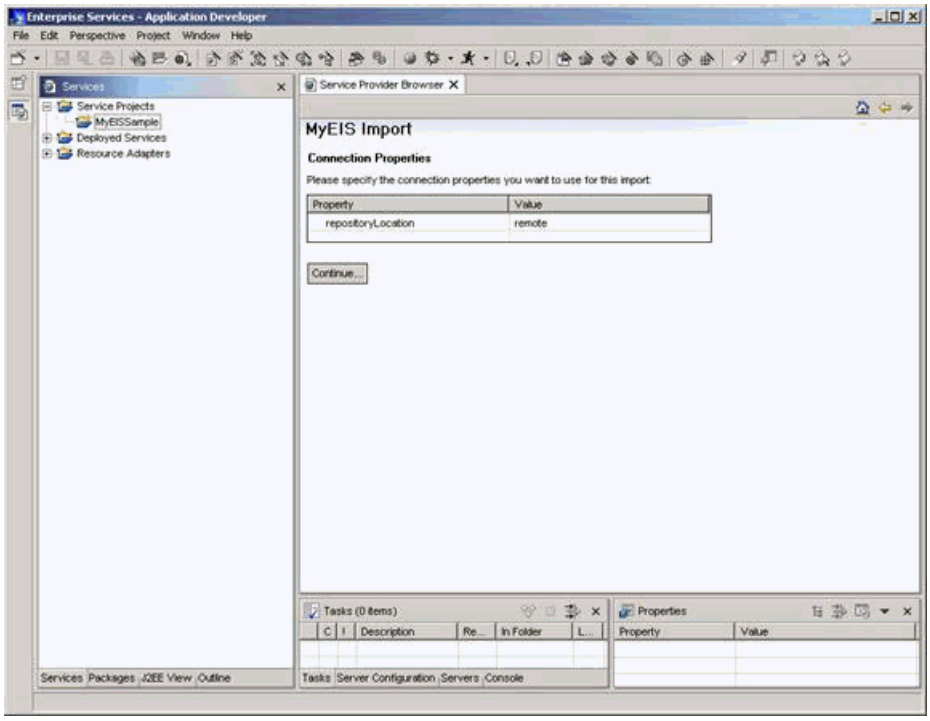
Developing with the MyEIS Connector

The following section demonstrates how the 'CustomerInfo' portType can be imported from the MyEIS connector, and a proxy class be created for this service.

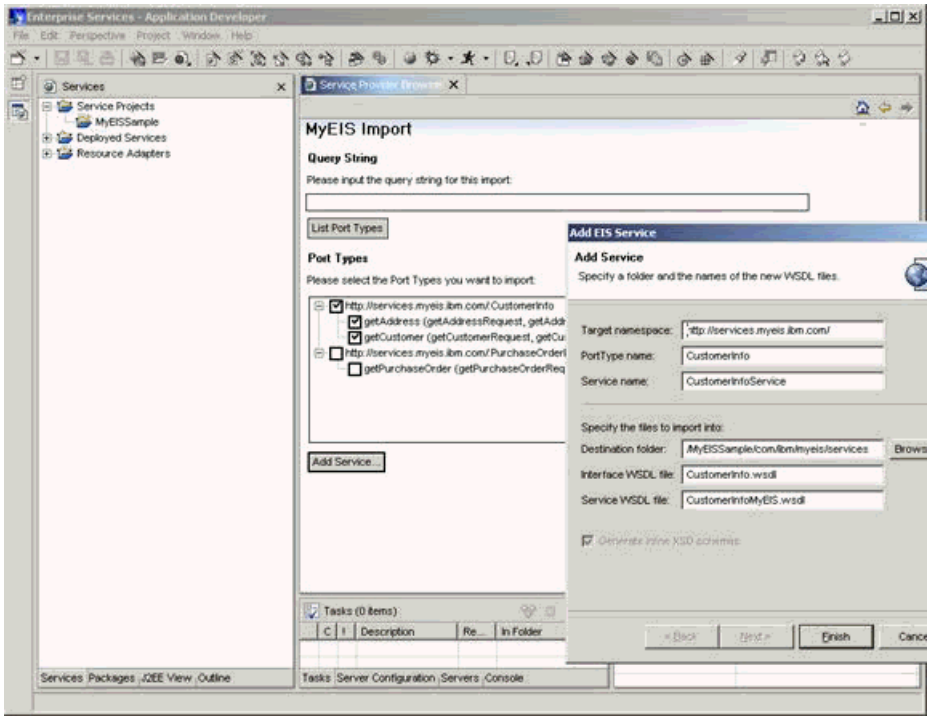
1. First create a service project and go to the service provider browser.
2. Select the MyEIS connector.



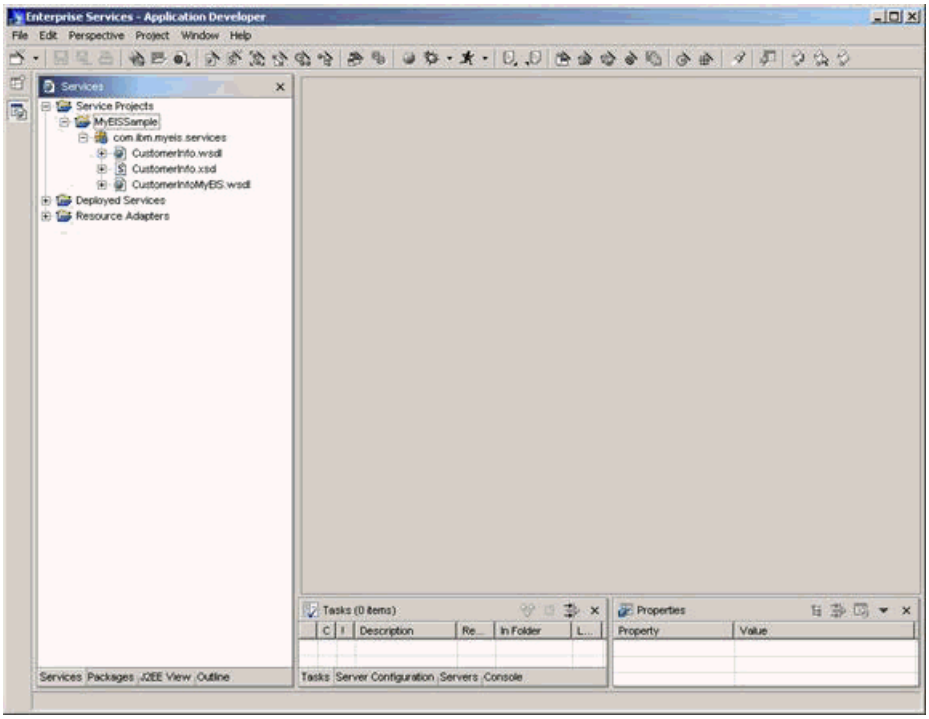
3. Input the connection properties for the EIS system you want to import from and select **Continue**.



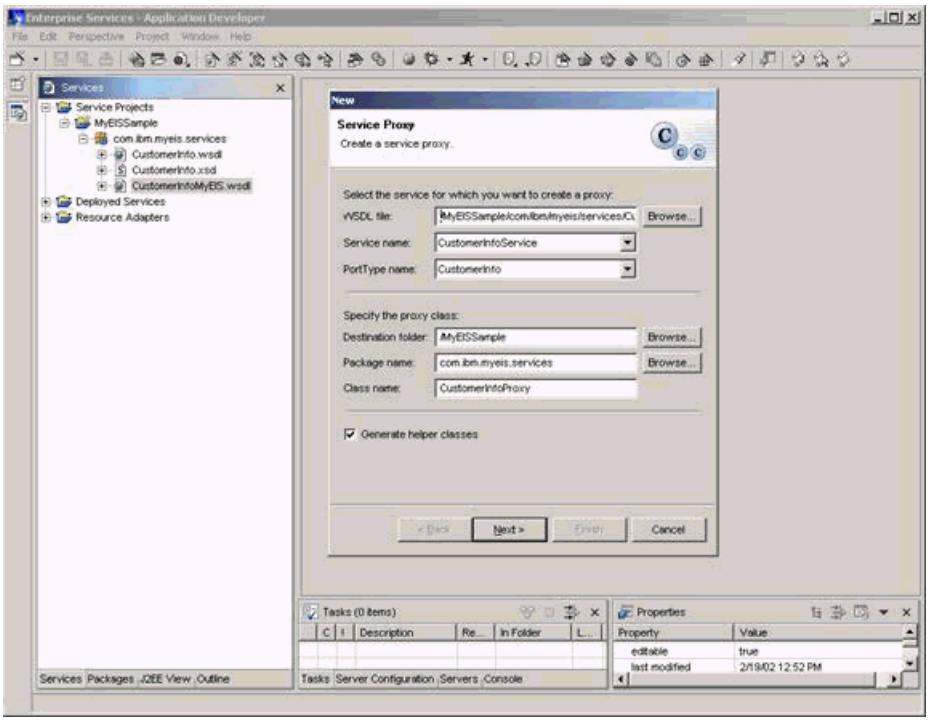
4. Click **List Port Types** to show the list of portTypes offered by MyEIS.
5. Select the CustomerInfo portType and then click **Add Service**.
6. In the wizard select the WSDL file properties you want to import to and then click **Finish**.



7. The imported WSDL document or documents should appear in your workspace along with the corresponding XSD files.

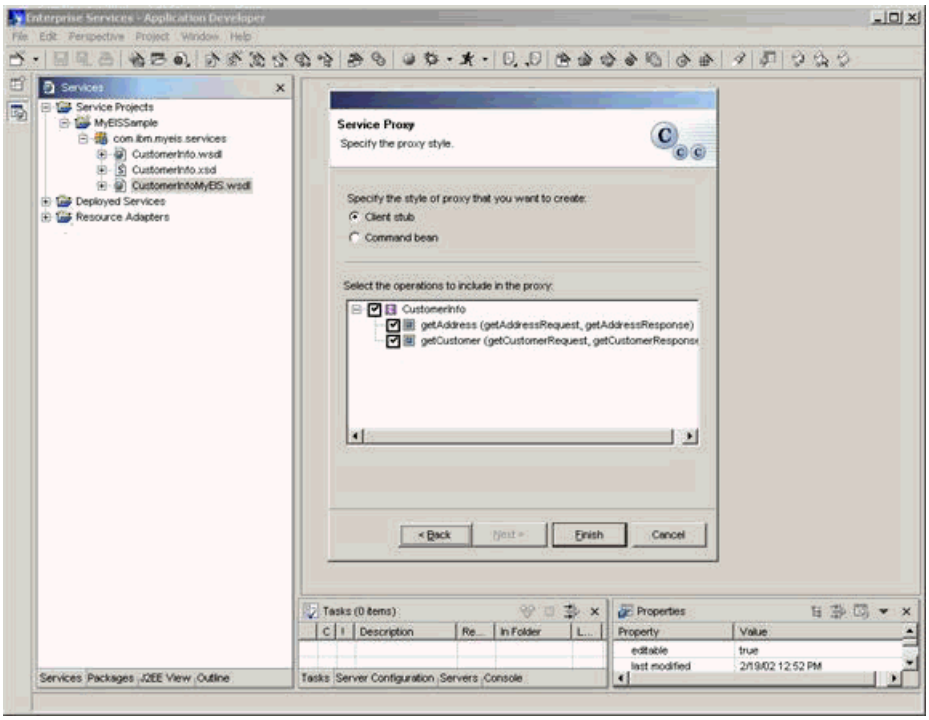


8. Select the CustomerInfoMyEIS.wsdl file, and choose the **Proxy** icon on the toolbar.
9. In the Proxy wizard, select the properties for the Java code that you want to generate (such as the package names, the Class name, and so on).

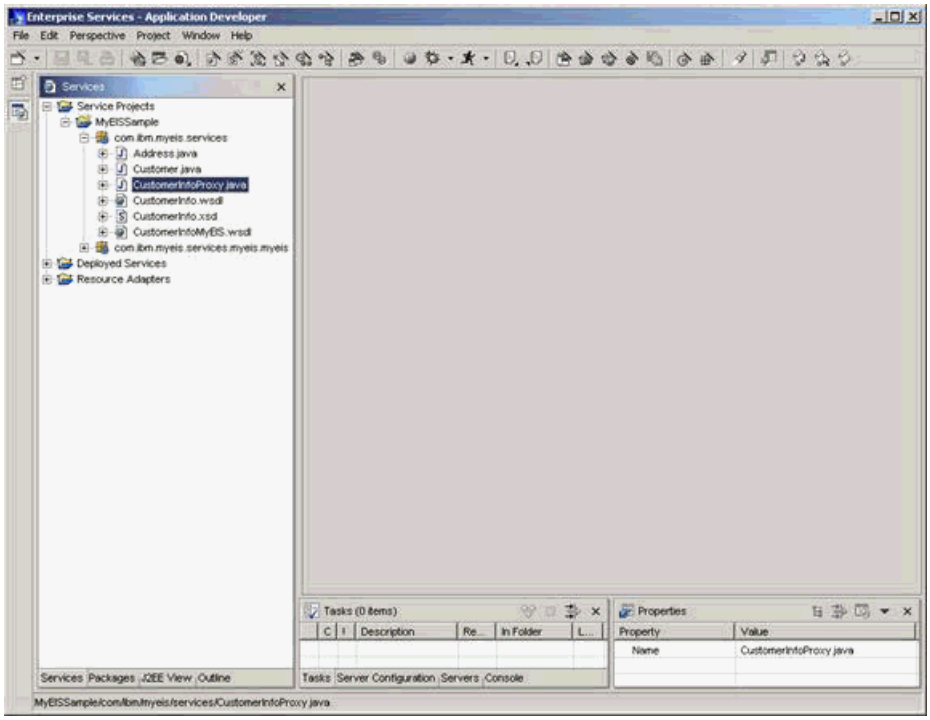


10. Select the generation style (Command Bean or Client Stub).

11. Select the operations that you want to include in the generated proxy.



12. Click **Finish** and the generated code, along with necessary helper classes, will be added to your workspace:



Notices

Note to U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this Documentation in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this Documentation. The furnishing of this Documentation does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this Documentation and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. 2000, 2002. All rights reserved.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AFS
- DB2
- DB2 Extenders
- DB2 Universal Database
- CICS
- IBM
- IMS
- OS/390
- OS/400
- VisualAge
- WebSphere
- WorkPad

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

ActiveX, Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

UNIX is a registered trademark of The Open Group

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.