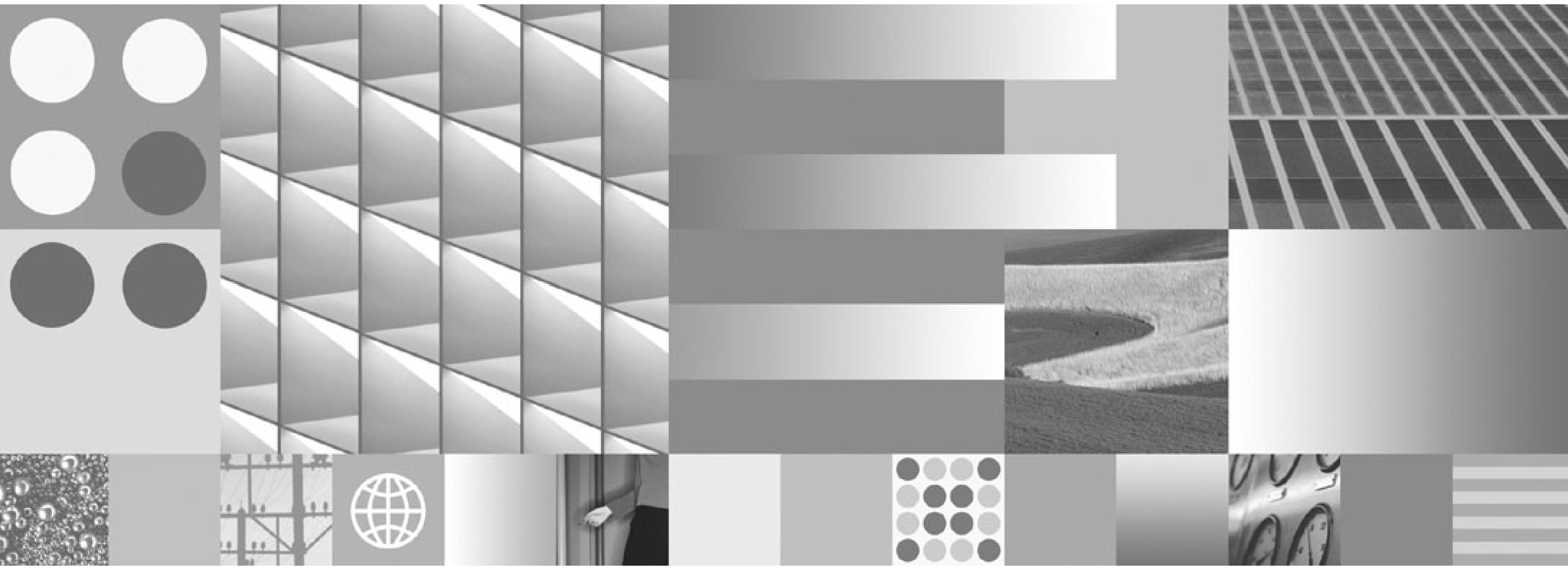
IBM Information Integration

Version 9.5

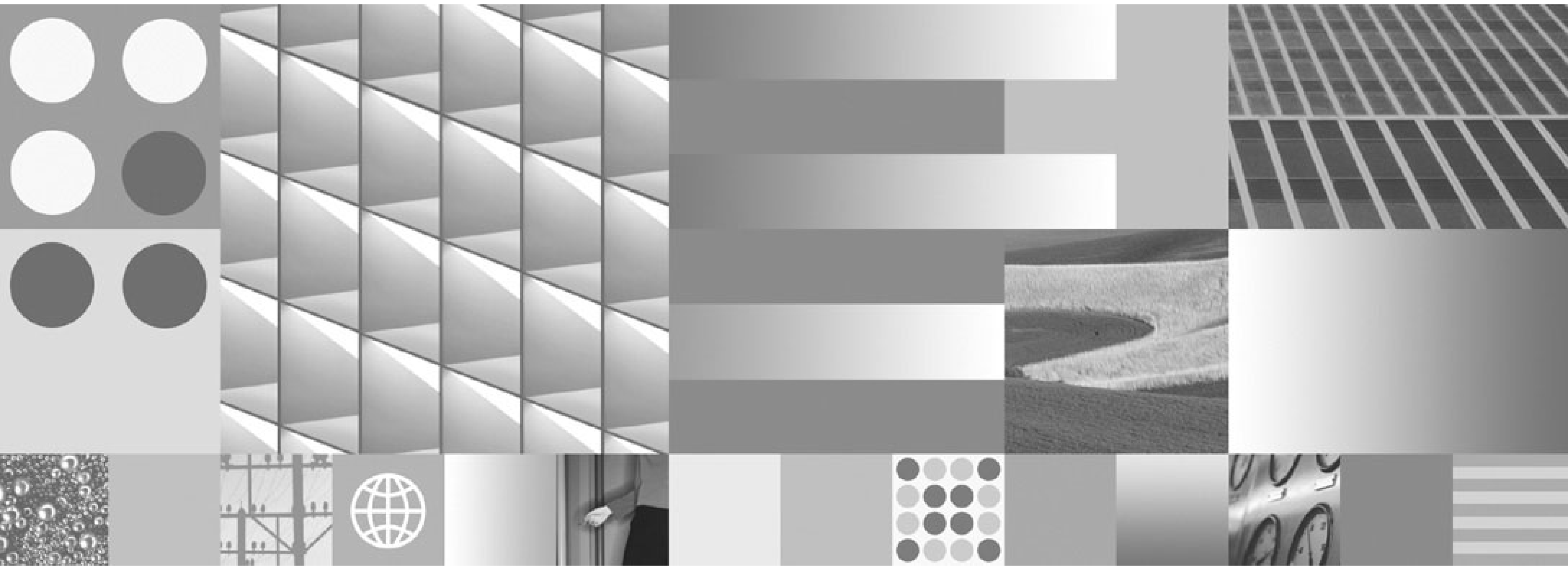


Application Development Guide for Federated Systems

SC19-1021-01

IBM Information Integration

Version 9.5



Application Development Guide for Federated Systems

SC19-1021-01

Note

Before using this information and the product that it supports, read the information in “Notices” on page 207.

Contents

Chapter 1. Overview of Web services application development

Web services allow you to access data from a variety of databases and internet locations. After you have accessed the data, a Web service consumer can search, mine, and then transform the data to use it with a data warehouse for further analysis.

You can use Web services to enable remote access to DB2® information. Web services include a set of application functions that perform some useful service on the behalf of a consumer, or a requester, such as informational or transactional functions. Web services perform functions, which can be anything from simple requests to complicated business processes. The consumer generally only needs to know the Web services description language interface to the Web service. In addition, the Web service can change usually without affecting the consumer, unless a change is made in the interface.

The Web service application programmer designs the interaction between a service provider, and a consumer, or service requester to be completely independent of platforms and languages. You can use just-in-time integration, because service requesters can find service providers dynamically. Web services reduce complexity through encapsulation. Service requesters and providers are concerned only with the interfaces necessary to interact with each other, not their underlying implementation. Web services give new life to legacy applications because you can cast an existing application as a Web service. The basic elements of Web services include simple object access protocol (SOAP), Universal Description, Discovery, and Integration (UDDI), and Web services description language (WSDL).

Web services and information integration

Web services provide key innovations for information integration.

Deprecating Web services object runtime framework (WORF)

The Web services objects runtime framework (WORF) is no longer supported and will not be updated.

In the Data Server Developer Workbench, you now can create Web services without writing document access definition extension (DADX) files. Use the Data Server Developer Workbench to create the SQL statements and stored procedures on which you can base the operations of your Web services. With the Data Server Developer Workbench you can now easily deploy a Web service

Read the Developing and deploying Web services for more detailed information about the feature within Data Server Developer Workbench.

To use your existing WORF applications, you must migrate your applications to Web services within the Data Server Developer Workbench. For the instructions on migrating to the Web services within the Data Server Developer Workbench, see Migrating Web applications that were developed for the Web Object Runtime Framework (WORF).

Extending the functions of stored procedures and SQL statements

By using the Web services object runtime framework (WORF) and Document Access Definition Extension (DADX), application servers can serve stored procedures as Web services to clients. The application servers can be WebSphere® Application Server or Apache Tomcat.

All SQL statements that a database executes, including stored procedures, can become SOAP clients and request Web services from SOAP servers. The Web service then presents the data either as an SQL value or as a table that you can combine with other SQL data.

Assisting the application developer

Web services promote interoperability. Web services design the interaction between a service provider and a service requester to be completely platform-independent and language-independent. The reality of interoperability assumes that the information technology industry uses a set of standards that provide guidance on the development and integration of Web services. The Web Services Interoperability Organization is an open industry effort that is chartered to promote and ensure Web services interoperability across platforms, applications and programming languages. The Web Services Interoperability Organization uses specifications that are developed by the World Wide Web Consortium (W3C) and the Universal Description, Discovery and Integration (UDDI) organization. Web services interoperability means that you can create your Web services on a variety of SOAP or Web Services platforms, including IBM® Web services SOAP provider, Apache Axis, or Microsoft® Visual Studio.Net.

Web services enable just-in-time integration. As service requesters use service brokers to find service providers, the discovery takes place dynamically.

Web services reduce complexity through encapsulation. Service requesters and providers concern themselves with the interfaces necessary to interact with each other. As a result, a service requester has no idea how a service provider implements its service, and a service provider has no idea how a service requester uses its service. Web services encapsulates those details inside the requesters and providers.

Web services technologies allow you to cast older applications as a Web service. This means that you can use the applications and packages that are already in place in your enterprise in interesting new ways. In addition, the infrastructure associated with the older applications (such as security, directory services, and transactions) can be *wrapped* as a set of services as well.

WebSphere Application Server

The WebSphere Application Server is infrastructure software for dynamic e-business. The WebSphere Studio application development environment provides the tools that you need to build, deploy, and integrate your e-business.

WebSphere Application Server is a J2EE-compliant application server that provides an environment for open distributed computing. WebSphere Application Server provides a middle ground between a client and the resource management systems (such as databases). It allows clients (such as applets or C++ clients) to interact with data resources (such as relational databases or WebSphere MQ) and with existing applications.

You can run Web services on WebSphere Application Server Advanced Edition. Web services object runtime framework (WORF) provides the run-time support for invoking document access definition extension (DADX) documents as Web services over Hypertext Transfer Protocol (HTTP) with IBM Web Service SOAP provider or Apache Axis 1.2 (or later). WebSphere Application Server 5 or higher, and other servlet engines support this. WebSphere lets you secure your SOAP Web services. See the WebSphere documentation on securing SOAP services for more information.

The IBM Web services SOAP provider supports the Web services security and Web services transactions that are part of the WebSphere Application Server environment. For additional information, see WebSphere and .Net Interoperability Using Web Services .

Web services components: provider and consumer

Web services provide a simple interface between the provider and consumer of application resources using a Web Service Description Language (WSDL).

Web services provider

A Web services client application can obtain access to a DB2® Version 9 database with a Web services description language (WSDL) interface. You can create a WSDL interface to DB2® Version 9 data by using the Web services Object Runtime Framework (WORF), also known as Document Access Definition Extension (DADX) files. After you define the operations to access DB2 data with the DADX file, then you deploy the DADX file and its runtime environment (IBM Web Service SOAP provider or Apache Axis version 1.2) to a supported Java™ Web application server environment (Apache Jakarta Tomcat or IBM® WebSphere® Application Server). After you have the DB2 Web service tested and deployed, any Web services client can start using the DB2 Web service.

Web services consumer - the user-defined functions

When DB2 Version 9 becomes the consumer, Web services can take advantage of the optimization that is built within the database. By using SQL statements, you can consume and integrate Web services data. By using SQL to access Web services data, you can reduce some application programming efforts because the data can be manipulated within the context of an SQL statement before that data is returned to the client application. You can convert an existing WSDL interface into a DB2 table or scalar function by using tools that are provided in WebSphere Studio version 5 and later. During the execution of an SQL statement, you establish a connection with the Web service provider, and then you receive a response document as a relational table or a scalar value.

Web services consumer - the Web services wrapper

Within the federated systems, a Web services wrapper is available to allow users to access Web services with SQL statements on nicknames and views that invoke Web services. You can create a Web services wrapper and nicknames that specify input to the Web service and access the output from the Web service with SELECT statements.

Figure 1 shows the participation by DB2 Version 9 in the Web services environment:

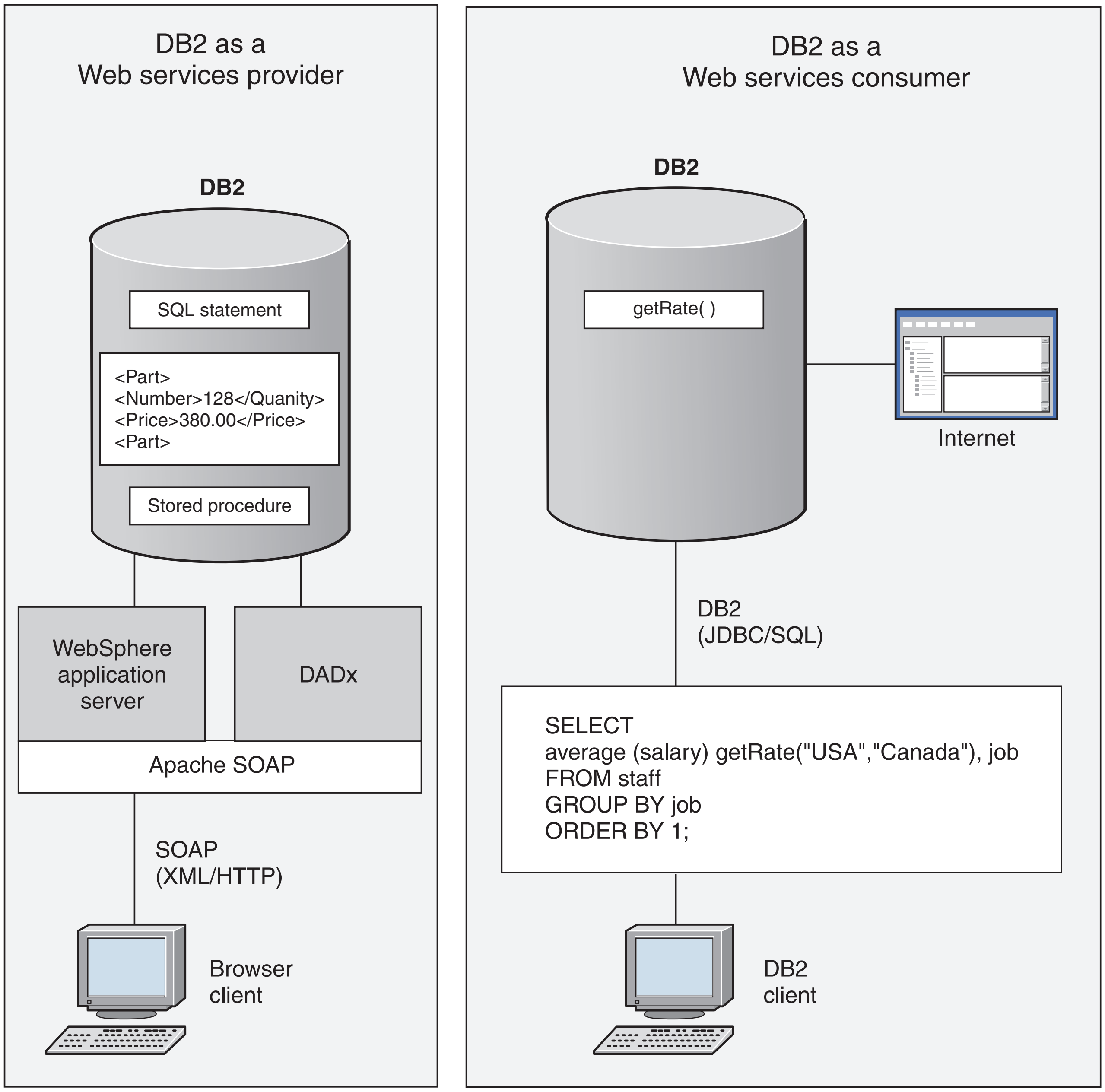


Figure 1. Web services provider and the SOAP user-defined functions

Web services fundamentals

The basic elements of Web services include simple object access protocol (SOAP), Universal Description, Discovery, and Integration (UDDI), and Web services description language (WSDL).

The Web service application programmer designs the interaction between a service provider, and a consumer, or service requester to be completely independent of platforms and languages. You can use just-in-time integration, because service requesters can find service providers dynamically. Web services reduce complexity through encapsulation. Service requesters and providers are concerned only with the interfaces necessary to interact with each other, not their underlying implementation. Web services give new life to legacy applications because you can cast an existing application as a Web service.

SOAP binding

The SOAP binding requests are XML documents that follow a certain schema.

The simple object access protocol (SOAP) binding uses Hypertext Transfer Protocol (HTTP) POST. The SOAP binding sends the operation name, input parameters, and other information as an XML request body.

The SOAP request binding issues a SOAP request over HTTP. SOAP is used as a message protocol, for request and response messages. The SOAP request specifies how the request and response message should appear.

SOAP operates on top of an HTTP POST request. HTTP is the transport protocol, and SOAP is the message protocol. A client application must know how to build SOAP request documents. The definition and the format of parameters, and other information is defined in separate Web services description language (WSDL) document.

Use the following uniform resource locator (URL) to access the SOAP binding (remember that the *your WebAppServer* identifier depends on your Web server configuration):

```
http://<your WebAppServer>/services/db2sample/HelloSample.dadx/SOAP
```

There is no operation name in the request. The information is now in the SOAP request document.

The following example which is for an RPC style, is a dynamic query service using a SOAP binding.

```
<SOAP-ENV:Envelope

  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <SOAP-ENV:Body>

<ns0:executeQuery


     xmlns:ns0="http://schemas.ibm.com/db2/dqs">
     <queryInputParameter>
           select * from employee
     </queryInputParameter>
     <extendedInputParameter>
          <properties/>
     </extendedInputParameter>
   </ns0:executeQuery>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

XML element in the example	Description
SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"	The SOAP envelope.
ns0:executeQuery	The operation name.

XML element in the example	Description
http://schemas.ibm.com/db2/dqs	The actual request document. In the WORF environment, this is an XML document that is now an actual Web service SOAP request.

`<queryInputParameter>`
    select * from employee
  `</queryInputParameter>`
  `<extendedInputParameter>`
    `<properties/>`
  `</extendedInputParameter>`
	The parameters.

The SOAP response is:

```
<?xml version='1.0' encoding='UTF-8'?><SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SOAP-ENV:Body>
   <ns1:executeQueryResponse
      xmlns:ns1="http://schemas.ibm.com/db2/dqs"
          SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
    <queryOutputParameter>
       ...
    </queryOutputParameter>
   </ns1:executeQueryResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The information that is contained within the `SOAP-ENV:Body` tag is the actual response document. In the WORF environment, the response document is an XML document. This is an actual Web service SOAP response.

Sending additional DB2 client information to DB2 with the SOAP request

By using the Web services provider, you can send additional information about the client to DB2 in the SOAP request header. If you are not using WebSphere Application Server data sources, use the IBM Universal JDBC Driver Provider. If you do use the WebSphere Application Server data sources, use the WebSphere Application Server 6.0 or later. On AIX® 64-bit platforms, you need WebSphere Application Server 6.0.2.3 or later.

Configure the following parameters in the group.properties file:

1. Enable the *JDBCAccounting=Header* to inform the Web services provider to expect the DB2 client information in the SOAP header.
2. Define the mapping between the SOAP headers and the DB2 client information with the following statements:

   ```
   ClientUserHeader=<QName>
   ClientWorkstationHeader=<QName>
   ClientApplicationInformationHeader=<QName>
   ClientAccountingInformationHeader=<QName>
   ```

   QName is a qualified name of an XML element in the header. The following examples show the use of QName:

   ClientUserHeader={http://ibm.com}DB2ClientUser
   ClientWorkstationHeader={http://ibm.com}DB2ClientWorkstation
   ClientApplicationInformationHeader=

```
        {http://ibm.com}DB2ClientApplicationInformation
    ClientAccountingInformationHeader=
        {http://ibm.com}DB2ClientAccountingInformation
```

The SOAP header entry in the SOAP request must look like the following example:

```
<soapenv:Header>

<ns1:DB2ClientUser
   soapenv:mustUnderstand="0"
   xsi:type="xsd:string"
   xmlns:ns1="http://ibm.com">Miguel</ns1:DB2ClientUser>

<ns2:DB2ClientWorkstation
   soapenv:mustUnderstand="0"
   xsi:type="xsd:string"
   xmlns:ns2="http://ibm.com">Miguel Workstation</ns2:DB2ClientWorkstation>

<ns3:DB2ClientApplicationInformation
   soapenv:mustUnderstand="0"
   xsi:type="xsd:string"
   xmlns:ns3="http://ibm.com">
         Miguel Axis Client
          </ns3:DB2ClientApplicationInformation>

<ns4:DB2ClientAccountingInformation
   soapenv:mustUnderstand="0"
   xsi:type="xsd:string"
   xmlns:ns4="http://ibm.com">Miguel</ns4:DB2ClientAccountingInformation>

</soapenv:Header>
```

Note: Consider using the SOAP binding for Java™ and JavaScript™ clients. WebSphere® Studio has the functionality to generate Java™ Web service clients.

Web services description language

Web service providers are described by Web services description language (WSDL) documents. The key to the Web service is the Web services description language document.

The WSDL is an XML document that describes Web services as a collection of endpoints, or ports. An endpoint is an addressable location at which a Web service can be accessed according to the associated binding of a specified interface. One Web service can have multiple endpoints. The endpoints in a WSDL operate on messages. A WSDL binding describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol. A WSDL SOAP binding can be either a document-oriented or procedure-oriented (RPC) style binding. A SOAP binding can also have an encoded use or a literal use.

As Figure 2 on page 8 shows, the Web service provider implements a service and publishes the interface to some service broker, such as UDDI. The service requester can then use the service broker to find a Web service. When the requester finds a service, the requester binds to the service provider so that the requester can use the Web service. The requester invokes the service by exchanging SOAP (Simple object access protocol) messages between the requester and provider.

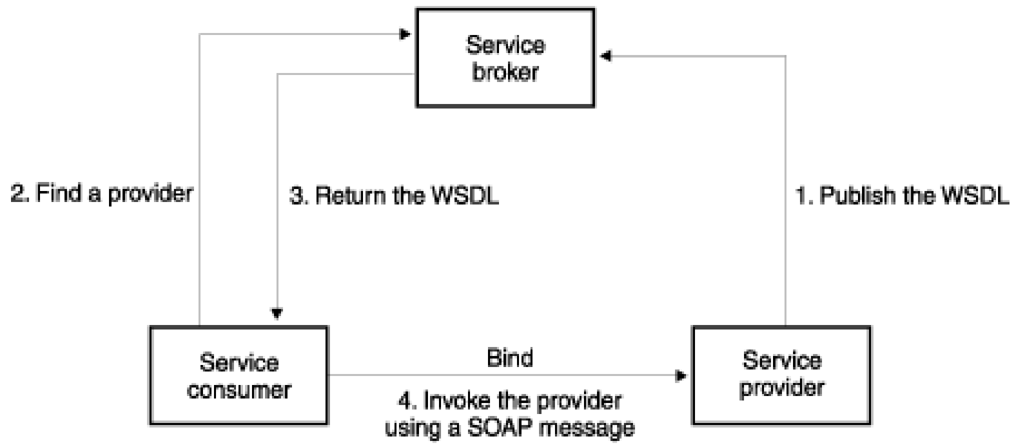


Figure 2. Web services as a service oriented architecture

The SOAP specification defines the layout of an XML-based message. A SOAP message is contained in a SOAP envelope. The envelope consists of an optional SOAP header and a mandatory SOAP body. The SOAP header can contain information about the actual message, such as encryption information or authentication information. The SOAP body contains the actual message. The SOAP specification also contains a default encoding for programming language bindings, which is called the SOAP encoding.

A WSDL document can contain one or more Web services. A service consists of one or more ports with a binding. The WSDL document can have one or more port types. A port type has one or more operations with abstract input and output messages. A binding refers to the process of associating protocol or data format information with an abstract entity like a message, operation, or a portType. A binding creates a concrete protocol and data format specification for a particular port type. A port is an endpoint that is a binding and a Web address.

The example in Figure 3 on page 10 shows the WSDL definition of a simple service providing stock quotes. The Web service supports a single operation that is named `GetLastTradePrice`. The service is deployed using the SOAP 1.1 protocol over HTTP. The request reads a ticker symbol as input, which is a string data type, and returns the price, which is a float data type. The type shown in this example is an XML schema definition. You can use XSD files to associate tables and columns in a DB2 table to your Web service.

The WSDL style that is specified in the <soap:binding> element is the document style. The <soap:operation> element provides information for the operation as a whole. The style attribute in the <soap:operation> element indicates whether the operation is RPC-oriented (messages containing parameters and return values) or document-oriented (messages containing documents). The value of this attribute also affects the way in which the body of the SOAP message is constructed. If the attribute is not specified, it defaults to the value specified in the <soap:binding> element. The Web services provider contains samples that are set to use RPC style. For new applications, you should use document style for maximum interoperability. In Version 8.2, the Web services provider uses an RPC style with literal usage and type nodes instead of an RPC style with literal usage and element nodes. However, there is no change to the SOAP messages from earlier releases of Web services provider.

The complete example and the WSDL specification is at the W3C site (http://www.w3.org/TR/2001/NOTE-wsdl-20010315).

```
<?xml version='1.0'?>
<definitions name='StockQuote'
...


<types>
       <schema targetNamespace='http://example.com/stockquote.xsd'
              xmlns='http://www.w3.org/2000/10/XMLSchema'>
           <element name='TradePriceRequest'>
              <complexType>
                  <all>
                      <element name='tickerSymbol' type='string'/>
                  </all>
              </complexType>
           </element>
           <element name='TradePrice'>
              <complexType>
                  <all>
                      <element name='price' type='float'/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>

<message name='GetLastTradePriceInput'>
...
</message>

    <portType name='StockQuotePortType'>
        <operation name='GetLastTradePrice'>
           <input message='tns:GetLastTradePriceInput'/>
           <output message='tns:GetLastTradePriceOutput'/>
        </operation>
    </portType>


    <binding
       name='StockQuoteSoapBinding'
       type='tns:StockQuotePortType'>
        <soap:binding
          style='document'
            transport='http://schemas.xmlsoap.org/soap/http'/>
        <operation name='GetLastTradePrice'>
           <soap:operation
             soapAction='http://example.com/GetLastTradePrice'/>
           <input>
               <soap:body use='literal'/>
           </input>
           <output>
               <soap:body use='literal'/>
           </output>
        </operation>
    </binding>

   <service name='StockQuoteService'>
        <documentation>My first service</documentation>
        <port name='StockQuotePort'
           binding='tns:StockQuoteBinding'>
           <soap:address
             location='http://example.com/stockquote'/>
        </port>
    </service>

</definitions>
```

Figure 3. Example of a WSDL

Since WSDL documents have a certain structure, Web services developers might need to use types from external schemas either at the definitions level of the WSDL document, or the types level of the WSDL document. To use external schemas, you can use imported schema definitions in your WSDL. WORF supports two types of imports during the WSDL generation:

An import at the /definitions scope of a WSDL
: http://schemas.xmlsoap.org/wsdl/:import

An import at the /definitions/type/schema scope of a WSDL
: http://www.w3.org/2001/XMLSchema:import

You can add import definitions by using a group.imports file. If a group.imports file exists in the resources of the Web service group directory, then WORF includes the group.imports information in the generated WSDL. The following example is a group.imports file:

```
<?xml version='1.0' encoding='UTF-8'?>
<imports xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
         xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
   <wsdl:import namespace='http://some/namespace/1'
           location='schema1.xsd'/>
   <wsdl:import namespace='http://some/namespace/2'
           location='schema2.xsd'/>
   <xsd:import namespace='http://some/namespace/3'
           schemaLocation='schema3.xsd'/>
   <xsd:import namespace='http://some/namespace/4'
           schemaLocation='schema4.xsd'/>
</imports>
```

This example defines two imports in the /definitions scope that will be added to the WSDL (schema1.xsd and schema2.xsd). The example defines two imports in the /definitions/types/schema scope that will be added to the WSDL (schema3.xsd and schema4.xsd). The WSDL that WORF generates that includes the import definitions from the above file has the following structure:

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions>
  <wsdl:documentation xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
       xmlns='http://schemas.xmlsoap.org/wsdl/'>
     Documentation Text Node
  </wsdl:documentation>
  <import location='schema2.xsd'
            namespace='http://some/namespace/2'/>
  <import location='schema1.xsd'
            namespace='http://some/namespace/1'/>
  <types>
    <schema>
      <import namespace='http://some/namespace/4'
             schemaLocation='schema4.xsd'/>
      <import namespace='http://some/namespace/3'
             schemaLocation='schema3.xsd'/>
      <element name='executeQueryResponse'> .... </element>
      <element name='executeQuery'> .... </element>
    </schema>
  </types>
   ...
</definitions>
```

If you installed the WORF examples, and have an application called `services`, then you can request a Web services description language (WSDL) document for the service, HelloSample.dadx. Use the following uniform resource locator (URL) to request the WSDL. The localhost port number, designated here by `<yourWebAppServer>` depends on your own current machine:

```
http://<yourWebAppServer>/services/db2sample/HelloSample.dadx/WSDL
```

WORF automatically generates the WSDL document from DADX.

UDDI business registries

Register your Web service in a Universal Discovery, Description, and Integration (UDDI) business registry.

The recommended practice is to split the WSDL document into a service instance document and a binding document. To learn more about UDDI and best practices, see UDDI Best Practices.

The service instance document contains the address from which you deploy the service and it imports the binding document. Many service instances might refer to a common binding document. You register the binding document in UDDI as a reusable `tModel`. The `tModel` is the information about a specification for a Web service.

Request the WSDL service instance document with the uniform resource locator (URL). The localhost port number, designated here by `<yourWebAppServer>` depends on your own current machine:

```
http://<yourWebAppServer>/services/db2sample/HelloSample.dadx/WSDLservice
```

Request the WSDL binding document with the URL:

```
http://<yourWebAppServer>/services/db2sample/HelloSample.dadx/WSDLbinding
```

WSDL for UDDI registration

The Universal Description, Discovery, and Integration (UDDI) best practices document explains the use of Web services description language (WSDL) with UDDI registries. You should split the WSDL document into two parts; the deployment, and the reusable parts.

The deployment part includes the <service> element which contains the URLs where the service is deployed. The deployment part imports the reusable part which contains the other top-level WSDL elements.

The reusable part corresponds to a UDDI <tModel> element. The deployment part corresponds to a UDDI <businessService>. Within the <businessService> element, each WSDL <port> element corresponds to a UDDI <bindingTemplate> element.

To learn more about UDDI and Web service registration, see the Universal Description, Discovery, and Integration of Business for the Web site.

The deployment part

To generate the deployment part of the WSDL document, submit a URL with the `WSDLservice` command. The syntax is:

```
http://yourWebAppServer:port/webapp_name/
   group_name/DADX_file.dadx/WSDLservice
```

Here is an example:

```
http://yourWebAppServer/sales_db/part_orders.dadx/WSDLservice
```

The reusable part

To generate the reusable part of the WSDL document, submit a URL with the `WSDLbinding` command. The syntax is:

```
http://yourWebbAppServer:port/webapp_name/
    group_name/DADX_file.dadx/WSDLbinding
```

Here is an example:

```
http://yourWebAppServer/sales_db/part_orders.dadx/WSDLbinding
```

The above example deals with the case in which the service implementer creates a Web service that is unique to a company. One of the usage scenarios that UDDI is designed to handle is one where a standards body or vendor defines a Web service interface tModel. Then, service implementers use the Web service. For example, the airline industry might define a Web service that provides flight schedules that airlines can implement. UDDI allows users to search for all registered services that implement a given tModel. Then, a travel planning application can locate all the airline flight schedule services.

The <implements> tag

Use the DADX <implements> element to declare that the service implements a Web Service described by a reusable WSDL document that is defined elsewhere. An example of an <implements> element is shown in Figure 4 on page 14.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://schemas.ibm.com/db2/dxx/dadx"
  ...
  elementFormDefault="qualified">
  <import namespace="http://schemas.xmlsoap.org/wsdl/"
   schemaLocation="wsdl.xsd"/>
        ....
   <element name="DADX">
    <annotation>
      <documentation>
        Defines a Web Service.
        The Web Service is described by an optional
         WSDL documentation element.
        ....
      </documentation>
    </annotation>
    <complexType>
      <sequence>
        <element ref="wsdl:documentation" minOccurs="0"/>
        <element ref="dadx:implements" minOccurs="0"/>
        <element ref="dadx:result_set_metadata" minOccurs="0"
                maxOccurs="unbounded"/>
        <element ref="dadx:operation" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
 ...
  <element name="implements">
    <annotation>
      <documentation>
        Defines the namespace and location of a set of WSDL bindings
        defined elsewhere. This information is imported into the
        WSDL document generated for this Web Service.
      </documentation>
    </annotation>
    <complexType>
      <attribute name="namespace" type="anyURI" use="required"/>
      <attribute name="location" type="anyURI" use="required"/>
    </complexType>
  </element>
...
</schema>
```

Figure 4. Element <implements>

The following example shows how the <implements> tag is used in a DADX file:

```
<?xml version="1.0"?>

<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"

  xmlns:xsd="http://www.w3.org/2001/XMLSchema"

  xmlns:xhtml="http://www.w3.org/1999/xhtml">

<documentation>

  Provides queries for part order information at myco.com.

  This Web Service is compliant with the Part Ordering Industry
   Association standard.

  </documentation>


<implements namespace="http://www.poia.org/PartOrders.wsdl"

  location="http://www.poia.org/PartOrders.wsdl"/>


<operation name="findAll">

<documentation%gt;

Returns an order with its complete details.

</documentation>

    ...


  </operation>
...
```

Figure 5. Example DADX file using an implements tag

XML schema definitions

An XML schema defines the data types used in the Web service interface.

Request the XML schema definitions for the service by the uniform resource locator (URL). The localhost port number, designated here by `<yourWebAppServer>` depends on your own current machine:

`http://<yourWebAppServer>/services/db2sample/HelloSample.dadx/XSD`

WORF generates an XML schema file similar to the example in Figure 6 on page 16.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://localhost:8080/services/sample/HelloSample.dadx/XSD"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://localhost:8080/services/sample/HelloSample.dadx/XSD">
  <element name="listDepartmentsResult">
    <complexType>
      <sequence>
        <element maxOccurs="unbounded" minOccurs="0" name="listDepartmentsRow">
          <complexType>
            <sequence>
              <element name="DEPTNO" type="string"/>
              <element name="DEPTNAME" type="string"/>
              <element name="MGRNO" nillable="true" type="string"/>
              <element name="ADMRDEPT" type="string"/>
              <element name="LOCATION" nillable="true" type="string"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 6. The XML schema definition file

The DB2® XML Extender can use document type definitions (DTDs) to define the schema of XML documents, so the WORF run-time automatically translates the DTD into an XML Schema. For example, if the order.dtd DTD defines an XML document, then you can use the following URL to request the translation into XML Schema:

```
http://<yourWebAppServer>/services/db2sample/order.dtd/XSD
```

Preparing the Web services environment on the Web Application Server

You can deploy Web applications by using the WebSphere Application Server. You must ensure that the required software is available and the configurations are complete for the specific computer environment that you are using.

A developer creates the files comprising a Web application, and then assembles the Web application components into a Web module. Then, a developer in a unit-testing environment or an administrator in a production environment, installs the Web application on the server.

Preparing the Web services environment in UNIX and Windows

Prepare the environment to use Web services with DB2 and federated server in UNIX® and Windows®.

Before you begin

Ensure that you have the required software installed.

Procedure

To prepare the Web services environment in UNIX and Windows:

1. Stop any services that use DB2 (such as WebSphere Application Server).
2. Stop DB2.
3. Optional: For advanced mapping control between XML and relational data, install the DB2 XML Extender.
4. For DB2 Universal Database versions earlier than Version 8, select Java Database Connectivity (JDBC) 2.0. Run the C:\SQLLIB\java12\usejdbc2.bat file, assuming that you installed DB2 in C:\SQLLIB\ when using a Windows environment.
5. Restart DB2.
6. Verify the DB2 installation by creating the DB2 SAMPLE database, if it is not already created.
7. Start WebSphere Application Server Advanced Edition 5.1 or later from its install directory. These instructions assume that you installed WebSphere Application Server in a Windows environment in C:\WebSphere\Appserver.

Preparing the Web services environment in z/OS or OS/390

Prepare the environment to use Web services with DB2 and the federated server in z/OS® or OS/390®.

Before you begin

Ensure that you have the required software installed.

Procedure

To prepare the Web services environment in z/OS or OS/390:

1. Create a new directory to store application extensions, if one does not already exist.
2. Set the APP_EXT_DIR environment variable in your designated J2EE server instance to this application extensions directory
3. Add the following JAR files to the application extensions directory:
   - xerces.jar: This file is in the IBM XML Toolkit for z/OS which you can download from http://www.ibm.com/servers/eserver/zseries/software/xml/
   - mail.jar: This file is in JavaMail.
   - activation.jar: This file is located in the Java Beans Activation Framework.
   - j2ee.jar: Download this file from http://java.sun.com/products
   - qname.jar: Download this file from http://java.sun.com/products
   - wsdl4j.jar: Download this file from http://oss.software.ibm.com/developerworks/projects/wsdl4j
4. Verify the configuration of the J2EE server instance with the following steps:
   a. Ensure that the soap.jar included with WebSphere Application Server is part of your CLASSPATH.
   b. Add the following settings to the jvm.properties file of the J2EE server instance:
   ```
   com.ibm.ws390.server.classloadermode=2
   com.ibm.ws.classloader.ejbDelegationMode=false
   ```
5. Restart the J2EE server.

Preparing the Web services environment in iSeries

Prepare the environment to use Web services with DB2 and a federated server in iSeries™.

Before you begin

Ensure that you have the required software installed.

Procedure

To prepare the Web services environment in iSeries:

1. Create the SAMPLE database from interactive SQL with the following command:
   ```
   CALL QSYS/CREATE_SQL_SAMPLE('SAMPLE')
   ```
2. For advanced mapping control between XML and relational data, install the DB2 XML Extender. You can verify that you have DB2 XML Extenders on your system by issuing the CL command, GO LICPGM. For DB2 for iSeries, V5R2, if you have DB2 XML Extenders, the following entries display as a result of the GO LICPGM command:
   - 5722DE1 *COMPATIBLE DB2 Extenders
   - 5722DE1 *COMPATIBLE DB2 Text Extender
   - 5722DE1 *COMPATIBLE DB2 XML Extender
   - 5722DE1 *COMPATIBLE Text Search Engine
3. Enable DB2 XML extenders with the following CL command:
   ```
   CALL PGM(QDBXM/QZXMADM) PARM(enable_db LOCALRDB)
   (LOCALRDB is the *LOCAL database name in the relational database directory.)
   ```
4. To work with the relational database entries, issue the following CL command:
   ```
   WRKRDBDIRE
   ```
5. Optional: If you use the document type definitions (DTDs) that are in the sample files, run the setup-dxx.cmd script.
6. Stop any services that use DB2 (such as WebSphere Application Server).
7. For DB2 Universal Database™ versions earlier than Version 8, select Java Database Connectivity (JDBC) 2.0. Run the C:\SQLLIB\java12\usejdbc2.bat file, assuming that you installed DB2 in C:\SQLLIB when using a Windows environment.
8. When using stored procedures, authorize the *PGM object that is created for each CREATE PROCEDURE statement. When you use Java™ stored procedures, you should authorize the user (or *PUBLIC) to the Java class file. When you use Java stored procedures, store all of the class files in the following directory:
   ```
    /QIBM/UserData/OS400/SQLLib/Function
   ```

   Make sure that Spserver.class is in this directory. The stored procedure SAMPLE.TESTRS is the only stored procedure that is an SQL stored procedure. It has no dependency on the Java class.
9. To define the sample stored procedures and catalog them in DBD:
   a. From the command line, specify the following:
   ```
   >qsh
   ```
   b. From the command line, specify the following:

```
>cd  /QIBM/UserData/WebASAEs4/worf/
   installedApps/servicesApp.ear/services.war/WEB-INF/
    classes/groups/dxx_sample
```

where WebASAEs4 is the version of WebSphere®, and worf is the name of the WebSphere instance.

c. From the command line, specify the following:

```
>db2 -f Spcreate.db2
```

10. To remove the stored procedure definitions, issue the following command:

```
>db2 -f Spdrop.db2
```

11. Start WebSphere Application Server Advanced Edition 4.01 or 5.0, assuming that you installed WebSphere Application Server in C:\WebSphere\Appserver.

Application server for DB2

The DB2 Embedded Application Server is an application server packaged with the DB2 product. This component is included to provide a means to run the web applications that are supplied with the DB2 server product without the need to purchase a separate application server.

The DB2 Embedded Application Server enables you to run the Web applications supplied with DB2 without needing to purchase an application server. In Version 8, the DB2 Embedded Application Server was also referred to as the application server for DB2 UDB.

Installing the application server for DB2 in a federated server

DB2, provides an embedded application server, referred to as the application server for DB2. If you use the application server for DB2, you do not need to install a separate application server to run your DB2 Web applications on Windows, Linux®, AIX, and Solaris.

Before you begin

- DB2 Version 9 or later
- At least one DB2 instance must exist
- Issue the following command for your environment:

```
<db2instance path>/sqllib/db2profile (for Windows)
. <db2instance path>/sqllib/db2profile (for UNIX systems)
```

Restrictions

You can have only one DB2 application server in a system that has one or multiple DB2 instances.

About this task

Application servers enable enterprises to develop, deploy, and integrate next-generation e-business applications. You can use application servers as tools to administer your Web applications.

Procedure

To install the application server for DB2:

1. Insert the *Java application development and Web administration tools supplement for DB2* CD. DB2 provides this CD with the DB2 installation package.

2. Type the following command from the command line:

   ```
   db2appserverinstall
        -asroot path
        -hostname name
   ```

   -asroot
   : The absolute path for the application server installation.

   -hostname
   : The name of the host system.

Starting and stopping the application server for DB2

You can start and stop the application server for DB2 from the bin subdirectory of the application server for DB2 directory. You can also use a stored procedure named DB2EAS.SERVER to start and stop the application server.

Procedure

To start and stop the application server for DB2:

1. To start the application server for DB2, type the following command from the command line:

   ```
   startServer serverName
   ```

   The command requires the following parameter:

   serverName
   : The name of the application server that you want to start. The server name must be `server1`.

2. To stop the application server for DB2, type the following command from the command line:

   ```
   stopServer serverName
   ```

   The command requires the following parameter:

   serverName
   : The name of the application server that you want to stop. The server name must be `server1`

You must use port number 20000 for the Web services that are running under the application server for DB2. Invoke the WORF samples with the following URL:

```
http://localhost:20000/services
```

In this example, *services* is the context root that you specify when you install the Web services.

Refer to *WebSphere Application Server System Administration* for information on deploying and managing applications so that you can deploy the WORF samples with the application server for DB2. After you deploy the WORF samples with the application server for DB2, you can access the WORF test page from your browser.

Installing Web services provider samples on the application server for DB2

You can install and configure the Web services provider, the Web services provider applications, including the Web services provider samples, and JDBC providers and enable and disable a trace on an application server for DB2 by using the worf_eas_admin.jacl script.

Before you begin

The application server for DB2 must be running.

About this task

The worf_eas_admin.jacl script is included in the bin directory of the dxxworf.zip file. The examples assume that you are familiar with deploying the Web services provider examples.

Procedure

To install the Web services provider samples on the application server for DB2:

1. Issue the following command to install a JDBC provider that is used by the Web services provider samples Web application. The command syntax uses the Windows notation:

   ```
   $appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl configureJDBC
   -name db2jdbc
   -classpath "C:\\SQLLIB\\java\\db2java.zip"
   -iClassName COM.ibm.db2.jdbc.app.DB2Driver
   ```

   From the command line in Windows, use either two backslashes (\\) or one forward slash (/) as the directory delimiter.
2. Issue the following command to install the Web services provider samples. The command syntax uses the Windows notation:

   ```
   $appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl installApp
   -warPath "C:\\worf\\lib\\axis-services.war"
   -contextRoot services
   -appName WorfAxis
   ```

Installing Web applications on the application server for DB2

You can install and configure the Web services provider, the Web services provider applications, including the Web services provider samples, and JDBC providers and enable and disable a trace on an application server for DB2 by using the worf_eas_admin.jacl script.

Before you begin

The application server for DB2 must be running.

About this task

The worf_eas_admin.jacl script is included in the bin directory of the dxxworf.zip file. The examples assume that you are familiar with deploying the Web services provider examples.

Procedure

To install the Web applications on the application server for DB2:

Issue the following command to install and configure Web Services provider Web applications:

```
worf_eas_admin.jacl keyword
```

Keyword	Parameter
installApp Installs a Web application that is based on the parameters	*warPath* The path to the Web archive (WAR) package. *contextRoot* The context root of the application in the application server for DB2. *appName* The name of the application in the application server for DB2. The name must not contain any blank characters.
configureJDBC Configures a JDBC provider. The parameters are required.	*name* The name of the JDBC provider in the application server for DB2. *classPath* The CLASSPATH to the Java archive (JAR) packages of the JDBC driver. *iClassName* The name of the implementation class.
uninstallApp Removes an installed application. You must provide the name of the application in the application server for DB2.	None
removeJDBC Removes a JDBC provider. You must provide the name of the JDBC provider in the application server for DB2.	None
enableTrace Enables the trace for the application server for DB2. The application server for DB2 must be running.	None
disableTrace Disables the trace for the application server for DB2. The application server for DB2 must be running.	None

Examples of worf_eas_admin.jacl

- The following example shows the installApp keyword:

  ```
  $appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl installApp
  -warPath "C:\\My Files\\WORF\\axis-services.war"
  -contextRoot services
  -appName WorfAxis
  ```

  From the command line in Windows, use either two back slashes (\\) or one forward slash (/) as the directory delimiter.

- The following example shows the configureJDBC keyword:

```
$appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl configureJDBC
 -name db2jcc
 -classPath "C:\\SQLLIB\\java\\db2jcc.jar;
    C:\\SQLLIB\\java\\db2jcc_license_cu.jar;
    C:\\SQLLIB\\java\\db2jcc_license_cisuz.jar"
 -iClassName com.ibm.db2.jcc.DB2Driver
```

- The following example shows the uninstallApp keyword:

```
$appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl uninstallApp
  -appName WorfAxis
```

- The following example shows the removeJDBC:

```
$appserv_install_dir\bin\wsadmin.bat -f worf_eas_admin.jacl removeJDBC
  -name db2jcc
```

- The following example shows the enableTrace:

```
$appserv_install_dir\bin\wsadmin -f worf_eas_admin.jacl enableTrace
```

- The following example shows the disableTrace:

```
$appserv_install_dir\bin\wsadmin -f worf_eas_admin.jacl disableTrace
```

Preparing to install the Web services provider

Determine the capacity of your system and plan for the software that needs to be installed for your application programming interfaces and Web service applications.

Packaging: dxxworf.zip

In addition to being packaged with IBM WebSphere Federation Server, the Web services provider is also part of DB2 Version 9. The Web services provider is in the following path in DB2 Version 9: <DB2 installed location>\sqllib\samples\webservices\dxxworf.zip.

Operating systems

You can install the Web services provider on any of the following operating systems:

- Windows NT®
- Windows 2000
- Linux
- AIX
- Solaris Operating Environment
- OS/390 Version 2.8 or later
- z/OS Version 1.1 or later

Database environments

You can use the following database environments:

- IBM DB2 Version 9 or later

  http://www.ibm.com/software/data/db2. DB2 includes DB2 XML Extender, which is needed for advanced mapping control between XML and relational data.
- DB2 Universal Database for OS/390 Version 8.
- DB2 Universal Database for z/OS.

- Informix® Dynamic Server (IDS) Version 9.3

Web server

- WebSphere Application Server Advanced Edition Version 6.http://www.ibm.com/software/webservers/appserv/
- Apache Jakarta Tomcat Version 3.3.1 through 4.0.3 or later.http://www.apache.org/
- Application server for DB2.

Installing WORF to work with WebSphere Application Server Version 5 or later for Windows and UNIX

You can install the Web services provider on Windows or UNIX to work with WebSphere Application Server Version 5 or later.

Before you begin

Install WebSphere Application Server on your work station in a path such as C:\WebSphere\Appserver (in a Windows environment).

Procedure

To install WORF Version 9:

1. Unzip dxxworf.zip to a directory, such as C:\worf so that the directory has the following contents:

   readme.html

   lib\websphere-services.war and lib\axis-services.war
   Sample Web applications that contain Web services which use WORF.

   lib\worf.jar
   WORF library. You install this on the class path of the servlet engine.

   lib\worf-servlets.jar

   schemas\
   XML schemas for the DADX and namespace tables (NST) XML files, including `wsdl.xsd`, `db2WebRowSet.xsd`, and `dadx.xsd`

   tools\ The tools directory contains the DAD and DADX checker tools.
2. Verify that the directory of the server you are using (such as WebSphere Application Server) contains the appropriate Web services engine jar files. If the files are not in the directory, copy the jar files from the directory that contains the Apache Axis or IBM Web Service SOAP provider files so that you can enable the appropriate Web services engine.
   a. If you are using the Apache Axis framework, copy axis.jar to c:\WebSphere\AppServer\lib. Then, copy the contents of axis/lib to c:\WebSphere\AppServer\lib to access the other Apache Axis JAR files.
3. Copy worf.jar to C:\WebSphere\AppServer\lib.
4. If your WebSphere server is a release earlier than WebSphere 5.0.2, you must download a file from http://java.sun.com/xml/downloads/saaj.html, named `saaj.jar`. Copy `saaj.jar` to C:\WebSphere\AppServer\lib.
5. Start the WebSphere Application Server.
6. Open the Administrator's console by selecting **Start** → **Programs** → **IBM WebSphere** → **Administrator's Console**.

7. Configure WebSphere to run with your DB2 environment:
   a. From the left navigation pane, click **Servers** → **Application Servers**.
   b. Click on the name of your server in the right content pane.
   c. Click **Process Definition** → **Java Virtual Machine**.
   d. On the Configuration page, specify the class path as the path to the Java database information. If you installed DB2 in directory sqllib\, and you use the group.properties file that comes with the WORF samples, the following example is a valid path:
      ```
      C:\SQLLIB\java\db2java.zip
      ```
   e. Click **Apply** or **OK**.
   f. Save the configuration.
8. Stop the WebSphere Application Server.

Installing WORF on z/OS or OS/390

You can install the Web services provider on z/OS.

Procedure

To install the Web services provider on z/OS:

1. Download and unpax dxxworf.pax to an empty directory, such as /u/USER/worf/. Unpax the file by using the following command:
   ```
   pax -rvf dxxworf.pax
   ```
   After you expand the file, the directory has the following contents:
   - readme.txt
   - lib/websphere-services.war and lib/axis-services.war - sample Web applications that contain Web services which use WORF. The commands and instructions that are included here refer to services.war or services.ear. Please use the correct SOAP files for the SOAP engine you choose to run. For example, an example might refer to a services.war file, but if you installed the IBM Web Service SOAP provider engine, then the file is websphere-services.war.
   - lib/worf.jar - WORF library.
   - lib/worf-servlets.jar
   - schemas/ - XML schemas for the DADX and NST XML files
   - tools/ – The tools directory contains the DAD and DADX checker tools.
2. Copy worf.jar to the application extensions directory of your J2EE server instance.
3. Start (or restart) the J2EE Server.

Installing the Web services provider software requirements for Apache Jakarta Tomcat on UNIX and Windows

This topic describes the steps for installing the Web services provider for Apache Jakarta Tomcat.

Before you begin

- Install the DB2 XML Extender for the advanced mapping control between XML and relational data. Verify the installation by creating the DB2 SAMPLE database. Refer to http://www.ibm.com/support/

docview.wss?uid=swg21192376 for information about DB2 environment variables that you must enable when using DB2 XML Extender.
- WORF requires Java Database Connectivity (JDBC) 2.0, which is the default in DB2 Version 9.
- Ensure that you have the required software installed. Verify your installation for your particular platform with the specific documentation.

Procedure

To install the software for the WORF environment:
1. Stop DB2.
2. Issue C:\SQLLIB\java12\usejdbc2.bat and select JDBC 2.0. This step assumes that you installed DB2 in C:\SQLLIB\ in a Windows environment. This step is required if you are not running a version of DB2 later than Version 8.
3. Restart DB2.
4. Install the following Internet software:
   - Apache Jakarta Tomcat Version 4.0.6 or later binary: http://jakarta.apache.org/site/binindex.html
   - Apache Axis 1.2: http://www.apache.org/
   - Apache Xerces 1.4.4: http://xml.apache.org/
   - Sun JavaMail 1.2: http://java.sun.com/products
   - Sun JavaBeans™ Activation Framework (JAF) 1.0 1:http://java.sun.com/products
   - Sun j2ee.jar, version 1.3 or later: http://java.sun.com/products
   - Sun qname.jar: http://java.sun.com/products
   - wsdl4j.jar: http://oss.software.ibm.com/developerworks/projects/wsdl4j
   - IBM Web Service SOAP provider: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/topic/com.ibm.

   Apache Jakarta Tomcat Version 4 standard comes with the appropriate Xerces parser. For earlier versions you must add the Xerces parser to your CLASSPATH to use it as the XML parser.

Installing WORF on Apache Jakarta Tomcat

You can install WORF on Apache Jakarta Tomcat.

Procedure

To install WORF Version 9 on Apache Jakarta Tomcat:
1. To run WORF with IBM Web Service SOAP provider, or with Apache Axis, add the following JAR files to the class path on your application server:
   - axis.jar for the Apache axis engine.
   - xerces.jar (or the jars of your Java XML parser)
   - mail.jar
   - activation.jar
   - worf.jar
   - j2ee.jar, version 1.3 or later
   - qname.jar
   - wsdl4j.jar. You can download this file from http://oss.software.ibm.com/developerworks/projects/wsdl4j.

- jaxrpc.jar
- log4j-1.2.8.jar
- commons-logging.jar
- commons-logging-api.jar
- commons-discovery.jar
- db2java.zip, ordb2jcc.jar , db2jcc_license_cisuz.jar, and db2jcc_license_cu.jar (or the JDBC implementation jar of your database server). The name of the driver class depends on the driver package that you use. You can modify the driver package that you use in the `group.properties` file.

2. Modify the files listed in Table 1. The modifications that you make depend on your Apache Jakarta Tomcat version and platform. Add a line for each of the jar files mentioned above. Replace <jarfile> with the actual location of the jar file. If you run Apache Jakarta Tomcat in an integrated development environment, make sure that all these jars are on the CLASSPATH that you use for starting Tomcat. The files that you need to modify are all in the directory in which you start Apache Jakarta Tomcat. You should start and stop the server with the startup.bat or shutdown.bat or startup.sh or shutdown.sh that is in the app_server/bin directory.
3. If your WebSphere server is a release earlier than WebSphere 5.0.2, you must download a file from http://java.sun.com/xml/downloads/saaj.html, named `saaj.jar`. Copy `saaj.jar` to C:\WebSphere\AppServer\lib.

Table 1. Class path designations

Platform	Server software	File to modify	Command to add
UNIX	Apache Jakarta Tomcat 3.2.x	\bin\tomcat.sh (before ″export CLASSPATH″)	CLASSPATH = $CLASSPATH: <jarfile>
	Apache Jakarta Tomcat 3.3.x	\bin\tomcat.sh (before ″export CLASSPATH″)	CLASSPATH = $CLASSPATH: <jarfile>
	Apache Jakarta Tomcat 4.x	\bin\setclasspath.sh (before ″export CLASSPATH″)	CLASSPATH = $CLASSPATH: <jarfile>
Windows	Apache Jakarta Tomcat 3.2.x	\bin\tomcat.bat (:setClasspath section)	set CP = %CP%; <jarfile>
	Apache Jakarta Tomcat 3.3.x	\bin\tomcat.bat	set CLASSPATH = %CLASSPATH%; <jarfile>
	Apache Jakarta Tomcat 4.x	\bin\setclasspath.bat	set CLASSPATH = %CLASSPATH%; <jarfile>

Installing the Web services provider software requirements for Apache Jakarta Tomcat on iSeries

You can install the Web services provider for Apache Jakarta Tomcat on iSeries after ensuring that the required software is available.

Procedure

To install the required software:

1. Install the following Internet software from Apache:
   - Apache Jakarta Tomcat Version 4.0.3 or later binary from http://jakarta.apache.org/site/binindex.html. (Apache Jakarta Tomcat Version 4 standard comes with the appropriate Xerces parser. For earlier versions you must add the Xerces parser to your CLASSPATH to use it as the XML parser.)
   - Apache Xerces 1.4.4 from http://xml.apache.org/
2. Install the following software from Sun:
   - JavaMail 1.2 from http://java.sun.com/products
   - JavaBeans Activation Framework (JAF) 1.0 1: http://java.sun.com/products
   - j2ee.jar, version 1.3 or later: http://java.sun.com/products
   - qname.jar: http://java.sun.com/products
   - wsdl4j.jar: http://oss.software.ibm.com/developerworks/projects/wsdl4j

Web services provider software requirements for OS/390 and z/OS

You can install the Web services provider (Web object runtime framework) on OS/390 and z/OS operating systems.

Operating systems

You can set up the Web services provider (Web object runtime framework) in any of the following operating systems:
- OS/390 Version 2.8 or later
- z/OS Version 1.1 or later

Database environments

You can use the following database environments:
- IBM DB2 Universal Database for OS/390 Version 7 or DB2 Universal Database for z/OS (http://www.ibm.com/software/data/db2/os390)
- IBM DB2 XML Extender for OS/390 Version 7 or later (http://www.ibm.com/software/data/db2/extenders/xmlext/index.html). Required for store and retrieve operations

Software to install

- WebSphere Application Server Version 4.01 Service Level W401505 or later
- JavaMail Version 1.2 (http://java.sun.com/)
- JavaBeans Activation Framework Version 1.0.1(http://java.sun.com/)
- j2ee.jar, version 1.3 or later (http://java.sun.com/)
- qname.jar (http://java.sun.com/)
- IBM XML Toolkit for z/OS and OS/390 Version 1.4 with program temporary fix (PTF) UW95866 (http://www.ibm.com/servers/eserver/zseries/software/xml/)
- wsdl4j.jar. You can download this file from http://oss.software.ibm.com/developerworks/projects/wsdl4j.

Install the Web services provider examples

The Web services provider includes examples to help you begin using Web applications.

The Web services provider includes examples in the dxxworf.zip file. These examples can be run in a variety of environments to help you understand how to create your own Web services applications.

Installing and deploying WORF examples on WebSphere Application Server Version 4.0.4 for z/OS or OS/390

You can deploy the examples that are provided with the Web services provider on z/OS.

About this task

These steps are for deploying Web applications for use in WORF on the z/OS or OS/390 platform. You can also verify that you have correctly installed and configured WORF and its prerequisites.

Procedure

To install and deploy the Web services provider examples on z/OS:

1. Configure the System Management Scripting application programming interface (API). For more information on configuring the scripting API on your OS/390 or z/OS system, see *IBM WebSphere Application Server V4 for z/OS and OS/390: Installation and Customization* and *IBM WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management Scripting API*.
2. Prepare an enterprise archive file (EAR). You use EAR files to deliver Java 2 platform enterprise edition (J2EE) applications. They consist of Web archive files (WAR) and Java archive files (JAR).
   a. In UNIX System Services (USS), copy the websphere-services.war or the axis-services.war to a temporary, writable directory and change your current directory to that location.
   b. Type the following command from the USS command line (all on one line):
   ```
   390fy -op "" -context_root "/services"
         -display_name "ServicesApp" services.war
   ```
   The command creates an initial EAR file with the name services.ear in the current directory. The EAR file has a Context Root of services and a Display Name of ServicesApp. The Context Root is the part of the Uniform Resource Locator (URL) that directs WebSphere Application Server to your application. The Display Name is a string that identifies your application in the Systems Management End User Interface (SM/EUI) and in USS. You can edit the Context Root and Display Name to any name you choose.
   c. Resolve the Java Naming and Directory Interface (JNDI) name mapping for services.ear. Do this by issuing the following command from the USS command line (all on one line):
   ```
   390fy -JNDIejbp "/<Sysplex>/<J2EE Server>"
            -op "_resolved" services.ear
   ```
   The terms used in the above example have the following definitions:

   <J2EE Server>
   The name of the J2EE server onto which you will deploy the application

   <Sysplex>
   The name of the Sysplex on which your J2EE server exists

The command creates a new file with the name services_resolved.ear in the current directory.

3. Deploy the application

   Restriction: When executing the commands for this step, you must be logged into USS with a user ID that is registered as a Systems Management Administrator for WebSphere.

   a. Copy the following sample files from the WebSphere Application Server sample directory (<WAS_Home>/samples/smapi/) to the temporary directory that contains the EAR file you just created.
      - inputcreateconversation.xml
      - inputprocessearfile.xml
      - inputcommitconversation.xml

   b. Set the environment variable DEFAULT_CLIENT_XML_PATH to the temporary directory that contains the EAR file.

   c. Edit the file inputcreateconversation.xml and specify a conversation name and optionally a description of the name. The conversation name and description can be any text that you want. However, the conversation name must remain the same when you process the input files in the next steps. Here is an example of the conversation name and description:
      ```
      <inputcreateconversation conversationname="WORFSamples"
                conversationdescription="WORF Sample Test" />
      ```

   d. Save the file inputcreateconversation.xml.

   e. Type the following command (all on one line) from the USS command line:
      ```
      CB390CFG -action createconversation
         -xmlinput inputcreateconversation.xml
         -output createconv.out
      ```

      This command creates a file createconv.out in the temporary directory of your system and contains the results of the operation. This file is not needed except to verify the success of the application deployment.

   f. Edit the file inputprocessearfile.xml. Specify the target J2EE server and the EAR file to deploy. The following is an example of specifying the J2EE server and the EAR file:
      ```
      <inputprocessearfile conversationname="WORFSamples"
                        j2eeservername="BBOASR2"
      earfilename="/tmp/worfsamp/services_resolved.ear"
                        processingmode="standard" />
      ```

   g. Save the file inputprocessearfile.xml.

   h. Type the following command (all on one line) from the USS command line:
      ```
      CB390CFG -action processearfile
          -xmlinput inputprocessearfile.xml
          -output processear.out
      ```

      This command creates a file processear.out in the temporary directory of your system and contains the results of the operation. This file is not needed except to verify the success of the application deployment.

   i. Edit the file inputcommitconversation.xml. Specify the conversation name that you used in the previous steps. For example:
      ```
      <inputcommitconversation conversationname="WORFSamples" />
      ```

   j. Save the file inputcommitconversation.xml.

   k. Type the following command (all on one line) from the USS command line:

```
CB390CFG -action commitconversation
  -xmlinput inputcommitconversation.xml
  -output commitconv.out
```

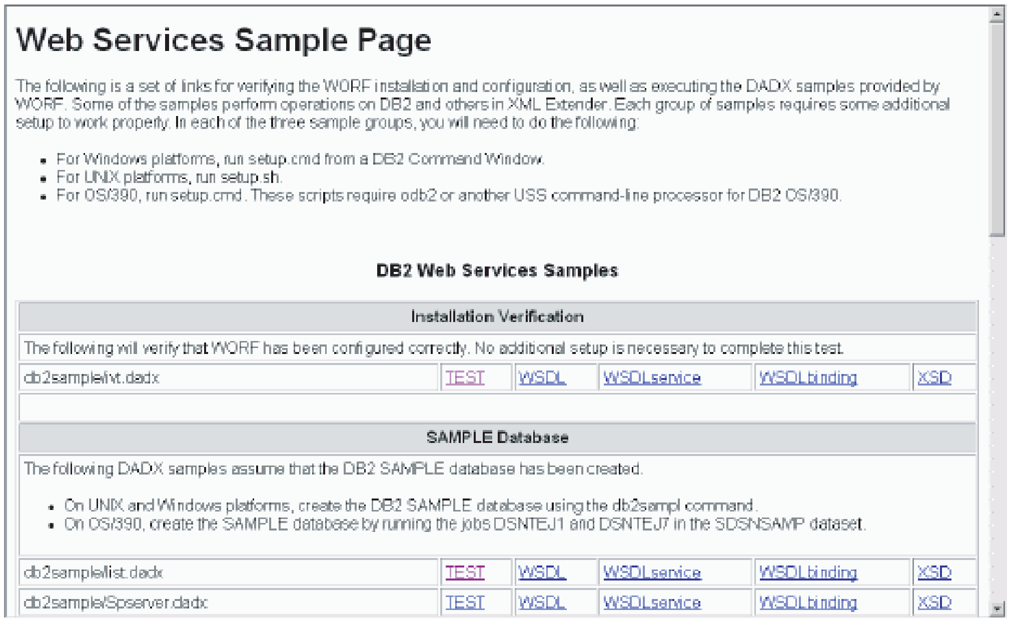
This command creates a file commitconv.out in the temporary directory of your system and contains the results of the operation. This file is not needed except to verify the success of the application deployment.

4. Set up the Web server.
   a. Ensure that the J2EE server allows the Context Root that you named in Step 2 on page 29. In the file webcontainer.conf of the server, ensure that at least one host has a specification as in the following example:
      ```
      host.<host_alias>.contextroots=/somewebapp,/services
      ```
      To accept any Context Root, you can use the following line of text:
      ```
      host.<host_alias>.contextroots=/*
      ```
      Using this method to specify the context root allows you to skip this step when you deploy future applications.
   b. If you use the Web server plug-in to access your J2EE server, add a Service statement to the file httpd.conf of your Web server. This statement specifies the Context Root of your deployed application. As an example, if you specified /services as your Context Root in Step 2 on page 29, your new Service statement is like the following example, which would be displayed on a single line:
      ```
      Service  /services/*
      <WAS_Home>/WebServerPlugIn/bin/was400plugin.so:service_exit
      ```
      The <WAS_Home> is the directory in which WebSphere Application Server is installed.
   c. Restart the Web server.
5. Verify the WORF configuration.
   a. Access the WORF Web Services Sample Page that is included in the sample WAR file. If you set the Context Root in Step 2 on page 29 to /services, type the following URL:
      ```
      http://<hostname>/services/
      ```
   b. In Figure 7 on page 32, the first section of samples, titled **Installation Verification**, shows a single DADX file, ivt.dadx. Click on the **TEST** link to open the built-in test facility of WORF.
   c. From the WORF test facility, Figure 8 on page 33, select the **testInstallation** operation. Click on **Invoke**.
   d. An XML document displays in the bottom frame of the window. Verify that the current time of day appears in the document, such as in the following example:
      ```
      <CURTIME>14:38:26.000Z</CURTIME>
      ```
      If this test fails, the configuration is not correct. Verify that you correctly installed the software requirements. Also, verify that you configured the WebSphere Application Server and that you have the authorization to access DB2.
6. Prepare and run the examples.
   a. On the Figure 7 on page 32 there are DADX samples that are shipped with the services.war application. Before you run the samples, follow the setup instructions that are contained within each category.

b. Execute an individual Web service by selecting the **TEST** link for each sample. Select the **WSDL**, **WSDLservice**, **WSDLbinding**, and **XSD** links to run those examples.

After you deploy the application, you can modify the application. You can also create additional WAR files for deployment. After you create a WAR file, deploy the Web application by using Step 2 on page 29, Step and Step 4 on page 31.

Here are examples of the Web services sample pages:



Web Services Sample Page

The following is a set of links for verifying the WORF installation and configuration, as well as executing the DADX samples provided by WORF. Some of the samples perform operations on DB2 and others in XML Extender. Each group of samples requires some additional setup to work properly. In each of the three sample groups, you will need to do the following:

- For Windows platforms, run setup.cmd from a DB2 Command Window.
- For UNIX platforms, run setup.sh.
- For OS/390, run setup.cmd. These scripts require odb2 or another USS command-line processor for DB2 OS/390.

DB2 Web Services Samples

Installation Verification

The following will verify that WORF has been configured correctly. No additional setup is necessary to complete this test.

db2sample/ivt.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD

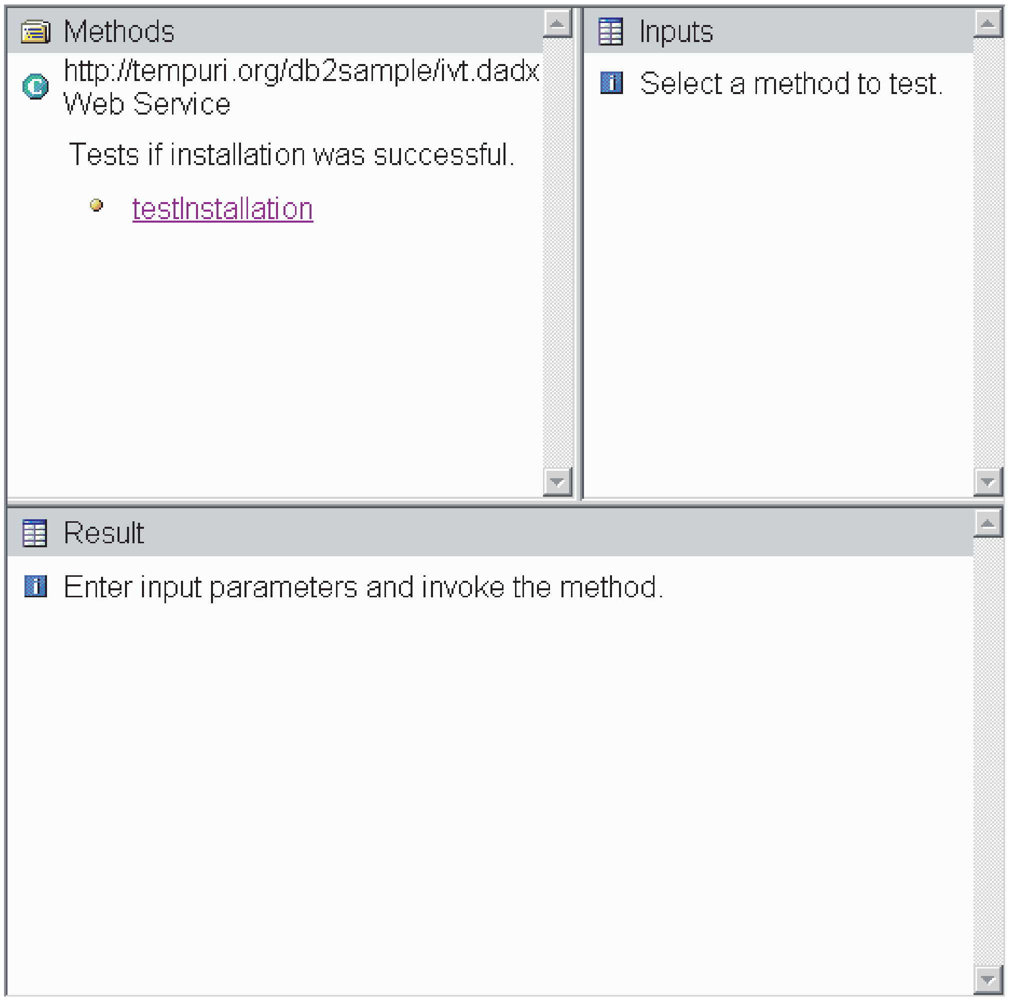
SAMPLE Database

The following DADX samples assume that the DB2 SAMPLE database has been created.

- On UNIX and Windows platforms, create the DB2 SAMPLE database using the db2sampl command.
- On OS/390, create the SAMPLE database by running the jobs DSNTEJ1 and DSNTEJ7 in the SDSNSAMP dataset.

db2sample/list.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/Spserver.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD

Figure 7. WORF sample page


Methods

http://tempuri.org/db2sample/ivt.dadx Web Service

Tests if installation was successful.

- testInstallation

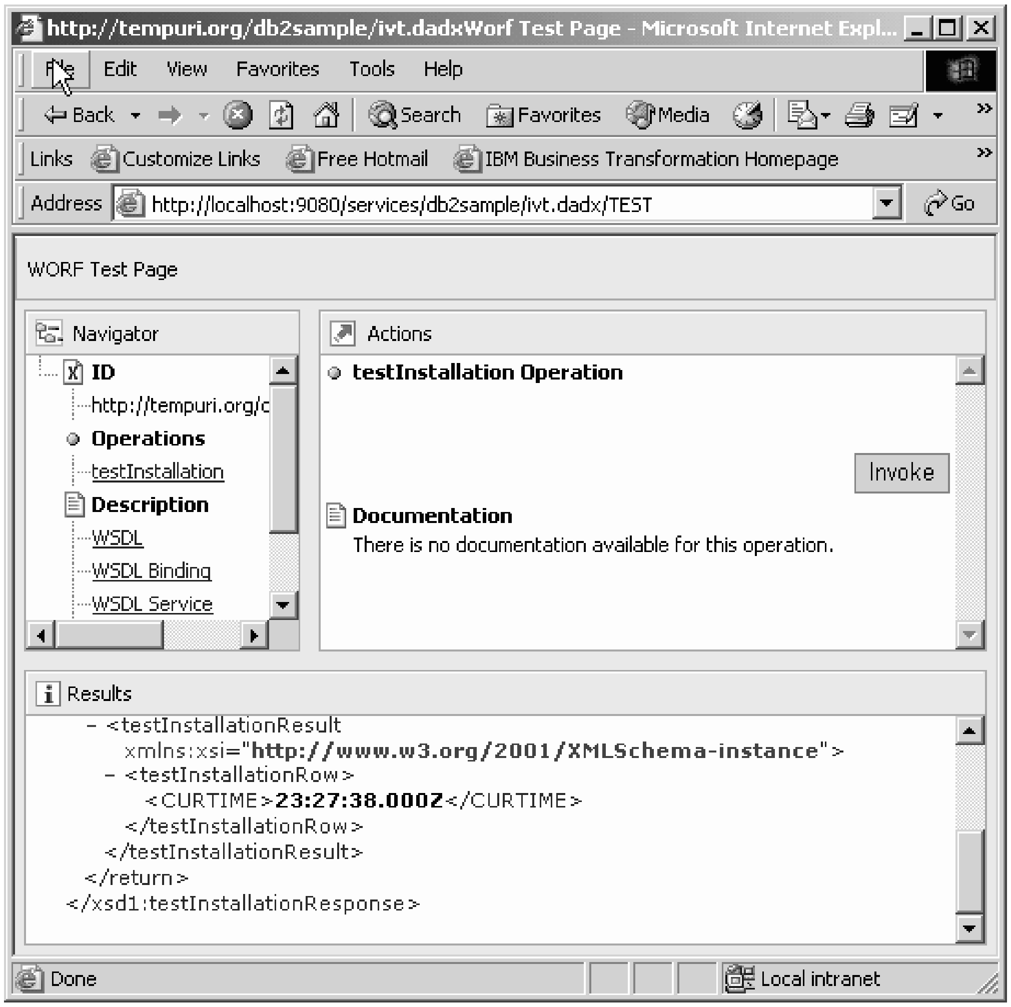
Inputs

Select a method to test.

Result

Enter input parameters and invoke the method.

Figure 8. WORF test facility


http://tempuri.org/db2sample/ivt.dadxWorf Test Page - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/db2sample/ivt.dadx/TEST
Go
WORF Test Page
Navigator
ID
http://tempuri.org/c
Operations
testInstallation
Description
WSDL
WSDL Binding
WSDL Service
Actions
testInstallation Operation
Invoke
Documentation
Results
Done
Local intranet

There is no documentation available for this operation.

```
    - <testInstallationResult
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      - <testInstallationRow>
          <CURTIME>23:27:38.000Z</CURTIME>
        </testInstallationRow>
      </testInstallationResult>
    </return>
  </xsd1:testInstallationResponse>
```

Figure 9. Result of WORF test-expected output

Deploying WORF examples on WebSphere Application Server Version 5.1 or later for Windows and UNIX

Web services provider includes examples that you can deploy.

Before you begin

- Install WebSphere Application Server on your work station in a path such as C:\WebSphere\Appserver (on your Windows environment).
- Install WORF.

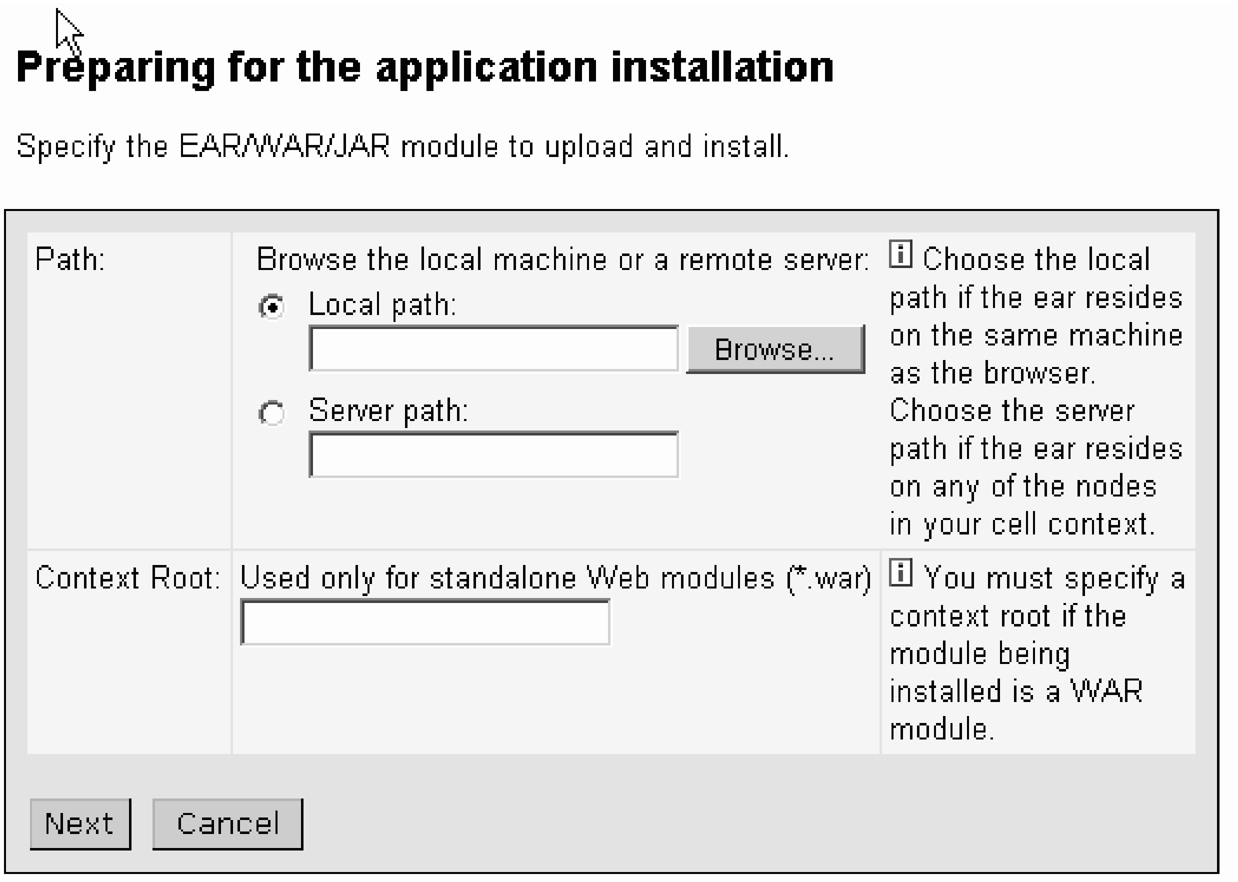
Procedure

To install and deploy the WORF examples:

1. Start the WebSphere Administration Server.
2. Open the Administrator's console by selecting **Start** → **Programs** → **IBM WebSphere** → **Administrator's Console**.
3. Select **Applications** → **Enterprise Applications** . The content window displays all of the enterprise applications that you installed on the current server.

4. Install websphere-services.war or axis-services.war as an enterprise application by clicking the **Install** push button.
   a. From the **Local path** field, click the **Browse** push button to locate the path to the correct services file that is included in the c:\WORF\lib directory. WORF ships two services files for you to choose from when running the samples. These files are websphere-services.war and axis-services.war. Select the correct services file for the SOAP engine that you are running.
   b. If you select to install the websphere-services.war, select the following options from the WebSphere Application Server administration console:
      - Select the **Generate Default Bindings** option to generate default bindings and mappings.
      - Select the **Use Binary Configuration** option to use the binary configurations.
      - Select the **Enable Class Reloading** option to enable class loading.

      You can install the websphere-services.war from the command line by using the worf_eas_admin.jacl script. For information on installing websphere-services.war from the command line see "Installing Web applications on the application server for DB2" on page 21
   c. Specify a context name for the Web application in the **Context Root** field. To execute the examples discussed here, you must specify `services` as the Context Root name. Figure 10 on page 36 shows the WebSphere Application Server Administrator's Console during the installation of the application:



Preparing for the application installation

Specify the EAR/WAR/JAR module to upload and install.

Path:	Browse the local machine or a remote server: [● Local path:] Browse... ○ Server path:	Choose the local path if the ear resides on the same machine as the browser. Choose the server path if the ear resides on any of the nodes in your cell context.
Context Root:	Used only for standalone Web modules (*.war)	You must specify a context root if the module being installed is a WAR module.

Figure 10. Specification of the application or module.

d. Click **Next**.

e. Accept all of the other defaults and click **Next** for the remainder of the Wizard. On the **Map virtual hosts for web modules** window, select the `.WAR` file and click **Next**. On the **Map modules to application servers** window, select the `.WAR` file and click **Next**. The configuration options specify a virtual host (for example: `default_host`) and an application server (for example: `Default Server`).

f. At the end of the Wizard, click **Finish**.

g. The final window displays the Save to Configuration. Click **Save**.

5. Verify that the database settings are correct (especially user ID and password) in the group.properties files.

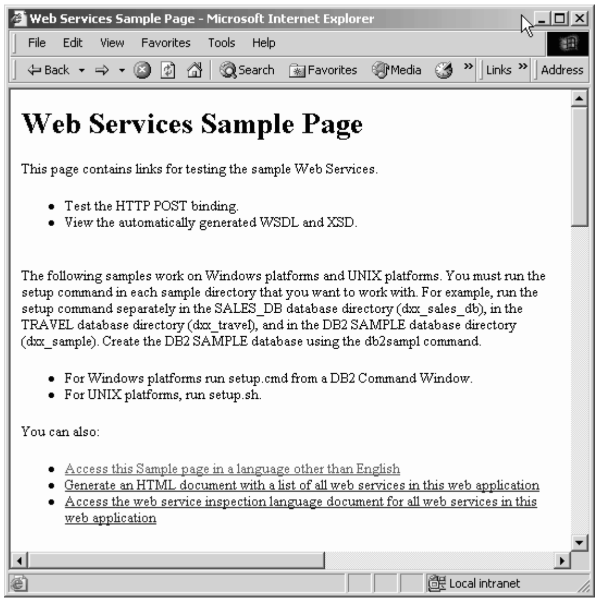
6. Issue setup.cmd in a DB2 command window in a Windows environment (the DB2 Command Line Processor window), or *setup.sh* in a UNIX environment command window in each of the database directories to create the database. For example, run setup.cmd in the dxx_sales_db directory to set up the SALES_DB database that uses DB2 XML Extender.

   If your application does not need the XML Extender functions, then you do not need to configure DB2 XML Extender. If you do use DB2 XML Extender, then configure your system with the appropriate DB2 environment variables. Refer to http://www.ibm.com/support/docview.wss?uid=swg21192376 for information about DB2 environment variables that you must enable when using DB2 XML Extender.

Attention: If you issue the setup command in the dxx_sample directory, the command drops and then recreates the SAMPLE database. If you use the SAMPLE database that is shipped with the DB2 Version 9.1 product, be aware that you will lose modifications that you have made to the database.

7. If you deployed your own application, copy the worf-servlets.jar file from the WORF directory to the WebSphere/AppServer/installedApps/<host>/<application WAR directory>/WEB-INF/lib directory.
8. Stop the current server.
9. Restart the server.
10. Verify that appropriate services.war is already running by selecting **Applications** → **Enterprise Applications**.
11. Open a browser window to test the installation by accessing the Web application welcome page. If you deployed the WORF sample application, and if you named the application services, then type http://localhost:9080/services from your browser to open the Web services sample page. The specific port number varies according to the WebSphere Application Server configuration.

Here are examples of the Web services sample pages.


Web Services Sample Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links Address
Local intranet

Web Services Sample Page

This page contains links for testing the sample Web Services.

- Test the HTTP POST binding.
- View the automatically generated WSDL and XSD.

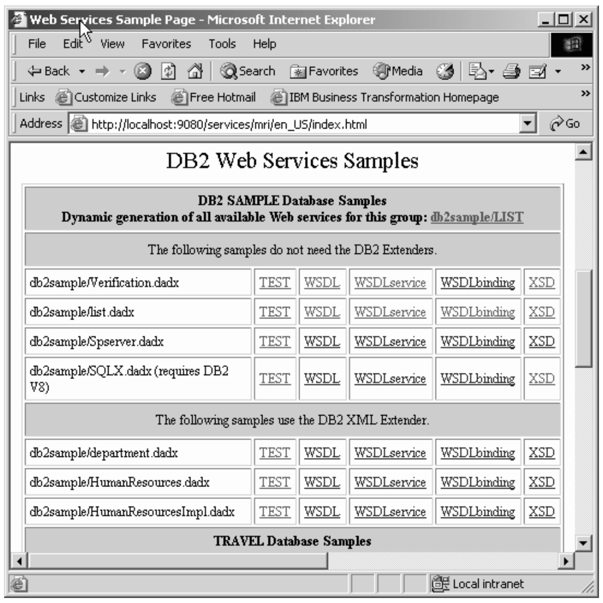
The following samples work on Windows platforms and UNIX platforms. You must run the setup command in each sample directory that you want to work with. For example, run the setup command separately in the SALES_DB database directory (dxx_sales_db), in the TRAVEL database directory (dxx_travel), and in the DB2 SAMPLE database directory (dxx_sample). Create the DB2 SAMPLE database using the db2sampl command.

- For Windows platforms run setup.cmd from a DB2 Command Window.
- For UNIX platforms, run setup.sh.

You can also:

- Access this Sample page in a language other than English
- Generate an HTML document with a list of all web services in this web application
- Access the web service inspection language document for all web services in this web application

Figure 11. Web services sample page.


Web Services Sample Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/mri/en_US/index.html
Go
Local intranet

DB2 Web Services Samples

DB2 SAMPLE Database Samples **Dynamic generation of all available Web services for this group:** db2sample/LIST					
The following samples do not need the DB2 Extenders.					
db2sample/Verification.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/list.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/Spserver.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/SQLX.dadx (requires DB2 V8)	TEST	WSDL	WSDLservice	WSDLbinding	XSD
The following samples use the DB2 XML Extender.					
db2sample/department.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/HumanResources.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
db2sample/HumanResourcesImpl.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
TRAVEL Database Samples					

Figure 12. Web services sample page-the test links

Now, you can click on some of the links to verify that the sample services work. The test page consists of a tree view of the operations, an input view and a results view. You access the test page from the TEST link within the Welcome Page of the samples, or by typing the following in your browser: <your-Web-server>:9080/ <context_root_name>/<group_name>/<dadx file>/TEST.

Installing and deploying the WORF examples in iSeries

You can install and deploy the Web services provider examples in the iSeries environment.

Procedure

To install the WORF examples:

1. Make sure that the worf.jar file is in
   ```
   /QIBM/UserData/WebASAEs4/worf/lib/app
   ```
   In this example, **WebASAEs4** is the version of WebSphere, and **worf** is the name of the WebSphere instance.
2. If file runtime.zip is not in directory /QIBM/UserData/java400/ext, execute the following commands:
   ```
   >qsh
   >ln -s  /QIBM/ProdData/OS400/Java400/ext/runtime.zip
             /QIBM/UserData/Java400/ext/runtime.zip
   ```
3. Copy the services.war into your tomcat\webapps directory.
4. If you already have a services.war file installed, then perform the following tasks:
   a. Stop Apache Jakarta Tomcat.
   b. Delete the services subdirectory under webapps and all of its contents.

      Note: Any of your previously deployed Web services in the services Web application will be lost with this action, so make sure this is acceptable.
   c. Restart Apache Jakarta Tomcat.

5. Stop and start Apache Jakarta Tomcat (unless you deleted the services directory in the previous step). The services context starts:

```
ContextManager: Adding context Ctx(\services)
```

6. Invoke the examples in the sample application by accessing the test page at http://<system>:<port>/services
   a. Invoke a sample with no parameters: http://<system>:<port>/services/travel/ZipCodes.dadx/findAll
   b. Invoke a sample with parameters:

   ```
   http://<system>:<port>/services/
       travel/ZipCodes.dadx/findCityByZipCode?zipcode=55901
   ```

7. Verify that your database settings are correct, especially user ID and password, in group.properties. If you do not use a value for user ID, the Web services code runs under QEJBSVR. Therefore, authorize this profile to any database objects that you want to access. Try the verification.dadx on your system (the dynamic test page and the WSDL).

Installing and deploying the WORF examples on Apache Jakarta Tomcat

You can install and deploy the WORF examples on Apache Jakarta Tomcat.

Procedure

To install the WORF examples:

1. Unjar the websphere-services.war or axis-services.war into your tomcat\webapps directory (depending on the SOAP engine that you install).
2. If you already have a websphere-services.war or axis-services.war file installed, perform the following tasks:
   a. Stop Apache Jakarta Tomcat.
   b. Delete the services subdirectory under webapps and all of its contents.

      Note: Any of your previously deployed Web services in the services web application will be lost with this action, so make sure that this is acceptable.
   c. Restart Apache Jakarta Tomcat.
3. Stop and start Apache Jakarta Tomcat (unless you deleted the services directory in the previous step). The services context starts:

   ```
   ContextManager: Adding context Ctx(\services)
   ```

4. Verify the installation by entering the following uniform resource locator (URL). The port number, designated here by *8080* depends on your own current machine:

   ```
   http://localhost:8080/services
   ```

   You should get a page that looks like Figure 11 on page 37.
5. Verify that your database settings are correct in the group.properties file, especially the user ID and password. Test the verification.dadx on your system (the dynamic test page and the WSDL).
6. To display the XML document, use Internet Explorer Version 5 or later or a text editor.
7. List the deployed SOAP services in your services context in your system. WORF automatically deploys the services, for each test you run. Click on the SOAP administration link from the Web services Sample Page.

Migrating Web services to WebSphere Federation Server Version 9.1

You can migrate earlier versions of the Web services provider to WebSphere Federation Server Version 9.1 on Windows or UNIX to work with WebSphere Application Server Version 5 or later or to work with Apache Jakarta Tomcat.

Procedure

To migrate to the Web services provider from an earlier version:

1. Locate the dxxworf.zip file in the following path: *DB2 Version 9.1 installed location*\samples\webservices\dxxworf.zip.
2. Locate the lib directory in the WebSphere Application Server.

   The lib directory can be found in the *WebSphere Application Server installed location*\WebSphere\AppServer\
3. Replace the worf.jar file by copying the lib\worf.jar from the Version 9.1 dxxworf.zip to the WebSphere Application Server lib directory.
4. For each application that you deployed with the earlier version of WORF, replace the JavaServer Pages files in the worf subdirectory of that application, with the files in the worf subdirectory of the websphere-services.war, or the axis-services.war.
5. In the lib directory of the WAR or EAR file of your Web application, replace the worf-servlets.jar file with the worf-servlets.jar file in the lib directory of the websphere-services.war file or the axis-services.war file.
6. Specify **apache-axis** or **was** as a parameter in the web.xml file for all servlet mappings.

   If you did not define the SOAP engine parameter in the web.xml file, the default SOAP engine is the primary SOAP engine that is supported by the application server. For the WebSphere Application Server it is the WebSphere soap engine. For Tomcat, it is Axis.
7. Re-deploy the application.

Migrating Web applications to work with WebSphere Federation Server Version 9.1

You can migrate Web applications that were created with earlier versions of the Web services provider to WebSphere Federation Server Version 9.1 on Windows or UNIX to work with WebSphere Application Server Version 5 or later or to work with Apache Jakarta Tomcat.

Procedure

To migrate your Web applications to the latest Web services provider from an earlier version:

1. Extract the Version 9.1 websphere-services.war or axis-services.war to a temporary directory, such as temp_v91.

   ```
   jar -xf temp_v91
   ```
2. Extract your Web application WAR file to a temporary directory such as temp_war.

   ```
   jar -xf temp_war
   ```
3. Optional: If you are using the Web services provider TEST page, replace all of the files in the WORF directory of the temp_war directory with the files in the WORF directory of the temp_v91 directory.

4. Replace the worf-servlets.jar file in the lib directory of the temp_war directory with the worf-servlets.jar file in the temp_v91 directory.
5. Modify the value of the soap-engine parameter in the web.xml file to reflect the SOAP engine to which you are migrating. For WebSphere Application Server, the default is **was** if the parameter is not already specified. The valid SOAP engines are **was** and **axis**.
6. Repackage the Web application by using the following command: jar -cf newWebApp.war.
7. Deploy the new Web application.

Introduction to using DB2 as a Web services provider – WORF

Web services are sets of business functions that applications or other Web services can invoke programmatically over the Internet by using a Web Service client interface.

Deprecating Web services object runtime framework (WORF)

The Web services objects runtime framework (WORF) is no longer supported and will not be updated.

In the Data Server Developer Workbench, you now can create Web services without writing document access definition extension (DADX) files. Use the Data Server Developer Workbench to create the SQL statements and stored procedures on which you can base the operations of your Web services. With the Data Server Developer Workbench you can now easily deploy a Web service

Read the Developing and deploying Web services for more detailed information about the feature within Data Server Developer Workbench.

To use your existing WORF applications, you must migrate your applications to Web services within the Data Server Developer Workbench. For the instructions on migrating to the Web services within the Data Server Developer Workbench, see Migrating Web applications that were developed for the Web Object Runtime Framework (WORF).

Accessing data by using a Web service

In the federated environment, you can define a basic Web service by using standard SQL statements, and DB2 XML Extender stored procedures. For Web services that involve advanced transformations between XML and relational data, use the DB2 XML Extender.

You can define a Web service to access data in the database by using a simple Document Access Definition Extension (DADX) file. You can create this DADX file, which is an XML file, by using a simple text editor, or by using the WebSphere® Studio Application Developer and the wizards available from WebSphere Studio.

The DADX file drives the Web services run-time environment, which includes various database management tools and the Web object runtime framework (WORF). The WORF runtime environment provides a simple mapping of XML schema to SQL data types. The DADX file can contain standard SQL statements, such as SELECT, INSERT, UPDATE, DELETE, and CALL statements to query and update a database and call stored procedures. If you want to process SQL statements at runtime, then you must enable the dynamic query service (DQS) that is provided by WORF by using a DADX file that includes only the <DQS/> tag.

If you do not use the WORF runtime environment, you need to write your own program to handle the details of creating the Web service, such as developing your own WSDL. Some of the functions that WORF provides include the following:

- Analyzing the Web service request
- Connecting to the database
- Executing the SQL request
- Encoding the output message from the SQL results
- Returning the message back to the client

The DADX file can also contain DB2 XML Extender elements, such as Document Access Definition (DAD) file references, XML collection operations to generate and store XML documents, or user-defined types (UDT), and user-defined functions (UDF). The DAD file defines a mapping between XML and relational data. DB2 XML Extender allows XML documents to be stored intact, and optionally indexed in side tables. DB2 XML Extender does this by using the XML column access method, or as a collection of relational tables by using the XML collection access method.

Web services provider features

The Web services provider uses features that are needed by application developers.

WORF provides the following features:

Resource-based deployment

A key feature of WORF is that it supports resource-based deployment of Web services.

Resource files, such as DADX files, describe the Web services to WORF, so that WORF can generate the appropriate Web services from these files. When you request the resource file, WORF loads the file and makes it available as a Web service. If you edit the resource file and request it again, WORF detects the change and loads the new version automatically. This process of automatically reloading the resource file makes Web service development more productive.

You can create your own resource files. The resource files must conform to specific syntax and semantic rules. The resource files can make references to each other (for example, a DADX file can contain references to DAD files). These references must be correct so that you can deploy the Web services properly.

In addition to specifying storage and retrieval operations on XML data, WORF allows stored procedures and SQL statements to be exposed as invokable Web service operations. You can expose any database stored procedure. WORF assumes that your stored procedure result sets have fixed metadata. Fixed metadata refers to data with a fixed number and a fixed shape, which implies a certain number of columns, with certain column names and data types. The operation signature includes the input and output parameters. You can execute stored procedures when you use dynamic query services (DQS) provided by WORF, with no fixed set of metadata or result sets that are required You can also specify SQL statements to select, insert, update and delete data. And, WORF provides simple mapping of XML schema to SQL data types. These particular features do not require the XML Extender.

Web services automatic reloading

Automatic reloading makes developing DADX Web services as simple as the developing of Java™ Server Pages.

During the course of development, you are likely to make frequent changes to your DADX files. WORF allows you to make changes to your DADX files while the application server is running, and automatically reloads the DADX file with the new updates. You can turn off automatic reloading when you deploy your DADX Web services to a production server.

Accessing the Web service with GET, POST, and SOAP bindings

You can access the Web service by using the HTTP GET, and SOAP bindings.

The Web object runtime framework (WORF) test page acts as a simple Hypertext Markup Language (HTML) client of the Web Service and uses the Hypertext Transfer Protocol (HTTP) POST binding. You can invoke the `listDepartments` operation with the HTTP GET, and POST bindings. The following example shows the basic syntax of the GET or POST binding:

```
http://server:port/contextRoot/group/dadx_file/operationName
```

If you have installed the WORF samples, you can type the following uniform resource locator (URL) to issue a GET request:

```
http://<yourWebAppServer:9080>/services/db2sample/HelloSample.dadx/listDepartments
```

The localhost port number, designated here by `<yourWebAppServer>` depends on your own current machine. The WORF `listDepartments` operation returns an XML response that you can save to a file. The HTTP response is the same for GET and POST.

```
<?xml version="1.0" ?>
<xsd1:listDepartmentsResponse
  xmlns:xsd1="http://schemas.ibm.com/sample/department.dadx/XSD"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<return>
 <xsd1:listDepartmentsResult
  xmlns:xsd1="http://schemas.ibm.com/sample/department.dadx/XSD"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<listDepartmentsRow>
  <DEPTNO>A00</DEPTNO>
  <DEPTNAME>SPIFFY COMPUTER SERVICE DIV.</DEPTNAME>
</listDepartmentsRow>
<listDepartmentsRow>
  <DEPTNO>B01</DEPTNO>
  <DEPTNAME>PLANNING</DEPTNAME>
</listDepartmentsRow>
<listDepartmentsRow>
  <DEPTNO>C01</DEPTNO>
  <DEPTNAME>INFORMATION CENTER</DEPTNAME>
</listDepartmentsRow>
<listDepartmentsRow>
  <DEPTNO>D01</DEPTNO>
  <DEPTNAME>DEVELOPMENT CENTER</DEPTNAME>
</listDepartmentsRow>
         ...
<listDepartmentsRow>
  <DEPTNO>E21</DEPTNO>
  <DEPTNAME>SOFTWARE SUPPORT</DEPTNAME>
</listDepartmentsRow>
</xsd1:listDepartmentsResult>
</return>
</xsd1:listDepartmentsResponse>
```

Figure 13. XML response document

The GET binding request does not send a request document. Instead, you attach all of the necessary parameters in the query string to the URL You can attach a query string to the URL with a question mark (?). The delimiter between any parameter=value pair is the ampersand (&). Any special characters must be URL encoded. The following example is a GET binding request that uses a query string with the question mark, and a delimiter.

```
http://server:port/contextRoot/group/dadx/
   operationName?param1=abc&param2=1234&param3=thi&20is&20a&20parameter
```

The following example is a dynamic query service. This is a GET binding request.

```
http://localhost:9080/services/db2sample/dqs.dadx/
   executeQuery?queryInputParameter=select+*+from+employee&extendedInputParameter=
      %3Cproperties%3E%0D%0A%3C%2Fproperties%3E%0D%0A
```

A POST binding issues an HTTP POST request. A POST bind request sends a request document. The document contains the request parameter, but the parameter is not in XML format. An HTTP client application, such as a Web browser, creates the request document. A Web browser usually creates a request document from input forms that are sent to the server.

The following syntax is a typical POST bind request:

```
http://server:port/contextRoot/group/dadx/operationName
```

The following example is a dynamic query service. This is a POST binding request.

```
http://localhost:9080/services/db2sample/dqs.dadx/executeQuery
```

The query is the same as the GET binding request, except that the information that follows the question mark (?) is in the request document, and not part of the URL. In the following example, the content type is www-urlencoded:

```
queryInputParameter=
   select+*+from+employee&extendedInputParameter=
%3Cproperties%3E%0D%0A%3C%2Fproperties%3E%0D%0A
```

The GET and POST response for the dynamic query service request is:

```
<?xml version="1.0"?>
<xsd1:executeQueryResponse
  xmlns:xsd1="http://schemas.ibm.com/db2/dqs/types/soap"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <queryOutputParameter>
      ...
   </queryOutputParameter>
</xsd1:executeQueryResponse>
```

HTTP GET, and POST bindings are generally the same as any other HTTP GET, and POST requests. The HTTP GET binding adds any input parameters to the operation to the uniform resource locator (URL). But, the HTTP POST binding sends the parameters in the request body.

WSDL from a DADX file

The Web services description language (WSDL) document is an XML vocabulary that is used to describe the interface of business services. The DADX document contains the information required to implement the Web service. It also contains the information required to generate the WSDL document that describes the Web service.

You can use the WSDL to publish services to a UDDI registry. WSDL allows development tools to programmatically create requester code and provider code for use in binding to a Web service. It also enables preconditioned applications to dynamically bind to a Web service. You can use WSDL to specify the data that are required for requests and responses. WSDL uses XML Schema for precise data definition.

A WSDL binding describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol. A WSDL SOAP binding can be either an RPC style binding or a document style binding. A SOAP binding can also have an encoded use or a literal use.

To generate the WSDL, submit the following uniform resource locator (URL). The *localhost* port number, designated here by <*yourWebAppServer*> depends on your own current machine:

```
http://yourWebAppServer:port/webapp_name/group_name/dadx_file.dadx/WSDL
```

Web services object runtime framework (WORF) dynamically generates the WSDL document. You can publish this in UDDI or some other Web service directory.

If you use the samples that WORF includes during the installation, you can submit the following URL:

```
http://yourWebAppServer/services/sales/PartOrders.dadx/WSDL
```

Web services documentation

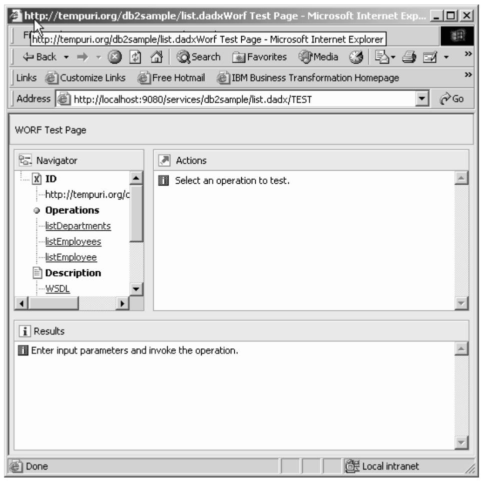
You can include documentation in the DADX file for the Web service as a whole and for each operation in the Web service.

Figure 14 illustrates how to add documentation:

```
<?xml version="1.0" encoding="UTF-8"?>
<DADX
  xmlns="http://schemas.ibm.com/db2/dxx/dadx">
  <documentation>
    Simple DADX example that accesses the SAMPLE database.
  </documentation>
  <operation name="listDepartments">
    <documentation>
      Lists the departments.
    </documentation>
    <query>
      <SQL_query>SELECT * FROM DEPARTMENT</SQL_query>
    </query>
  </operation>
</DADX>
```

Figure 14. HelloSample.dadx

The documentation can contain any valid XML. For proper display in a browser, you should use XHTML. If you use XHTML, then define the XHTML namespace for the documentation. When you request the test page, it also includes the documentation:


http://tempuri.org/db2sample/list.dadxWorf Test Page - Microsoft Internet Exp...

http://tempuri.org/db2sample/list.dadxWorf Test Page - Microsoft Internet Explorer

Address http://localhost:9080/services/db2sample/list.dadx/TEST

WORF Test Page

Navigator

ID

http://tempuri.org/c

Operations

listDepartments

listEmployees

listEmployee

Description

WSDL

Actions

Select an operation to test.

Results

Enter input parameters and invoke the operation.

Done

Local intranet

Figure 15. WORF test page with documentation

Web services that exist from Web services provider

Within a directory of Web applications, or a group of Web services, there is potentially a large number of Web services that you can use on your network. But before you can use these Web services, you must find them and get information about them. Web Services Inspection Language (WSIL) makes this search process easier.

Web services inspection language document

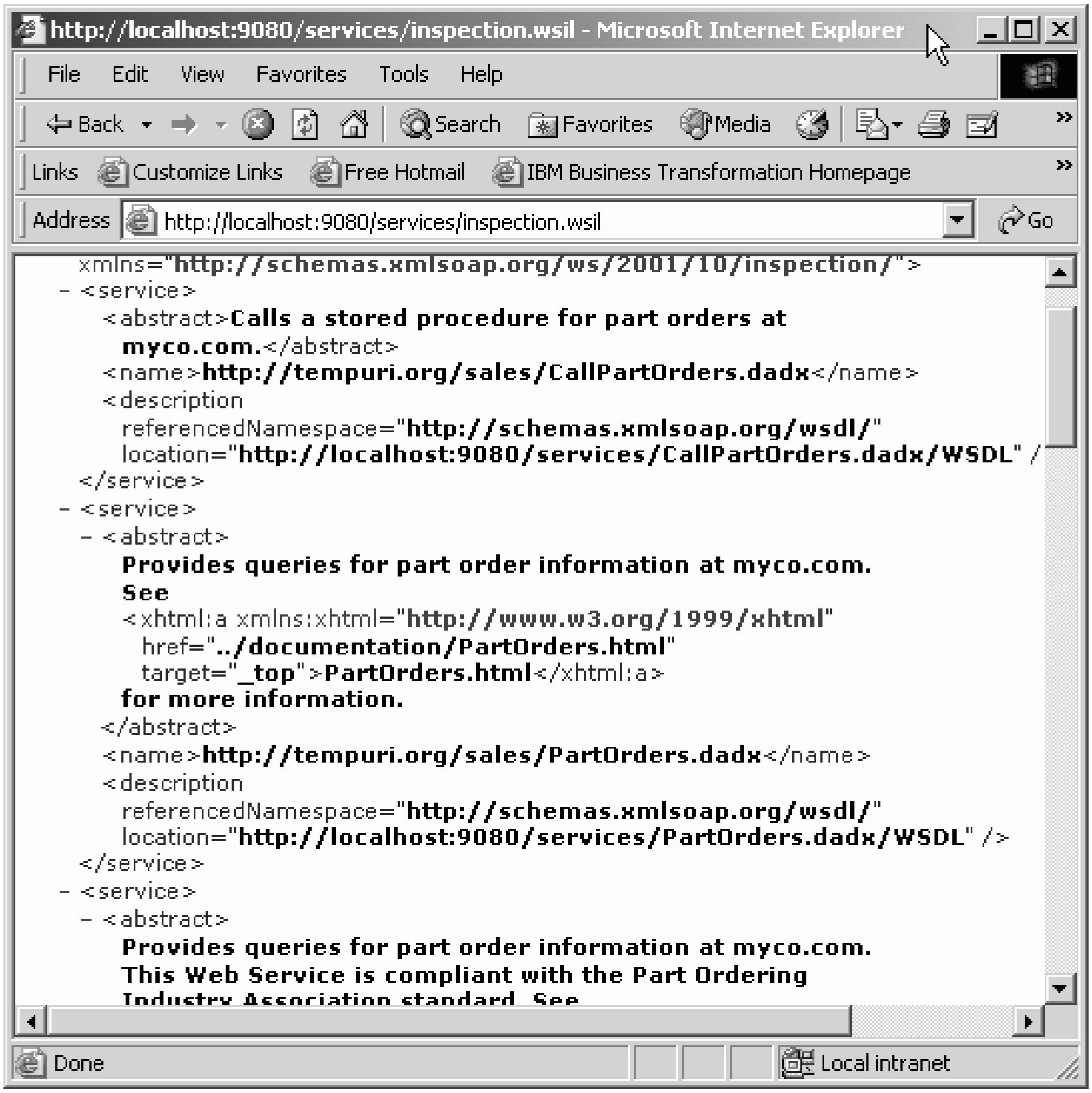
DB2® Web services provides a way to find the Web services operations that you need. By using WORF, you inspect all of the Web services available within an application, or within a group. The inspection generator produces an XML document that is a list of the Web services available to you. The list is a report of the Web services at the group directory level, if you run the generator from the group directory. The list is a report of the Web services at the application directory level if you run the generator from the application directory. You run the inspection generator from your browser by typing <your-Web-server>:9080/<context_root_name>/inspection.wsil in your browser.

The examples in this topic are based on the WORF samples and a WORF sample application named *services*

The following example creates a list of Web services that are available from the Web application named *services*:

```
http://localhost:9080/services/inspection.wsil
```

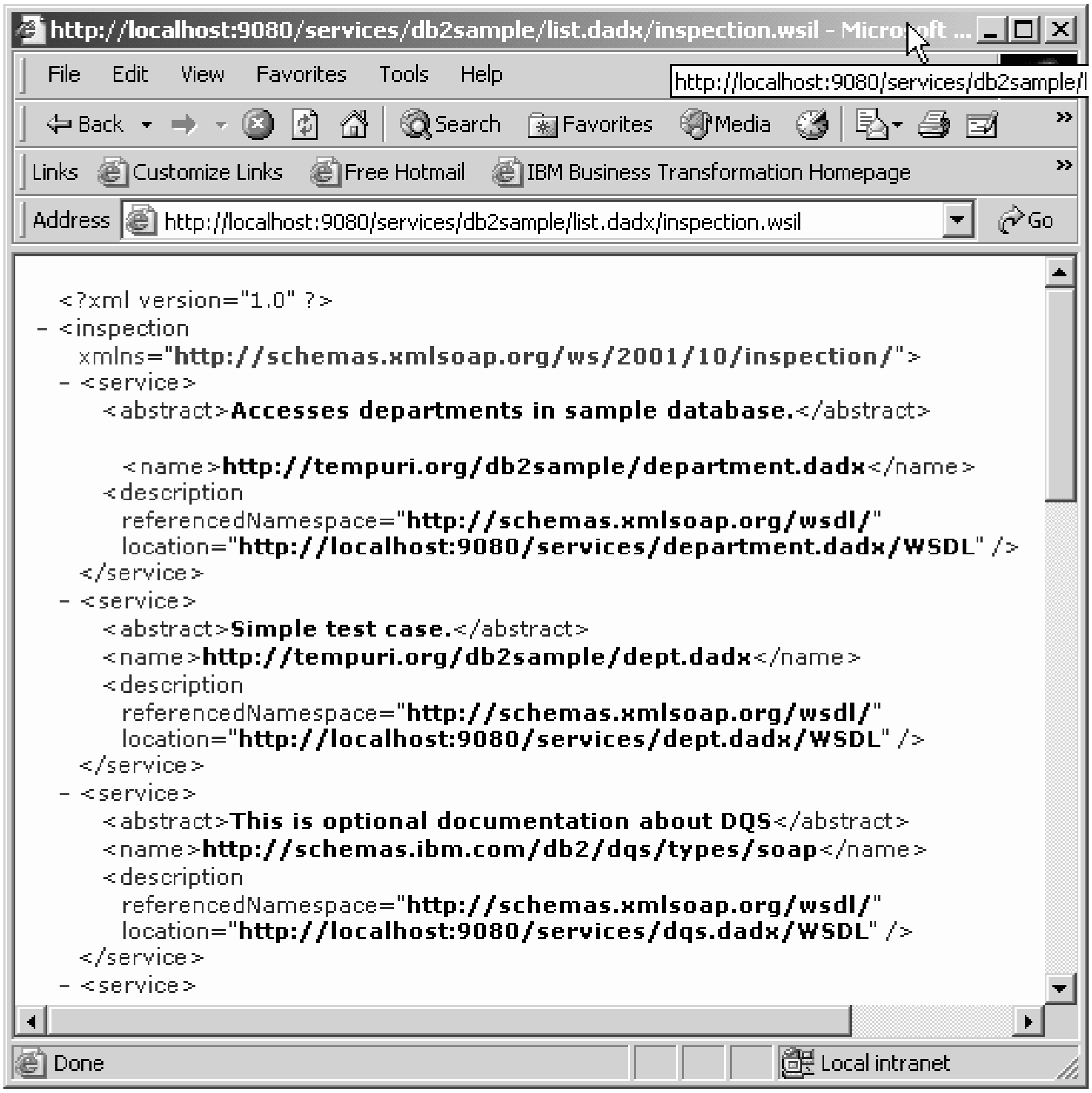
An inspection.wsil at the application level looks like this:


http://localhost:9080/services/inspection.wsil - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/inspection.wsil
Go
Done
Local intranet

```
   xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
 - <service>
     <abstract>Calls a stored procedure for part orders at
       myco.com.</abstract>
     <name>http://tempuri.org/sales/CallPartOrders.dadx</name>
     <description
       referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
       location="http://localhost:9080/services/CallPartOrders.dadx/WSDL" /
   </service>
 - <service>
   - <abstract>
       Provides queries for part order information at myco.com.
       See
       <xhtml:a xmlns:xhtml="http://www.w3.org/1999/xhtml"
         href="../documentation/PartOrders.html"
         target="_top">PartOrders.html</xhtml:a>
       for more information.
     </abstract>
     <name>http://tempuri.org/sales/PartOrders.dadx</name>
     <description
       referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
       location="http://localhost:9080/services/PartOrders.dadx/WSDL" />
   </service>
 - <service>
   - <abstract>
       Provides queries for part order information at myco.com.
       This Web Service is compliant with the Part Ordering
       Industry Association standard. See
```

Figure 16. WSIL at the application level

An inspection.wsil at the group level looks like this:


http://localhost:9080/services/db2sample/list.dadx/inspection.wsil - Microsoft ...
File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Links
Customize Links
Free Hotmail
IBM Business Transformation Homepage
Address
http://localhost:9080/services/db2sample/list.dadx/inspection.wsil
Go
Done
Local intranet

```
  <?xml version="1.0" ?>
- <inspection
    xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
  - <service>
      <abstract>Accesses departments in sample database.</abstract>

        <name>http://tempuri.org/db2sample/department.dadx</name>
      <description
        referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
        location="http://localhost:9080/services/department.dadx/WSDL" />
    </service>
  - <service>
      <abstract>Simple test case.</abstract>
      <name>http://tempuri.org/db2sample/dept.dadx</name>
      <description
        referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
        location="http://localhost:9080/services/dept.dadx/WSDL" />
    </service>
  - <service>
      <abstract>This is optional documentation about DQS</abstract>
      <name>http://schemas.ibm.com/db2/dqs/types/soap</name>
      <description
        referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
        location="http://localhost:9080/services/dqs.dadx/WSDL" />
    </service>
  - <service>
```

Figure 17. WSIL at the group level

To ensure that the WSIL that you generate includes a Web service, the Web service must conform to the following criteria:

- The DADX file that describes the Web service must be valid.
- You must define a group.properties file that belongs to the Web service.
- The web.xml file must contain appropriate servlet mappings that identify the group.

The servlet mapping in the web.xml file for the WSIL should look like the following example:

```
<servlet>
    <servlet-name>wsil</servlet-name>
    <display-name>wsil</display-name>
```

```
        <servlet-class>
           com.ibm.etools.webservice.rt.wsil.servlet.WSILInvoker
        </servlet-class>
        <init-param>
          <param-name>soap-engine</param-name>
          <param-value>apache-axis</param-value>
        </init-param>
        <load-on-startup>-1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>wsil</servlet-name>
        <url-pattern>/inspection.wsil</url-pattern>
    </servlet-mapping>
```

When a Web service is invoked, WORF reads the web.xml file for information about how to run the service, such as servlet information, and group information. The web.xml file includes the servlet definition so that the WSIL specification associates the correct Web operations with the WORF samples. WORF dynamically generates a WSIL document that contains all of the available Web services in the groups directory of the Web application server. If you add a Web service to the group.properties file after the Web application server is started, you do not need to restart the server before you start the WSIL generator.

Web services list page

Another kind of inspection document that you can produce is the Web services list page. The Web services list page is an HTML document that contains a list of all of the available Web services in an application directory or in the groups directory of the Web application server. The list also contains links to the WORF samples and the WSDL of the services. You can access this page by typing the following URL in your browser:

Application level

```
<your-Web-server>:9080/<context_root_name>/LIST
```

Group level

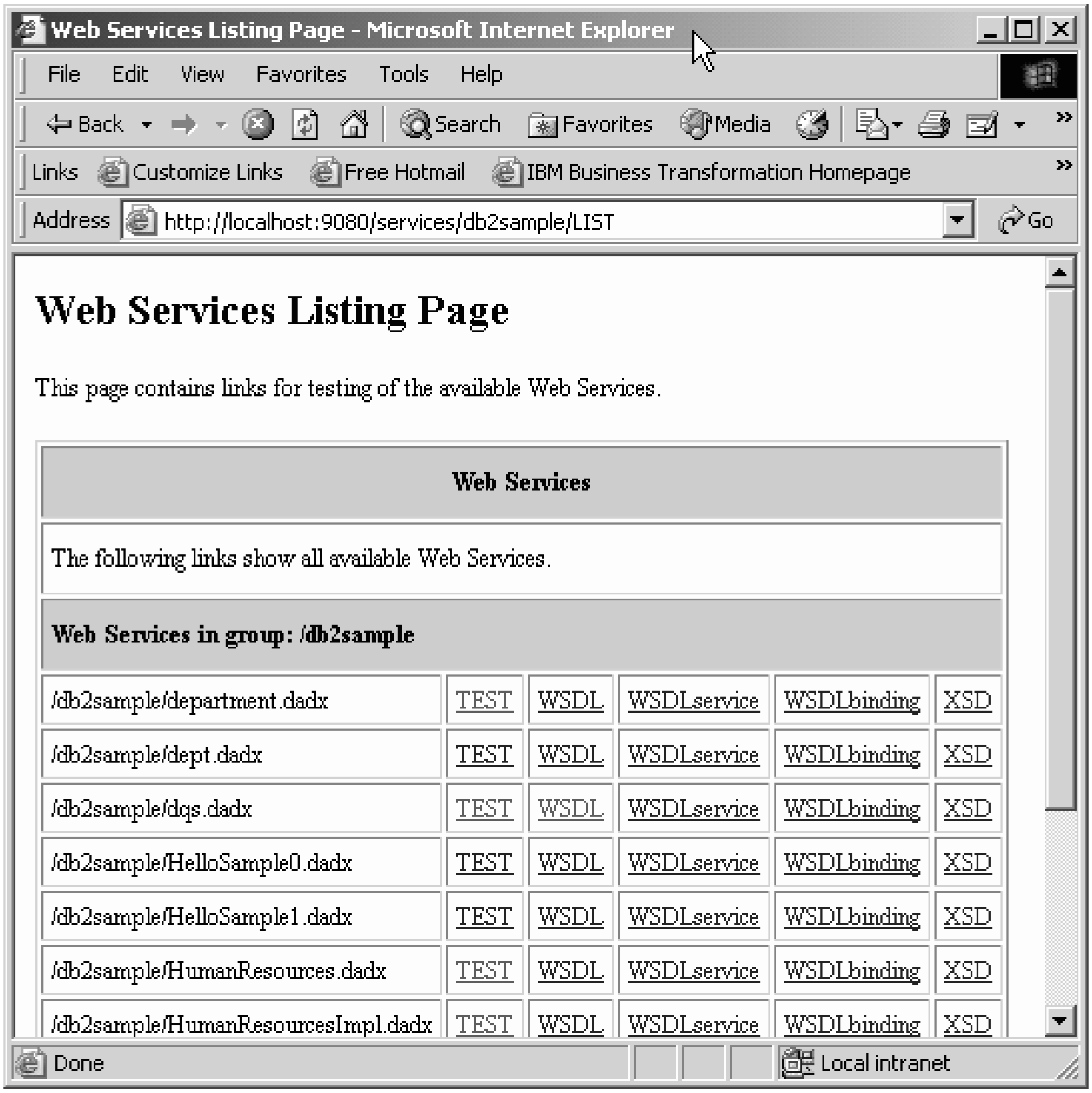
```
<your-Web-server>:9080/<context_root_name>/<group name>/LIST
```

The servlet mapping and URL pattern in the web.xml file for the list page should look like the following example:

```
<servlet>
        <servlet-name>list</servlet-name>
        <display-name>list</display-name>
        <servlet-class>
          com.ibm.etools.webservice.rt.list.servlet.ListInvoker
        </servlet-class>
        <init-param>
          <param-name>soap-engine</param-name>
          <param-value>apache-axis</param-value>
        </init-param>
        <load-on-startup>-1</load-on-startup>
    </servlet>

   <servlet-mapping>
        <servlet-name>list</servlet-name>
        <url-pattern>/LIST</url-pattern>
    </servlet-mapping>
```

A list page at the group level looks like this:


Web Services Listing Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/db2sample/LIST
Go
Done
Local intranet

Web Services Listing Page

This page contains links for testing of the available Web Services.

Web Services					
The following links show all available Web Services.					
Web Services in group: /db2sample					
/db2sample/department.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/dept.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/dqs.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HelloSample0.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HelloSample1.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HumanResources.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HumanResourcesImpl.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD

Figure 18. Web services list page

Chapter 2. Creating a Web services provider from a database

You can make DB2® Version 9.1 a Web services provider.

About this task

Some of these tasks are performed by a singe individual, and some are allocated for database administrators or Web application developers.

Procedure

To make DB2® Version 9.1 a Web services provider:

1. Create and configure the databases. This is usually done by the database administrator.
2. Optional: Enable the databases for DB2 XML Extender. The retrieveXML, storeXML, and XML column operations require DB2 XML Extender. See the administration chapters of *DB2 XML Extender Administration and Programming* to learn how to enable the databases for XML Extender.
3. Extract the websphere-services.war file or the axis-services.war file to a temporary location, such as temp_new by using the following command: `jar -xf temp_new`. The WAR file that you unjar depends on the SOAP engine that you want to use for the Web application.
   a. You can create the enterprise archive or Web archive (EAR or WAR) files on the WebSphere Application Server.
   b. You can create only WAR files on Tomcat.
4. Create, copy or rename a group.
5. Delete any groups that you do not want.
6. For each group, create or update the group.properties file with the database name and other parameters that are necessary for that group, which is located in the WEB-INF directory.
7. Create the DADX files and replace the DADX files in the group with the new DADX files.
8. Optional: Create the a Document Access Definition (DAD) file to map XML and the relational data conversions (required when you use XML Extender stored procedures).
9. Update other files that pertain to the group, such as group.imports as necessary.
10. Update the web.xml file to contain the servlet names and servlet mappings for your Web application.
11. Repackage all of the files to create a new WAR file for your Web application by using the following command: jar -cf myWebApp.war
12. Deploy the WAR file.
13. Verify that the Web service functions properly by using the DADX test page that is available if you deploy the Web services provider examples. You can copy the JavaServer Pages that are located in the install directory of the Web services provider in the websphere-services.war file or the axis-services.war file to test some of the Web services provider functionality in your application.

When the enterprise archive or Web archive (EAR or WAR) file is deployed, it becomes a folder containing the Web application where the developer can do further modifications. The following example shows the directory path for the enterprise archive or Web archive files:

WAR files
: **`Web application → group → DADX file (the Web service) → the SQL operations`**

EAR files
: **`Enterprise application → Web application → group → DADX file (the Web service) → the SQL operations`**

The EAR file is an enterprise application that is described in the Java™ 2 Enterprise Edition specification (J2EE). WAR files are also described in the J2EE specification. The Web application folder is on a server that is a collection of related files and tools. Web applications include the interfaces, program flow, program logic, and data access information to create an infrastructure for doing business over the Internet.

Defining a group of Web services

Groups are containers for Web services that share common configuration options. The groups directory contains the resources for all DADX Web service groups.

You can use groups to share these configuration options:
- Database configuration
- Namespace setup
- Message encoding setup

You create the group directory during the Web application configuration. This directory is in the WEB-INF\classes\groups subdirectory of the base directory of the Web application.

DADX files contain a description of the Web services. The Web services provider contains the implementation of the Web services and are therefore similar to Java™ classes. The classes directory is part of the Java CLASSPATH for the Web application. This means that the Java class loader can load your DADX files.

Within the groups directory, the Web services provider stores each group of DADX Web services in a directory with the same name as its servlet instance. The application server looks for the right servlet instance to call by the given URL that is based on the web.xml file.

Group imports file

WORF uses another resource, the group.imports file. This file is an optional resource that helps the Web service consumers or tools that use the generated WSDL to find the schemas that are used. If the group.imports file exists, then the WSDL document generates the imports elements based on the content of the group.imports file and the scope of the element. If no group.imports file exists, then no import elements are generated for WSDL documents for non-dynamic query services. Import elements are always generated for WSDL documents for dynamic query services. For dynamic query services, the WSDL document contains some data types that are in db2WebRowSet.xsd. With no group.imports to define a location of db2WebRowSet.xsd, WORF assumes that this schema file is in the default location, such as in the following example:

```
http://server:port/contextRoot/db2WebRowSet.xsd
```

Location of example DADX files

In the examples relating to the DB2® Web services, the WORF application stores the DADX files in the WEB-INF\classes\groups\dxx_sample directory, the WEB-INF\classes\groups\dxx_sales_db directory, and the WEB-INF\classes\groups\dxx_travel directory.

Defining the web.xml and group.properties files

There is a group.properties file for each group, and one web.xml file for all of the groups in a Web application. There is a servlet and a servlet-mapping for each group in the web.xml file.

About this task

You can create the web.xml and group.properties files by using the Rational® Web Developer for WebSphere Software. If you are migrating to a new version of WORF, make sure that the web.xml and group.properties files contain the values that you expect for your environment.

Procedure

To define a new group of DADX Web services, complete the following steps:

Defining the web.xml file

The Web services provider reads the web.xml file for information about how to run the Web service, such as servlet information, and group information, and to determine which SOAP engine classes to load.

About this task

You can create the web.xml file by using Rational Web Developer for WebSphere Software. If you are migrating to a new version of WORF, make sure that the web.xml file contains the values that you expect for your environment.

The default encoding for the web.xml file is UTF-8. You update the file in UTF-8 for OS/390 or z/OS, by sending the web.xml file to a UNIX or Windows system. You can send the file by using the File Transfer Protocol (FTP) binary transfer. Then update the file, and return the file to the original system.

Procedure

To define the web.xml file:

1. Choose a group name for the DADX group that reflects your application, such as *myapp_group*.
2. Edit the web.xml file in the WEB-INF directory, to define the group name. You can have multiple group names in the same web.xml file.

Elements required in the web.xml file

When editing the web.xml file, you must provide information about the servlets and the servlet mappings, as well as the SOAP engine.

<servlet>

The `<servlet>` element defines a new servlet instance for the group. At least one `<servlet>` element must exist for each group, but a group can have multiple `<servlet-mapping>` elements. See the Java servlet specification for more information on servlets. In the web.xml file example, the `<servlet-name>` element defines a group named `myapp_group`

<servlet-name>

The `<servlet-name>` element is a child element of the `<servlet>>` element and of the `<servlet-mapping>` element. The `<servlet-name>` element defines the name of the group. The `<servlet-name>` must be a valid directory name under the groups directory. You use this name to store the DADX resources for this group of Web services. For example: `myapp_group` is defined in both the `<servlet>` and `<servlet-mapping>` elements.

<servlet-mapping>

You must have at least one `<servlet-mapping>` element to introduce a mapping between a URL and the group. The child element <servlet-name>, defines the group name and must be the same as the directory name for the group, which also means that the `<servlet-name>` must be the same as in the `<servlet>` group. The `<servlet-name>` element is the link between the `<servlet>` and the `<servlet-mapping>` elements.

<url-pattern>

The `<url-pattern>` element is the uniform resource locator (URL) that is associated with the group. The `<servlet-mapping>` element associates the dxx_sales_db servlet with URLs of the form `/`*`url_pattern`*`/*`. The URL pattern must be of this form for WORF to operate correctly. For example: `/myapp/*`. The `<servlet-name>` in this example is `myapp_group`.

<init-param>

You can update the name of the SOAP engine that you want to use. The parameter name to specify the soap engine is **<soap-engine>**. If you want to use the IBM Web Service SOAP provider engine, then the parameter value is *was*. If you want to use Apache Axis, then the parameter value is *apache-axis*. If you do not specify a **<soap-engine>** parameter, the default soap engine is *was* in the WebSphere Application Server and *apache-axis* in Tomcat.

Example of a web.xml file

The following code shows an example of the web.xml file. The `<servlet-mapping>` element and the specific values that are defined are in **bold** :

```
web.xml <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
     DTD Web Application 2.2//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
 <servlet>
```

```
  <servlet-name>
    myapp_group
  </servlet-name>
  <servlet-class>
   com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker
  </servlet-class>
  <init-param>
  <param-name>
   faultListener
  </param-name>
  <param-value>
   org.apache.soap.server.DOMFaultListener
  </param-value>
  </init-param>
  <init-param>
  <param-name>
   soap-engine
  </param-name>
  <param-value>was</param-value>
  </init-param>
  <load-on-startup>-1</load-on-startup>
 </servlet>
<servlet-mapping>
<servlet-name>myyapp_group</servlet-name>
<url-pattern>/myapp/*</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

Defining the group.properties file

The group.properties file contains information about the database connection and other related information used by the Web services provider.

About this task

A group is a number of Web services operations that access a database. You can have one group for each database or even multiple groups for the same database, and within that group, you can define one or more DADX files. The DADX file, which specifically defines the Web service contains the operations that execute the Web service.

You can create the group.properties file by using the Rational Web Developer for WebSphere Software.

Procedure

To define the group.properties file:

1. Choose a group name for the DADX group that reflects your application, such as *myapp_group*.
2. From the groups directory, create a subdirectory with the name of the group that is specified in the `<servlet-name>` element added in the web.xml file. The subdirectory will contain the resources for this group.
3. In the group directory, create a group.properties file that defines the database connection information and other common attributes for each group of DADX Web services.

Example of a group.properties file

The following example shows what the group.properties might look like for the new group:

```
group.properties example
# myapp_group group properties
dbDriver=com.ibm.db2.jcc.DB2Driver

dbURL=jdbc:db2:sample
userID=
password=
namespaceTable=myapp.nst
autoReload=true
reloadIntervalSeconds=5
For Informix, use the following database driver and URL:
dbDriver=com.informix.jdbc.IfxDriver
dbURL=jdbc\:informix-sqli://::informixserver=
For OS/390 and z/OS, use the following database driver and URL:
dbDriver=COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
dbURL=jdbc:db2os390:
```

Replace the dbURL value in the above example with the following choices:

Type 2 connection

dbURL=jdbc:db2:<database name>

Type 4 connection

dbURL=jdbc:db2:<http://<host>:<port>/<database name>

Parameters for the group.properties file

The group.properties file is a standard Java properties file. You must define the group.properties with at least one parameter.

You define the group.properties based on the type of connection. There following types of connections are used for group.propertes:

Connection pooling
Define group.properties using parameter initialContextFactory with datasourceJNDI.

Regular JDBC database connections
Define group.properties using parameter dbURL with dbDriver.

If you define both types of connections, the Web application tries the data source first. If WORF cannot obtain the data source, then it tries the JDBC. The complete set of properties is listed below.

Database configuration parameters

There parameters are specific to the database configuration.

initialContextFactory
Used with `datasourceJNDI`. Required for WebSphere connection pooling. This parameter specifies the Java class name of the JNDI initial context factory that is used to locate the data source for the database. This parameter, along with the datasourceJNDI parameter, enables connection pooling.

datasourceJNDI
Used with `initialContextFactory`. Required for WebSphere connection pooling. The parameter specifies the JNDI name of the data source for the

database. When you use this with the `initialContextFactory` parameter, the parameter defines a data source for the database connection. This parameter enables connection pooling. You must define either the data source or the Java Database Connectivity (JDBC) connection.

dbDriver
Specifies the Java class name of the Java Database Connectivity (JDBC) driver for connecting to the database.

dbURL
Used with dbDriver. This parameter specifies the JDBC uniform resource locator (URL) of the database.

userID
The default is the user ID under which the Web services provider runs, which can be the same user ID used for connecting to the database. This parameter specifies the user ID for the database. The user ID is required for the DB2 universal JDBC driver that uses type 4 connections.

password
Used in conjunction with the user ID. This parameter specifies the password for the database. There are algorithms that are available to help you encode and decode your password. The password is required for the DB2 universal JDBC driver that uses type 4 connections.

enableXmlClob
Specifies whether retrieveXML operations will use the CLOB-based XML Extender stored procedures. The default value is *true*. This parameter is available only for backward compatibility. For OS/390 and z/OS platforms, either do not define this property, or always set the value to *true*.

Web Service configuration parameters

There parameters are specific to the Web service configuration.

groupNamespaceUri
Defines the target namespace in the generated Web service description language (WSDL) and XML schema files (XSD). The target namespace is for Web services in this group. The default is http://tempuri.org. To use the default, you must remove the groupNamespaceURI from the group.properties file. This parameter is only applicable to the RPC style encoding.

useDocumentStyle
Specifies which encoding to use. The default value is *false*, which means that the Web services at run time use RPC encoding. If you set this value to *true*, then the Web services at run time use document style and literal encoding. The Web services provider contains samples that are set to use RPC style. For new applications, use document style for maximum interoperability. The IBM Web Service SOAP provider engine only supports document style encoding.

namespaceTable
Specifies the resource name of the namespace table. The parameter references a Namespace Table (NST) resource that defines the mapping from DB2 XML Extender DTDIDs to XML Schema (XSD) namespaces and locations.

Runtime configuration parameters

autoReload
Optional, but mandatory with `reloadIntervalSeconds`. Specifies whether to reload a resource. Values can be *true* or *false*. The default value is *false*.

reloadIntervalSeconds
Optional, but mandatory with `autoReload`. Controls resource loading and caching. It specifies the integer automatic reloading time interval in seconds. The default is *0*, which means that WORF checks for a newer resource on every request.

The options autoReload and reloadIntervalSeconds control resource loading and caching. If autoReload is absent or false, then there is no resource reloading, and the application ignores reloadIntervalSeconds. If autoReload is true, then, when WORF accesses a resource, such as a DADX file, it compares the current time with the time at which the resource was previously loaded. If more than the value of `reloadIntervalSeconds` has passed, then WORF checks the file system for a newer version and reloads the changed resource. Automatic reloading is useful at development time, in which case set `reloadIntervalSeconds` to zero. If the Web services are in production, set `autoReload` to false, or set `reloadIntervalSeconds` to a large value to avoid impacting server performance.

Sample servlet for iSeries

The sample applications that are provided with the Web services provider includes a servlet that applies a style sheet to the generated XML. You can use this servlet example for any combination of XML Web services and XSL style sheets for iSeries.

After you define the web.xml file and the group.properties file, you can use the sample servlet. The servlet class and source are in the following directory:

```
 /QIBM/UserData/WebASAEs4/worf/
    installedApps/servicesApp.ear/
    services.war/WEB-INF/classes
```

The sample file contains the following setting:

```
serveServletsByClassnameEnabled="true"
```

Invoke the servlet by running the following file:

```
/QIBM/UserData/WebASAEs4/worf/
    installedApps/servicesApp.ear/
    services.war/WEB-INF/ibm-web-ext.xmi
```

You can recompile the servlet inside qshell by running the following compile statement:

```
javac -J-Djava.ext.dirs=/qibm/proddata/webasaes4/lib
        -d . SampleXSLTServlet.java
```

Invoke the servlet from the Internet by running the following file:

```
http://system:port/services/servlet/
SampleXSLTServlet?XML=
http://system:port/services/travel/ZipCodes.dadx/
   findAll&XSL=
file:///home/zipcodes.xsl
```

The above example assumes that the zipcodes.xsl file is in the /home subdirectory.

Definition of a DADX file

A document access definition extension (DADX) file specifies how to create a Web service. A Web service is a function that you invoke over the Web.

You can create a Web service by using a set of operations that are defined by SQL statements, stored procedure calls, or DAD files. Web services store XML documents or retrieve XML documents, including some that are managed by DB2® XML Extender. Web services that are specified in a DADX file are called `DADX Web services`, or IBM® DB2 Web services.

WORF provides the run-time support for invoking DADX documents as Web services. These Web services use the Apache Axis engine (Version 1.2) or the WebSphere Web services engine. Both of these SOAP engines are supported by WebSphere® Application Server and Apache Jakarta Tomcat.

The Web services developer creates the DADX document. The content of the DADX file determines if you can use dynamic queries in your Web application. A DADX file with a dynamic query services tag (</DQS>) contains only that tag which acts as a switch to enable dynamic queries for that group only. If the DADX file contains a dynamic query services tag (<DQS/>), then you can specify the SQL operations from a browser or embed the operations in an application if you installed the WORF test Web application. A non-dynamic DADX file defines a set of predetermined SQL operations and contains information that is used to create the Web service.

You can create DADX documents by using a simple text editor, or with tools that are provided in WebSphere Studio with only minimal knowledge of XML or SQL.

Defining the Web service with the document access definition extension file

The document access definition extension (DADX) file specifies a Web service. It does this by using SQL statements, a list of parameters, and optionally, document access definition (DAD) file references that define a set of operations.

You can define a set of dynamic Web service operations with a DADX file that contains only the dynamic query service tag (<DQS/>). Operations are similar to methods that you can invoke.

You can define the operations in a DADX Web service by the following operation types:

- SQL Operations (non-dynamic)

  <query>
  Queries the database by using a select operation

  <update>
  Performs an update, insert or delete operation on the database

  <call> Calls stored procedures

- SQL Operations (dynamic query services)

  <getTables>
  Retrieves a description of available tables.

  <getColumns>
  Retrieves a description of columns.

  <executeQuery>
  Issues a single SQL statement.

  <executeUpdate>
  Issues a single INSERT, UPDATE, DELETE.

<executeCall>
Calls a single stored procedure.

<execute>
Issues a single SQL statement.

- XML collection operations (requires DB2® XML Extender)

<retrieveXML>
Generates XML documents

<storeXML>
Stores XML documents

Syntax of the DADX file

The Document Access Definition Extension (DADX) file is an XML document. This topic illustrates the syntax of the DADX file.

DADX syntax definitions and Figure 19 on page 63 describe the elements of the DADX. The numbers next to the nodes and elements in DADX syntax definitions identify the child groupings. The numbering scheme expresses the XML document hierarchy. For example, when the identifiers change from 1.3 (result_set_metadata) to 1.3.1 (column), this means that the column is a child of result_set_metadata. A change from 1.1 (documentation) to 1.2 (implements) means that these elements are siblings.

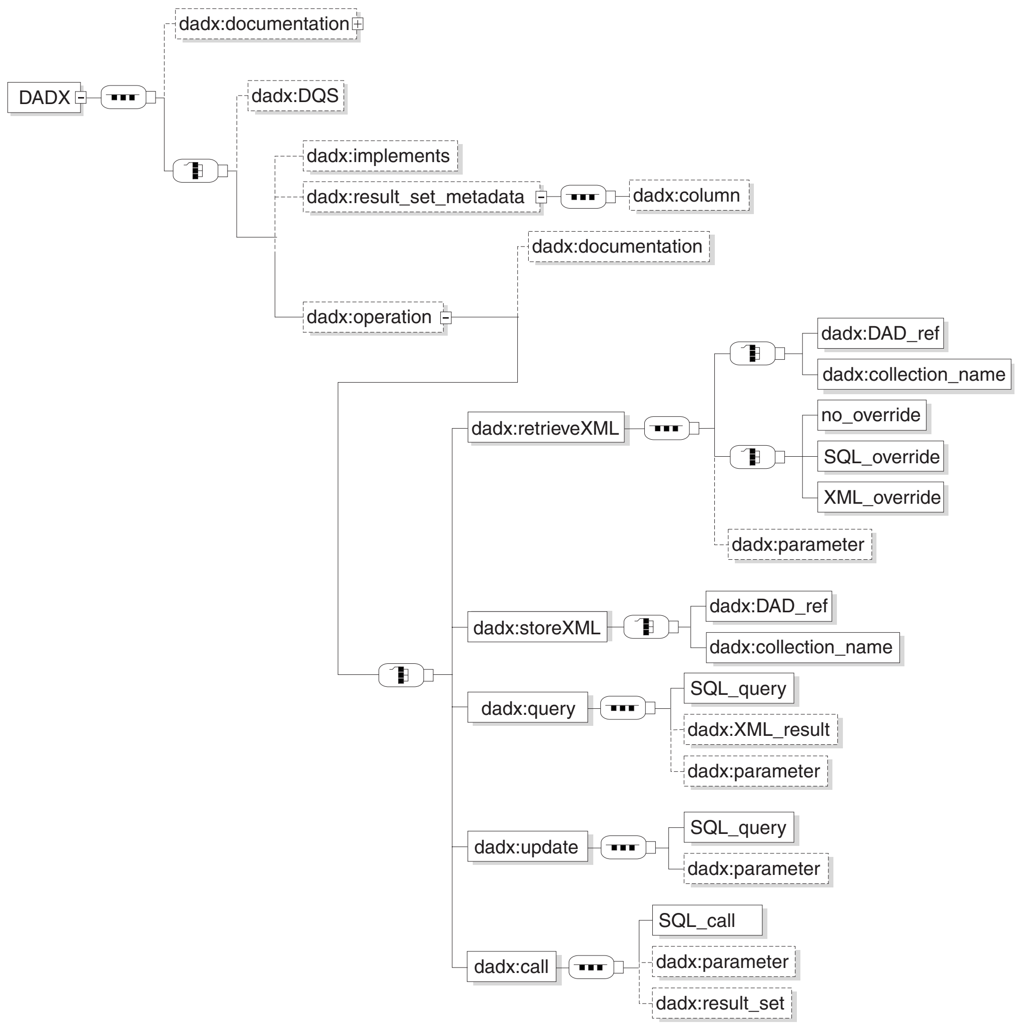


Figure 19. DADX syntax

0. Root element: <DADX>

Attributes:

xmlns:dadx
The namespace of the DADX.

xmlns:xsd
The namespace of the Extensible Markup Language (XML) Schema specification

Children:

0.1 <documentation>
Specifies a comment or statement about the purpose and content of the Web service. You can use XHTML tags.

1. DADX functions that specify non-dynamic operations

1.2 <implements>
Specifies the namespace and location of the Web service description files. It allows the service implementer to declare that the DADX Web service implements a standard Web service described by a reusable WSDL document defined elsewhere; for example, in an UDDI registry.

1.3 <result_set_metadata>

Stored procedures can return one or more result sets. You can include them in the output message. Metadata for a stored procedure result set must be defined explicitly in the non-dynamic DADX using the <result_set_metadata> element. At run-time, you obtain the metadata of the result set. The metadata must match the definition contained in the DADX file.

Note: You can only invoke stored procedures that have result sets with fixed metadata.
This restriction is necessary in order to have a well-defined WSDL file for the Web Service. A single result set metadata definition can be referenced by several <call> operations, using the <result_set> element. The result set metadata definitions are global to the DADX and must precede all of the operation definition elements.

Attributes:

name Identifies the root element for the result set.

rowname
Used as the element name for each row of the result set.

Children:

1.3.1 <column>

Defines the column. The order of the columns must match that of the result set returned by the stored procedure. Each column has a name, type, and nullability, which must match the result set.

Attributes:

name Required. This specifies the name of the column.

type Required if you do not specify **element**. It specifies the type of column.

element
Required if you do not specify **type**. It specifies the element of column.

as Optional. This provides a name for a column.

nullable
Optional. Nullable is either true or false. It indicates whether column values can be null.

1.4 <operation>

Specifies a Web service operation. The operation element and its children specify the name of an operation, and the type of operation the Web service performs. Web services can compose an XML document, query the database, or call a stored procedure. A single DADX file can contain multiple operations on a single database or location. The following list describes these elements.

- Attribute:

  name A unique string that identifies the operation. The string must be unique within the DADX file. For example: `"findByColorAndMinPrice"`

- Children:

  Document the operation with the following element:

  1.4.1 <dadx:documentation>
  Specifies a comment or statement about the purpose and content of the operation. You can use XHTML tags.

  1.4.2 <retrieveXML>

  This element specifies to generate zero or one XML documents from a set of relational tables when using the XML collection access method. Depending on whether you specify a DAD file or an XML collection name, the operation calls the appropriate XML Extender composition stored procedure.

  Children:

  – Specify which of these stored procedures you want to use. You do this by passing either the name of a DAD file, or the name of the collection by using one of the following elements:

    1.4.2.1 <DAD_ref>
    The content of this element is the name and path of a DAD file. If you specify a relative path for the DAD file, then the application assumes that the current working directory is the group directory.

    1.4.2.2 <collection_name>
    The content of this element is the name of the XML collection. You define collections by using the XML Extender administration interfaces, as described in DB2 XML Extender Administration and Programming.

  – Specify override values with one of the following elements:

    1.4.2.3 <no_override/>
    Specifies that the values in the DAD file are not overridden. Required if you do not specify either <SQL_override> or <XML_override>.

1.4.2.4 <SQL_override>
Specifies to override the SQL statement in a DAD file that uses SQL mapping.

1.4.2.5 <XML_override>
Specifies to override the XML conditions in a DAD file that uses RDB mapping.

– Define parameters by using the following element:

1.4.2.6 <parameter>
Required when referencing a parameter in an <SQL_override> or an <XML_override> element. This element specifies a parameter for an operation. Use a separate parameter element for each parameter referenced in the operation. Each parameter name must be unique within the operation. A parameter must have its contents defined by either an XML Schema element (a complex type) or a simple type.

Attributes:

name The unique name of the parameter.

element
Use the ″element″ attribute to specify an XML Schema element.

type Use the ″type″ attribute to specify a simple type.

kind Specifies whether a parameter passes input data, returns output data, or does both. The valid values for this attribute are:

- in

1.4.3 <storeXML>

This element specifies to store (decompose) an XML document in a set of relational tables using the XML collection access method. Depending on whether you specify a DAD file or an XML collection name, the operation calls the appropriate XML Extender decomposition stored procedure. Children:

– Specify which of these stored procedures you want to use. You do this by passing either the name of a DAD file, or the name of the collection by using one of the following elements:

1.4.3.1 <DAD_ref>
The content of this element is the name and path of a DAD file. If you specify a relative path for the DAD file, the application assumes that the current working directory is the group directory.

1.4.3.2 <collection_name>
The content of this element is the name of an XML collection. You define collections by using the XML Extender administration interfaces, as described in DB2 XML Extender Administration and Programming.

1.4.4 <query>
Specifies a query operation. You define the operation by using an SQL SELECT statement in the <SQL_select> element. The statement can have zero or more named input parameters. If the statement has input parameters then each parameter is described by a <parameter> element.

This operation maps each database column from the result set to a corresponding XML element. You can specify XML Extender user-defined types (UDTs) in the <query> operation. However, this requires an <XML_result> element and a supporting document type definition (DTD) that defines the type of the XML column queried.

Children:

1.4.4.1 <SQL_query>
Specifies an SQL SELECT statement.

1.4.4.2 <XML_result>
Optional. This defines a named column that contains XML documents. The XML Schema element of its root must define the document type.

Attributes:

name Specifies the root element of the XML document stored in the column.

element
Specifies the particular element within the column

1.4.4.3 <parameter>
Required when referencing a parameter in the <SQL_query> element. It specifies a parameter for an operation. Use a separate parameter element for each parameter referenced in the operation.

Each parameter name must be unique within the operation. A parameter must have its contents defined by one of the following: an XML Schema element (a complex type) or a simple type.

Attributes:

name The unique name of the parameter.

element
Use the ″element″ attribute to specify an XML Schema element.

type Use the ″type″ attribute to specify a simple type.

kind Specifies whether a parameter passes input data, returns output data, or does both. The valid values for this attribute are:

– in

1.4.5 <update>

The operation is defined by an SQL INSERT, DELETE, or UPDATE statement in the <SQL_update> element. The statement can have zero or more named input parameters. If the statement has input parameters then each parameter is described by a <parameter> element.

Children:

1.4.5.1 <SQL_update>

This specifies an SQL INSERT, UPDATE, or DELETE statement.

1.4.5.2 <parameter>

Required when referencing a parameter in the <SQL_update> element. It specifies a parameter for an operation. Use a separate parameter element for each parameter referenced in the operation. Each parameter name must be unique with the operation. A parameter must have its contents defined by one of the following: an XML Schema element (a complex type) or a simple type.

Attributes:

name The unique name of the parameter.

element
Use the ″element″ attribute to specify an XML Schema element.

type Use the ″type″ attribute to specify a simple type.

kind Specifies whether a parameter passes input data, returns output data, or does both. The valid values for this attribute are:

- in

1.4.6 <call>

Specifies a call to a stored procedure. The processing is similar to the update operation, but the parameters for the call operation can be defined as ′in′, ′out′, or ′in/out′. The default parameter kind is ′in′. The ′out′ and ′in/out′ parameters appear in the output message.

1.4.6.1 <SQL_call>

Specifies a stored procedure call.

1.4.6.2 <parameter>

Required when referencing a parameter in an <SQL_call> element. This specifies a parameter for an operation. Use a separate parameter element for each parameter referenced in the operation. Each parameter name must be unique within the operation. A parameter must have its contents defined by one of the following: an XML Schema element (a complex type) or a simple type.

Attributes:

name The unique name of the parameter.

element
Use the ″element″ attribute to specify an XML Schema element.

type Use the ″type″ attribute to specify a simple type.

kind Specifies whether a parameter passes input data, returns output data, or does both. The valid values for this attribute are:

- in
- out
- in/out

1.4.6.3 <result_set>

This defines a result set and must follow any <parameter> elements. The result set element has a name which must be unique among all the parameters and result sets of the operation. It must refer to a <result_set_metadata> element. One <result_set> element must be defined for each result set returned from the stored procedure.

Attributes:

name A unique identifier for the result sets in the SOAP response.

metadata
A result set metadata definition in the DADX file. The identifier must refer to the name of an element.

2. <DQS>
Dynamic query services.

A simple DADX file

The following example is a simple DADX file that contains one operation with an SQL query.

This DADX file is for non-dynamic queries. See Configuring and running dynamic database queries as part of Web services provider for an example of a DADX file used to enable dynamic query services.

Figure 20 shows a DADX file that defines a simple Web service:

```
<?xml version="1.0" encoding="UTF-8"?>
<DADX
  xmlns="http://schemas.ibm.com/db2/dxx/dadx"
  >
  <documentation>
    Simple DADX example that accesses the SAMPLE database.
  </documentation>
  <operation name="listDepartments">
    <documentation>
      Lists the departments.
    </documentation>
    <query>
      <SQL_query>SELECT * FROM DEPARTMENT</SQL_query>
    </query>
  </operation>
</DADX>
```

Figure 20. Simple DADX file

This simple DADX file defines a Web service with a single operation named `listDepartments` which lists the contents of the DEPARTMENT table. The operation name identifies the Web service activity, and is similar to a method name in programming languages.

Using overrides in the DADX file

The DADX file can override XML values and SQL statements in the DAD file by using the <XML_override> and <SQL_override> elements.

The type of override is determined by whether the DAD file uses SQL mapping or RDB mapping. If you do not need to override the DAD values, use the <no_override/> element, shown in Figure 35 on page 105.

The following example uses an SQL override statement.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <documentation >
    Provides queries for part order information at myco.com.
    See <xhtml:a href="../documentation/PartOrders.html" target="_top">
         PartOrders.html</xhtml:a> for more information.
  </documentation>
  <operation name="findAll">
    <documentation >
      Returns all the orders with their complete details.
    </documentation>
    <retrieveXML>
      <DAD_ref>getstart_xcollection.dad</DAD_ref>
        <SQL_override>
           select o.order_key, customer_name, customer_email,
            p.part_key, color, quantity, price, tax, ship_id, date, mode
           from order_tab o, part_tab p,
             table(select substr(char(timestamp(generate_unique())),16)
               as ship_id, date, mode, part_key from ship_tab) s
           where p.order_key = o.order_key and s.part_key = p.part_key
               order by order_key, part_key, ship_id
       </SQL_override>
   </retrieveXML>
  </operation>
</DADX>
```

Figure 21. Example of a DADX file that generates an XML document with an SQL override

Although you can override the SQL statement, the new SQL statement must produce a result set that is compatible with the SQL mapping defined in the DAD file. For example, the column names that appear in the DAD file must also appear in the SQL override.

If the DAD file uses RDB node mapping, you have to override the RDB nodes by using the <XML_override> element. RDB node elements define DB2 tables, columns, and conditions that are to contain XML data. The example in Figure 22 on page 72 shows a DADX file that references an RDB node DAD file. The <XML_override> element content overrides the conditions specified in the DAD file. The override string can contain input parameters using the host variable syntax. You must define the name and type of all parameters in a list of parameter elements that are uniquely named within this operation. In this example, the override parameter overrides the query by limiting the price to be greater than $50.00 and restricting the date to be greater than 1998-12-01.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <documentation >
    Provides queries for part order information at myco.com.
    See <xhtml:a href="../documentation/PartOrders.html" target="_top">
         PartOrders.html</xhtml:a> for more information.
  </documentation>
  <operation name="findByExtendedPriceAndShipDate">
    <documentation >
      Returns all the orders with an extended price greater than $50.00
      and a ship date later than 1998-12-01.
    </documentation>
<retrieveXML>
      <DAD_ref>order_rdb.dad</DAD_ref>
        <XML_override>
           /Order/Part/ExtendedPrice > 50.00 AND
           Order/Part/Shipment/ShipDate > '1998-12-01'
        </XML_override>
    </retrieveXML>
</operation>
</DADX>
```

Figure 22. Example of a DADX file that generates an XML document with an XML override

Declaring and referencing parameters in the DADX file

You declare parameters with a <parameter> element. The parameters have simple XML schema file (XSD) types that correspond to the built-in SQL data types.

You can use parameters in each of the operations. The <SQL_query>, <SQL_update>, and <SQL_call> statements for the SQL operations can reference parameters. The Extensible Markup Language (XML) and SQL overrides that you use in the <retrieveXML> and <storeXML> operations can also reference parameters. You declare the parameters by using the <parameter> element. The parameters have simple XML Schema types that correspond to the built-in SQL data types. Table 2 describes the supported types.

Table 2. Supported XML Schema and SQL types

XML Schema Simple Type	SQL Type
string	CHAR, VARCHAR, CLOB, LONGVARCHAR
decimal	DECIMAL, NUMERIC
int	INTEGER
short	SMALLINT
float	FLOAT
double	REAL, DOUBLE PRECISION
date	DATE
time	TIME
timestamp	TIMESTAMP
long	BIGINT
byte	TINYINT

To reference a parameter, use a colon prefix. For example:

```
<SQL_query>
 select * from order_tab where customer_name =:customer_name
</SQL_query>
```

To define the parameter, use the <parameter> element, as in the following example:

```
<parameter name="customer_name" type="xsd:string"/>
```

You must define each parameter that you reference with a <parameter> element. The name attribute for this element identifies the parameter and must be unique within the operation.

The example in Figure 23 shows a query operation that retrieves a set of relational data by using an SQL SELECT statement. The statement contains one input parameter by using the parameter syntax.

```
<operation name="findCustomerOrders">
    <documentation>Returns all the orders for a given customer.
    </documentation>
    <query>
      <SQL_query>select * from order_tab where customer_name =
        :customer_name</SQL_query>
      <parameter name="customer_name" type="xsd:string"/>
    </query>
  </operation>
```

Figure 23. Query operation with a parameter

The example in Figure 24 shows parameters in an SQL override that are used by a retrieveXML operation:
You can modify the WHERE clause of the SQL statement to include search

```
<operation name="findByColorAndMinPrice">
    <documentation>Returns all the orders that have the specified color and
       at least the specified minimum price.
    </documentation>
    <retrieveXML>
      <DAD_ref>getstart_xcollection.dad
</DAD_ref>
      <SQL_override>
       select o.order_key, customer_name, customer_email,
         p.part_key, color, quantity, price, tax, ship_id, date, mode
       from order_tab o, part_tab p,
         table(select substr(char(timestamp(generate_unique())),16)
           as ship_id, date, mode, part_key from ship_tab) s
       where p.order_key = o.order_key and s.part_key = p.part_key
         and color = :color and price >= :minprice
       order by order_key, part_key, ship_id
      </SQL_override>
      <parameter name="color" type="xsd:string">
      <parameter name="minprice" type="xsd:decimal">
    </retrieveXML>
  </operation>
```

Figure 24. SQL override used by a retrieveXML operation

conditions. The SQL override can include one or more parameters that are identified by using a colon. In this example, `findByColorAndMinPrice` references `:color` and `:minprice`.

DADX operation examples

The following samples show DADX files with query, update, call, retrieve, and store operations.

Example 1: Query operation

This example shows a Query operation, using the default tags. This example does not need XML Extender. This operation selects all of the orders for a given customer. To run this sample, you need the sales_db XML Extender sample database.

```
<?xml version="1.0"?>
 <DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <documentation>
      mycompany part orders service.
    </documentation>
  <implements namespace="http://www.poia.org/part_orders.wsdl"
      location="http://www.poia.org/part_orders.wsdl"/>
<operation name="findCustomerOrders">
    <documentation>Returns all the orders for a given customer.
    </documentation>
    <query>
      <SQL_query>select * from order_tab
                   where customer_name = :customer_name
      </SQL_query>
      <parameter name="customer_name" type="xsd:string"/>
    </query>
  </operation>
```

Figure 25. DADX with Query operation

A list of parameter elements that are uniquely named within this operation must define the input parameters. If you need more control over the mapping, then you can use a DAD file.

You can use the *Query* operation to use the XML Extender user-defined types (UDT) and user-defined functions (UDF). This operation allows you to query, extract, and update data from an XML column that contains XML documents. These XML documents require that you create a document type definition (DTD) that defines the type of the <XML_result> element. This element specifies the column name and the root element of the XML document contained in it.

The example in Figure 26 on page 76 shows a Query operation that uses the VARCHAR UDT declared by the <XML_result> element. The `retrieveOrders` operation retrieves all the XML order documents from the SALES_TAB table by using the UDF db2xml.varchar. You store the documents by using the XML Extender UDT XMLVARCHAR.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:dtd1="http://schemas.myco.com/sales/order.dtd">
  <documentation>
      Queries part orders at myco.com.
  </documentation>

<operation name="retrieveOrders">
    <documentation>
        Retrieves all the Order documents.
    </documentation>
    <query>
      <SQL_query>
        select db2xml.varchar(order) from sales_tab
      </SQL_query>
      <XML_result name="ORDER" element="dtd1:Order"/>
    </query>
  </operation>
</DADX>
```

Figure 26. Query operation with UDF and UDT

When you have XML documents in a column and you want the WSDL to refer to the type of this document, you can use the XML_result tag. In Figure 26 the example specifies that the ORDER column contains fragments of element *dtd1:Order*. The element <XML_result name = ″ORDER″ element = ″dtd1:Order″/> refers to the namespace declaration. XML Extender stores XML documents that have no namespaces and that are defined by DTDs. Web services use XML Schemas (XSD) instead of DTDs, and make use of namespaces. You associate a namespace with a DTD by making an entry in the namespace table. WORF adds the namespace when it retrieves an XML document and removes the namespace when it stores a document. WORF also automatically translates DTDs to XSD. The line, <XML_result name = ″ORDER″ element = ″dtd1:Order″/> defines column information in file *order.dtd*. The specific declaration that it refers to is in the following example:

```
<?xml encoding="US-ASCII"?>
<!ELEMENT Order (Customer, Part+)>
<!ATTLIST Order key CDATA #REQUIRED>
...
```

To point to the DTD, use a namespace table file, (NST) file. Refer to Figure 27 on page 77 as an example.

```
<?xml version="1.0"?>
  <namespaceTable xmlns="http://schemas.ibm.com/db2/dxx/nst">
    <mapping dtdid="c:\dxx\samples\dtd\getstart.dtd"
      namespace="http://schemas.ibm.com/db2/dxx/samples/dtd/getstart.dtd"
      location="/dxx/samples/dtd/getstart.dtd/XSD"/>
    <mapping dtdid="getstart.dtd"
      namespace="http://schemas.myco.com/sales/getstart.dtd"
      location="/getstart.dtd/XSD"/>
    <mapping dtdid="order.dtd"
      namespace="http://schemas.myco.com/sales/order.dtd"
      location="/order.dtd/XSD"/>
  </namespaceTable>
```

Figure 27. NST file

You must reference this file in the group.properties file.

Example 2: Update operation

The example in Figure 28 shows an operation that updates the electronic mail (e-mail) address of a customer for a given order. The update operation can contain SQL INSERT, DELETE, or UPDATE statements in the <SQL_update> element.

```
  <operation name="updateOrderEmail">
    <documentation>Updates the email address for an order.
    </documentation>
    <update>
      <SQL_update>update order_tab set customer_email = :email
                   where order_key = :key</SQL_update>
      <parameter name="key" type="xsd:int"/>
      <parameter name="email" type="xsd:string"/>
    </update>
  </operation>
 </DADX>
```

Figure 28. Update operation

Example 3: Call operation

If your stored procedure returns result sets, you must define these result sets in the result_set_metadata tag in the DADX file. This is to let WORF generate the WSDL and XML schema files (XSD) for this Web service operation. Figure 29 on page 78 shows the definition of a result set metadata that is referenced two times.

```
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<result_set_metadata name="employeeSalaryReport" rowName="employee">
  <column name="NAME" type="VARCHAR" nullable="true" />
  <column name="JOB" type="CHAR" nullable="true" />
  <column name="3" as="SALARY" type="DOUBLE" nullable="true" />
</result_set_metadata>
<operation name="twoResultSets">
<call>
<SQL_call>CALL TWO_RESULT_SETS (:salary, :sqlCode)
</SQL_call>
  <parameter name="salary" type="xsd:double" kind="in" />
  <parameter name="sqlCode" type="xsd:int" kind="out" />
  <result_set name="employees1" metadata="employeeSalaryReport" />
  <result_set name="employees2" metadata="employeeSalaryReport" />
  </call>
 </operation>
</DADX>
```

Figure 29. Definition of a result set metadata referenced two times

You can also call a stored procedure by using the format shown in Figure 30.

```
<operation name="callProc1">
    <documentation>Call the Proc1 stored procedure.
    </documentation>
    <call>
      <SQL_call>
        CALL Proc1 (:x, :y, :z)
      </SQL_call>
      <parameter name="x" type="xsd:string" kind="in"/>
      <parameter name="y" type="xsd:int" kind="in/out"/>
      <parameter name="z" element="dtd1:Order" kind="out"/>
    </call>
</operation>
```

Figure 30. DADX with alternate Call operation

Example 4: RetrieveXML operation

The DADX file in Figure 31 on page 79 implements one retrieveXML operation by using the stored procedure dxxGenXMLCLOB.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <documentation>
      mycompany part orders service.
    </documentation>
  <operation name="findByColorAndMinPrice">
    <documentation>Returns all the orders that have the specified color
        and at least the specified minimum price.</documentation>
    <retrieveXML>
      <DAD_ref>getstart_xcollection.dad</DAD_ref>
      <SQL_override>
       select o.order_key, customer_name, customer_email,
         p.part_key, color, quantity, price, tax, ship_id, date, mode
       from order_tab o, part_tab p,
         table(select substr(char(timestamp(generate_unique())),16)
           as ship_id, date, mode, part_key from ship_tab) s
       where p.order_key = o.order_key and s.part_key = p.part_key
         and color = :color and price >= :minprice
       order by order_key, part_key, ship_id
      </SQL_override>
      <parameter name="color" type="xsd:string"/>
      <parameter name="minprice" type="xsd:decimal"/>
    </retrieveXML>
  </operation>
</DADX>
```

Figure 31. DADX with retrieveXML operation

The operation in Figure 31 generates XML documents that are based on the mapping in the getstart_xcollection.dad file. The operation specifies an SQL override. The operation replaces the SQL statement defined in the DAD file and references two parameters in the override statement: `:color` and `:minprice`.

The DAD file for this example is in the appendix of *DB2 XML Extender Administration and Programming*.

Example 5: StoreXML operation

This example in Figure 32 on page 80 shows a DADX file that references a DAD by using RDB_node mapping, getstart_xcollection_rdb.dad.

```
<?xml version="1.0"?>
 <DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <documentation>
      mycompany part orders service.
    </documentation>

  <implements namespace="http://www.poia.org/part_orders.wsdl"
      location="http://www.poia.org/part_orders.wsdl"/>

   <operation name="storeOrder">
     <documentation>Stores an automotive part order.
     </documentation>
     <storeXML>
       <DAD_ref>getstart_xcollection_rdb.dad</DAD_ref>
     </storeXML>
   </operation>
 </DADX>
```

Figure 32. DADX with StoreXML operation

The storeXML operation is implemented by the dxxInsertXML stored procedure if a <collection_name> element is used instead of a <DAD_ref> element. It performs the same operations as the dxxShredXML procedure, but uses the name of an XML collection instead of a DAD file.

Web service provider operations used with DADX files

DADX files support three kinds of Web service operations: non-dynamic SQL operations, dynamic SQL operations, and XML collection operations.

SQL-based querying is the ability to send SQL statements, including stored procedure calls, to DB2® and to return results with a default tagging. Your application returns the data by using only a simple mapping of SQL data types, using column names as elements.

SQL operations: non-dynamic
The SQL operations can be non-dynamic. Non-dynamic operations are those that are predefined within the DADX file. There are three elements that make up the predefined SQL operations type:

<query>
Queries the database

<update>
Inserts into a database, deletes from a database, or updates a database

<call> Calls stored procedures that can return 0 or more result sets

SQL operations: dynamic
The SQL operations can be dynamic operations, depending on the content of the DADX file. Dynamic operations are those that are generated in a SOAP message with no predefined SQL operations. The following elements are dynamic operations:

<getTables>
Retrieves a description of available tables.

<getColumns>
Retrieves a description of columns.

<executeQuery>
: Issues a single SQL statement.

<executeUpdate>
: Issues a single INSERT, UPDATE, DELETE.

<executeCall>
: Calls a single stored procedure.

<execute>
: Issues a single SQL statement.

XML collection operations (requires DB2 XML Extender)

These storage and retrieval operations help you to map XML document structures to DB2 Universal Database™ tables. You can either compose XML documents from existing DB2 data, or decompose (storing untagged elements or attribute content) XML documents into DB2 data. This method is useful for data interchange applications, particularly when the application frequently updates the contents of XML documents.

There are two elements that make up the XML collection operation type:

<retrieveXML>
: Generates XML documents

<storeXML>
: Stores XML documents

The DAD file provides fine-grained control over the mapping of XML documents to a DB2 database for both storage and retrieval.

XML schema for the DADX file

The following XML schema, dadx.xsd, describes the DADX. All of the WORF schema files are in the dxxworf.zip file, which is part of the sqllib\samples\ webservices directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://schemas.ibm.com/db2/dxx/dadx"
  xmlns:dadx="http://schemas.ibm.com/db2/dxx/dadx"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" xml:lang="en">
 <annotation>
  <documentation>
      A Document Accession Definition Extension (DADX)
       document defines a Web Service
       that is implemented by operations that
       access a relational database and that optionally use
       stored procedures, types and functions provided
       by the DB2 XML Extender.
    </documentation>
 </annotation>
 <element name="DADX">
  <annotation>
   <documentation>
        Defines a Web Service.
        The Web Service is described by an optional
        WSDL documentation element.
        The Web Service may implement a set of WSDL
        bindings defined elsewhere.
        The Web Service consists of one or more
        uniquely named operations.
      </documentation>
  </annotation>

<complexType>
   <sequence>
    <element ref="dadx:documentation"
              minOccurs="0" maxOccurs="unbounded"/>
    <choice>
     <element ref="dadx:DQS"
                  minOccurs="0"/>
     <sequence>
      <element ref="dadx:implements" minOccurs="0"/>
      <element ref="dadx:result_set_metadata"
                     minOccurs="0" maxOccurs="unbounded"/>
      <element ref="dadx:operation"
                     maxOccurs="unbounded"/>
     </sequence>
    </choice>
   </sequence>
  </complexType>
  <key name="result_set_metadataNames">
   <selector xpath="dadx:result_set_metadata"/>
   <field xpath="@name"/>
  </key>
  <keyref name="resultSetMetatdata"
         refer="dadx:result_set_metadataNames">
   <selector xpath="dadx:operation/dadx:call/dadx:result_set"/>
   <field xpath="@metadata"/>
  </keyref>
  <unique name="operationNames">
   <selector xpath="dadx:operation"/>
   <field xpath="@name"/>
  </unique>
 </element>

<element name="DQS">
  <annotation>
   <documentation>
        Defines the DQS tag.
      </documentation>
  </annotation>
  <complexType/>
 </element>

 <element name="documentation">
  <annotation>
   <documentation>
        Defines WSDL documentation for the Web service or an operation.
      </documentation>
```

Web services encoding algorithm

This is an algorithm that encodes and decodes the password within the group.properties file.

1. Convert the clear text information into a sequence of data bytes by using UTF-8 character encoding. Let L be the length of the data byte sequence.
2. Convert the data bytes into a further sequence of data bytes, data8, that is 8 times longer. You compute byte k of data8 as follows. Let $k = j * L + i$ where $0 <= i < L$ and $0 <= j < 8$. First mask bit *j* of data byte *i*. Second, *exclusive or* this with *k*. This step distributes the bits of each data byte throughout the length of the data8 sequence.
3. Apply the standard base64 encoding algorithm to data8. This step renders the bytes as printable characters and also increases the length by a factor of four-thirds (4/3).
4. Prefix the encoded string with *encoded*: to denote that it has been encoded.

Web services command reference

You can use Web services provider commands to encode passwords, validate a DADX file and validate a DAD file.

Encoder

Encodes or decodes a password in the group.properties file.

- Example of encoding (assumes that worf.jar is listed in the CLASSPATH):

```
 java com.ibm.etools.webservice.rt.util.Encoder
     -in group.properties -out group.properties
```

- Example of decoding (assumes that worf.jar is listed in the CLASSPATH):

```
java com.ibm.etools.webservice.rt.util.Encoder
     -action decode -in group.properties -out group.properties
```

Check_install

Validates a DADX file.

- Example:

```
java com.ibm.etools.webservice.util.Check_install
   [-srv] [-schdir pathToSchemasDir]
   [-sch schemaLocations]  [-out outputFile] fileToCheck
```

dadchecker

Validates a DAD file.

- Example:

```
java dadchecker.Check_dad_xml  [-dad | -xml] [-all]
    [-dup dupName] [-enc encoding][-dtd dtdPath]
    [-xstruct xmlDocument] [-out outputFile] fileToCheck
```

Chapter 3. Dynamic database queries that use the Web services provider

With dynamic query services you can dynamically build and submit queries at run time that select, insert, update and delete application data, and call stored procedures rather than run queries that are predefined at deployment time.

A Web application can use the Web services interface to access a database and extract information about the tables and columns that are available. Then, the application can query the tables and modify the data in the database through Web services. The Web application can also perform data definition language actions on the database, such as creating tables. By using the dynamic query services of the Web services provider, Web applications can be more flexible.

WORF can generate two styles of Web services description language files (WSDL) from the DADX files that contain a dynamic query service tag (<DQS/>):

- A WSDL file that uses the document-oriented information style
- A WSDL file that uses the procedure-oriented information style (RPC)

The style that is generated is defined on a group level and depends on the existence of `useDocumentStyle=true` in the group.properties file. For more information about the Web services description language information styles, look in the Web services description language specifications on your browser. The WSDL file contains service, port, and definition information. Dynamic query tags in the DADX files do not affect static DADX functions.

Consider using the dynamic query service when you do not know the query search criteria until you run your application.

The dynamic query component of the Web services provider supports Web service operations that are generally defined by the following categories:

Obtain metadata
: You can retrieve the tables that exist in a database and the column information for those tables.

Execute DDL
: You can issue a CREATE TABLE statement.

Execute DML
: You can issue SELECT, INSERT, UPDATE and DELETE statements, and the CALL statement to run stored procedures.

The server administrator controls access to a specific database by defining a group with specific user ID and password settings in the group.properties file. The administrator can also create a separate WORF instance to handle access to a database.

Configuring and running dynamic database queries as part of Web services provider

With dynamic query services, you can build, execute stored procedures and submit database queries at run time that access a previously deployed Web service. You no longer need to define all of your database queries in your Document Access Definition Extension (DADX) file.

Before you begin

- Ensure that a group.properties file exists for the group in which you want to run dynamic Web queries.
- The Web application must establish a connection to the target database for each Web service operation that is defined in the Web services description language (WSDL) document.
- You must ensure that the XML schema description file, db2WebRowSet.xsd is included in the context root of your Web application, unless you define an import definition in the WSDL. The db2WebRowSet.xsd file is included in the dxxworf.zip file.

Restrictions

- When your application uses the dynamic query services of the Web services provider, the application cannot use cursors or perform any operation that assumes a state on the server. You must obtain your results in a single query.
- The XML tag (<DQS/>) that identifies a dynamic query service operation cannot coexist with any DADX-specific Web service definitions within the same file.

About this task

You can run dynamic queries at the group level, or within the scope of the group directory, based on the information in the group.properties file.

Procedure

To prepare your Web services environment to run dynamic queries on a DB2 with Web services provider:

1. Create a DADX file that includes the XML tag <DQS/>. This tag enables a group to perform dynamic queries. No other tag is needed in the DADX file.
2. Save the file in the directory of the group for which you will run dynamic queries.
3. Using the WSDL, develop a client for the application. The client must contain at least the following information:
   - A group name
   - The name of the DADX file, such as mydqs.dadx
   - A Web service operation, such as `getTables`
4. Modify the client to issue one of the accepted DQS operations, such as the getTables operation.
5. Run the client that issues the `getTables` operation.

The result of the query is metadata that describes the rows and columns of the table, and the data that is contained in the tables. The SQL statements run in autocommit mode. The client can also call a dynamic query service in other groups. The only information that needs to change is the endpoint URL. However, clients are only compatible for either an RPC style WSDL or a document style WSDL. You cannot use a dynamic query services client that is defined by an RPC style WSDL for a group that uses a document style WSDL.

Dynamic query services-example queries

This topic shows how to use dynamic query services.

Example 1: the DADX file

In the following example, the DADX file is named mydqs.dadx. The file mydqs.dadx is in the directory of the group for which you will execute dynamic queries.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx">
    <documentation>
        This is optional documentation about DQS
    </documentation>
      <DQS/>
</DADX>
```

Example 2: Running the dynamic query from a browser

The following example is a simple dynamic query that you can run from a browser. You can also include this statement in an application. The required information for a dynamic query in Web services provider is in **bold** print. The Web service operation in this example is `executeQuery`. The parameter associated with the operation is `queryInput`. The statement fetches all rows of column `lastname` from table `employee`:

```
http://localhost:9080/services/<group_name>
  /somefile.dadx/executeQuery?queryInputParameter
     =select%20lastname%20from%20employee
```

The example issues a GET binding request rather than a complete SOAP envelope. . The following output is from the executeQuery operation and it is defined by the db2WebRowSet schema definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:executeQueryResponse
    xmlns:ns1="http://schemas.ibm.com/db2/dqs">
  <queryOutputParameter>
   <db2WebRowSet
       xmlns="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<metadata>
 <column-count>14</column-count>
 <column-definition>
  <column-index>1</column-index>
  <nullable>0</nullable>
  <column-name>EMPNO</column-name>
  <column-precision>6</column-precision>
  <column-scale>0</column-scale>
  <column-type>CHAR</column-type>
  <column-type-name>CHAR</column-type-name>
  <xml-type>string</xml-type>
 </column-definition>
 <column-definition>
  <column-index>2</column-index>
  <nullable>0</nullable>
  <column-name>FIRSTNME</column-name>
  <column-precision>12</column-precision>
  <column-scale>0</column-scale>
  <column-type>VARCHAR</column-type>
  <column-type-name>VARCHAR</column-type-name>
  <xml-type>string</xml-type>
 </column-definition>
...
 <column-definition>
  <column-index>14</column-index>
  <nullable>1</nullable>
  <column-name>COMM</column-name>
  <column-precision>9</column-precision>
  <column-scale>2</column-scale>
  <column-type>DECIMAL</column-type>
  <column-type-name>DECIMAL</column-type-name>
  <xml-type>decimal</xml-type>
 </column-definition>
...
 <column-definition>
  <column-index>14</column-index>
  <nullable>1</nullable>
  <column-name>COMM</column-name>
  <column-precision>9</column-precision>
  <column-scale>2</column-scale>
  <column-type>DECIMAL</column-type>
  <column-type-name>DECIMAL</column-type-name>
  <xml-type>decimal</xml-type>
 </column-definition>
</metadata>
</data>
</db2WebRowSet>
</queryOutputParameter>
  </ns1:executeQueryResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Example 3: Importing db2WebRowSet.xsd

When the group contains a dynamic query services DADX, the db2WebRowSet.xsd file must be accessible to Web services consumers. To ensure the location of the db2WebRowSet.xsd file, the group.imports file defines the necessary schema locations. The following is an example of a group.imports file to import db2WebRowSet.xsd. This example assumes that you do not have file db2WebRowSet.xsd in your local groups directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<imports
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:import namespace="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
      schemaLocation="http://myServer.myCo.com/schemas/misc/ibm/db2WRS.xsd"/>
</imports>
```

If no group.imports file exists, then WORF generates the default import elements in the WSDL only for the dynamic query services. In this case, WORF assumes that the db2WebRowSets.xsd file is in the following location:

```
http://<server>:<port>/<contextRoot>/db2WebRowSet.xsd
```

Example 4: getTables

The following is an example of the getTables operation:

```
<?xml version="1.0" encoding="UTF-8"?>
 <SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns0:getTables
      xmlns:ns0="http://schemas.ibm.com/db2/dqs">
     <tablesInputParameter>
       <tablesInputData>
        <schemaPattern>MSCHENK</schemaPattern>
        <tableNamePattern>EMPLOYEE</tableNamePattern>
       </tablesInputData>
      </tablesInputParameter>
    </ns0:getTables>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 5: getColumns

The following is example of the getColumns operation:

```
<?xml version="1.0" encoding="UTF-8"?>
 <SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Body>
    <ns0:getColumns
      xmlns:ns0="http://schemas.ibm.com/db2/dqs">
      <columnsInputParameter>
        <columnsInputData>
          <schemaPattern>MSCHENK</schemaPattern>
          <tableNamePattern>EMPLOYEE</tableNamePattern>
          <columnNamePattern>EMPNO</columnNamePattern>
        </columnsInputData>
      </columnsInputParameter>
    </ns0:getColumns>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: executeQuery

The following example query fetches all rows from table `employee` and specifies several parameters:

```
<?xml version="1.0" encoding="UTF-8"?>
 <SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
     <ns0:executeQuery
       xmlns:ns0="http://schemas.ibm.com/db2/dqs">
      <queryInputParameter>
       select * from employee
      </queryInputParameter>
      <extendedInputParameter>
      <properties>
         <loginInfo>
           <userid>userid</userid>
           <password>some_password</password>
         </loginInfo>
         <readOnly>true</readOnly>
         <isolationLevel>READ_UNCOMMITTED</isolationLevel>
         <escapeProcessing>true</escapeProcessing>
         <startAtRow>4</startAtRow>
         <fetchSize>80</fetchSize>
         <maxFieldSize>20</maxFieldSize>
         <maxRows>100</maxRows>
         <queryTimeout>2000</queryTimeout>
      </properties>
     </extendedInputParameter>
   </ns0:executeQuery>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 7: executeUpdate

The following example shows a dynamic query services update statement:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
   <ns0:executeUpdate
     xmlns:ns0="http://schemas.ibm.com/db2/dqs">
     <queryInputParameter>
        update bo_events set OBJECTEVENTID=&'testestest&'
     </queryInputParameter>
     <extendedInputParameter>
       <properties/>
     </extendedInputParameter>
   </ns0:executeUpdate>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following example shows the response document that is returned:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:executeUpdateResponse
   xmlns:ns1="http://schemas.ibm.com/db2/dqs">
    <updateOutputParameter xsi:type="xsd:int">
        1
    </updateOutputParameter>
   </ns1:executeUpdateResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Example 8: executeCall

The example request calls the `multipleResultSets` stored procedure:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns0:executeCall
      xmlns:ns0="http://schemas.ibm.com/db2/dqs">
    <callInputParameter>
      <callInputData>
          <spName>
             multipleResultSets
          </spName>
          <parameters>
```

```
            <parameter>
              <inParam>
                 <kind>IN</kind>
                 <type>string</type>
                 <value>000130</value>
              </inParam>
            </parameter>
            <parameter>
              <inParam>
                <kind>INOUT</kind>
                <type>string</type>
                <value>000130</value>
              </inParam>
             </parameter>
             <parameter>
               <outParam>
                 <kind>OUT</kind>
                 <type>string</type>
               </outParam>
             </parameter>
          </parameters>
       </callInputData>
    </callInputParameter>
   <extendedInputParameter>
    <properties/>
   </extendedInputParameter>
  </ns0:executeCall>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following example shows the sample output:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:executeCallResponse
     xmlns:ns1="http://schemas.ibm.com/db2/dqs">
  <callOutputParameter>
  <dqs:callOutputData
     xmlns:dqs="http://schemas.ibm.com/db2/dqs/types/soap">
   <dqs:outputResultSequences>
   <db2WebRowSet
     xmlns="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <metadata>
      <column-count>5</column-count>
      <column-definition>
        <column-index>1</column-index>
        <nullable>0</nullable>
        <column-name>DEPTNO</column-name>
        <column-precision>3</column-precision>
        <column-scale>0</column-scale>
        <column-type>CHAR</column-type>
        <column-type-name>CHAR</column-type-name>
        <xml-type>string</xml-type>
      </column-definition>
        ...
      <column-definition>
        <column-index>5</column-index>
        <nullable>1</nullable>
        <column-name>LOCATION</column-name>
        <column-precision>16</column-precision>
        <column-scale>0</column-scale>
        <column-type>CHAR</column-type>
        <column-type-name>CHAR</column-type-name>
        <xml-type>string</xml-type>
 </column-definition>
</metadata>
<data>
 <row>
  <column>A00</column>
  <column>
     SPIFFY COMPUTER SERVICE DIV.
  </column>
  <column>000010</column>
  <column>A00</column>
  <column xsi:nil="true"/>
 </row>
...
 <row>
   ...
 </row>
</data>
</db2WebRowSet>
</dqs:outputResultSequences>
   <dqs:outputParameterSequences>
    <dqs:callOutputParam>
     <position>2</position>
     <type>string</type>
     <value>xxxxxx</value>
    </dqs:callOutputParam>
    <dqs:callOutputParam>
     <position>3</position>
     <type>string</type>
     <value>This is the value of name3</value>
    </dqs:callOutputParam>
    </dqs:outputParameterSequences>
    </dqs:callOutputData>
   </callOutputParameter>
  </ns1:executeCallResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Example 9: execute

The following example creates a table with one column:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <ns0:execute
    xmlns:ns0="http://schemas.ibm.com/db2/dqs">
     <queryInputParameter>
       create table temptable(in varchar(500))
     </queryInputParameter>
     <extendedInputParameter>
       <properties/>
     </extendedInputParameter>
  </ns0:execute>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following is the output from the execute operation:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
   <ns1:executeResponse
     xmlns:ns1="http://schemas.ibm.com/db2/dqs">
    <executeOutputParameter>
    <dqs:executeOutputData
       xmlns:dqs="http://schemas.ibm.com/db2/dqs/types/soap">
      <resultsPresent>false</resultsPresent>
     <dqs:outputResultSequences>
     </dqs:outputResultSequences>
    </dqs:executeOutputData>
    </executeOutputParameter>
  </ns1:executeResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Dynamic query service operations in the Web services provider

This topic describes the dynamic query operations that are supported in the DB2 Web services provider.

The following tables describe the dynamic query operations that are supported in the DB2 Web services provider.

Table 3. Operations for metadata retrieval

Web service operation	Description
getTables **tablesInputParameter** input; type = tablesInputData (see Table 7 on page 96) **tablesOutputParameter** output; type = "db2WebRowSet" on page 98	Retrieves a description of the tables in the specified catalog and schema, such as the name of the catalog, the name of the schema, and the name of the table. If you use schema as an input parameter, the Java database connectivity might require case sensitivity for schema.
getColumns **columnsInputParameter** input; type = columnsInputData (see Table 8 on page 97) **columnsOutputParameter** output; type = "db2WebRowSet" on page 98	Retrieves a description of the columns in the specified catalog, schema, and table. If you use schema as an input parameter, the Java database connectivity might require case sensitivity for schema.

Table 4. Operations to run queries and stored procedures

Operations	Description
executeQuery **queryInputParameter** required input; type = string **extendedInputParameter** required input; type = properties (see Table 5 on page 94) **queryOutputParameter** output; type = "db2WebRowSet" on page 98	Issues a single SQL SELECT statement on the database server and returns a single result set.

Table 4. Operations to run queries and stored procedures (continued)

Operations	Description
executeUpdate **queryInputParameter** required input; type = string **extendedInputParameter** required input; type = properties (see Table 5) **updateOutputParameter** output; type = int	Issues a single INSERT, UPDATE, DELETE statement on the database server and returns a completion code.
executeCall **callInputParameter** input; type = callInputData **extendedInputParameter** input; type = properties (see Table 5) **callOutputParameter** output; type = callOutputData	Calls a single stored procedure on the database server and returns a set of output parameters and a sequence of result sets.
execute **queryInputParameter** required input; type = string **extendedInputParameter** required input; type = properties (see Table 5) **executeOutputParameter** output; type = executeOutputData	Issues a single SQL statement on the database server and returns a completion code and a sequence of result sets.

You can use the optional parameters that are listed in Table 5 with the operations that are listed in Table 4 on page 93.

Table 5. Input data types for the extended parameters

Properties type	Description
loginInfo • userid • password	The loginInfo includes the user ID that is passed to the database for access control. It also includes the password that is associated with the user ID that is passed to the database for access control. These properties have a type of string. If you specify a user ID, then you must specify a password.
readOnly	Allows the Web application to specify that it will use the database for read-only purposes. This is a binary type and can be either true or false.
escapeProcessing	Allows the Web application to control escape processing on the query string. If escape scanning is enabled (true), the driver performs escape substitution before it sends the SQL to the database. This is a binary type and can be either true or false. The default value is true.
fetchSize	Specifies the number of rows to be fetched back to the Web application on any given fetch operation. This is type integer. The default value is 0.
maxFieldSize	Sets the limit for the maximum number of bytes in a column to the specified number of bytes. The value is the maximum number of bytes that can be returned for any column value. The is type integer.

Table 5. Input data types for the extended parameters (continued)

Properties type	**Description**
maxRows	Specifies the maximum number of rows to fetch back to the Web application. This is type integer. If the maxRows parameter is not specified, then a maximum of 1000 rows can be returned.
startAtRow	Allows the Web application to skip a specified number of rows in the result set. This is type integer.
queryTimeout	Allows the Web application to specify a timeout value for the query. Sets the number of seconds that the driver waits for a statement object to run to the given number of seconds. If the limit is exceeded, an exception occurs. A value of 0 seconds indicates that the driver can wait an unlimited number of seconds.
isolationLevel	Allows the Web application to control the isolation level of the query. • READ_UNCOMMITTED • READ_COMMITTED • REPEATABLE_READ • SERIALIZABLE • NONE

Table 6. Input data types for the callInputParameter

callInputData type	**Description**
spName type: string	The name of the stored procedure to invoke. This parameter is mandatory.
schema type: string	The schema of the stored procedure. This parameter is optional. If the parameter is not supplied, the value is the current schema.

Table 6. Input data types for the callInputParameter (continued)

callInputData type	Description
parameters type: sequence of parameters, each one consisting of either an inParam or an outParam **inParam** type defined as: • **kind**: either 'IN' or 'INOUT' • **type**: the type of the parameter (such as int, or string) • **value**: the value of the parameter **outParam** type defined as: • **kind**: either 'IN' or 'INOUT' • **type**: the type of the parameter	Stored procedures can have three kinds of parameters: IN, OUT, and INOUT. This parameter type is an extensible type. It allows any number of any combination of the inParam and outParam types. The Web application must know if the stored procedure that it plans to invoke needs any parameters. If it needs parameters, it needs to know how many parameters, and their type. If the stored procedure takes one of the unsupported data types as a stored procedure parameter, then this stored procedure cannot be executed through WORF. WORF accepts several XML types for the stored procedure parameters. The parameters correspond to the built-in SQL data types. Table 2 on page 72 describes the supported types. An input parameter can be set to NULL by using one of the following values: **absent** The <value/> tag for the input parameter is not provided. **nil = true** The tag is marked with the attribute nil, which is set to true, such as <value xsi:nil="true"/> The order of the input parameter must be the same as the order expected by the stored procedure.

Table 7. Input data types for the tablesInputData type

tablesInputData type	Description
catalogPattern type = "string" **schemaPattern** type = "string" **tableNamePattern** type = "string"	Each of the pattern values is optional. If the value is not specified, the value defaults to the blank value. The description and behavior of each is specified in JDBC. Use the getTables Web service operation to return the list of tables that are satisfy the catalogPattern, schemaPattern, and tableNamePattern that are specified.

Example (note that such things as the namespace definitions are not shown here for simplicity):

```
<tablesInputData>
 <catalogPattern></catalogPattern>
 <schemaPattern>userSchema
 </schemaPattern>
 <tableNamePattern>EMPLOYEE
 </tableNamePattern>
</tablesInputData>
```

Table 8. Input data types for columnsInputData types

columnsInputData type	Description
catalogPattern type = "string" **schemaPattern** type = "string" **tableNamePattern** type = "string" **columnNamePattern** type = "string"	Each of the pattern values is optional. If the value is not specified, the value defaults to the blank value. The description and behavior of each is specified in JDBC. Use the getColumns Web service operation to receive a list of columns that satisfy the catalog string pattern, schemaPattern, table name, and columnNamePattern that is specified.

Example (note that such things as the namespace definitions are not shown here for simplicity):

```
<columnsInputData>
 <catalogPattern></catalogPattern>
 <schemaPattern>userSchema
 </schemaPattern>
 <tableNamePattern>EMPLOYEE
 </tableNamePattern >
 <columnNamePattern>LASTNAME
 </columnNamePattern>
</columnsInputData>
```

Table 9. Output data types for the callOutputData types

callOutputData type

outputResultSequences
: contains a sequence of all result sets returned by the stored procedure as type db2WebRowSet

outputParameterSequences:
: contains a sequence of callOutputParam (parameters that were returned from the stored procedure that can be either kind=INOUT or kind=OUT)

callOutputParam
: returned Parameter: contains

- <position>

  type: int - the position of the parameter in the stored procedure parameter list
- <type>

  type: string - the XML data type (see callInputData for type information)
- <value>

  type: any - the value of the parameter

If an output parameter is NULL the absent method is used.

The result contains

```
<value xsi:nil="true"/>
```

Table 9. Output data types for the callOutputData types (continued)

callOutputData type
Example (note that such things as the namespace definitions are not shown here for simplicity):

```
<callOutputParameter>
   <dqs:callOutputData
      xmlns:dqs="http://schemas.ibm.com/db2/dqs/types/soap">
      <dqs:outputResultSequences>
         <db2WebRowSet
           xmlns="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <metadata>
             ....
         </db2WebRowSet>
     </dqs:outputResultSequences>
     <dqs:outputParameterSequences>
            <dqs:callOutputParam>
                <position>1</position>
                <type>short</type>
                <value>123</value>
            </dqs:callOutputParam>
            <dqs:callOutputParam>
                <position>2</position>
                <type>int</type>
                <value xsi:nil="true" />
            </dqs:callOutputParam>
         </dqs:outputParameterSequences>
      </dqs:callOutputData>
</callOutputParameter>
```

Table 10. Output data types for the executeOutputData types

executeOutputData type	**Description**
resultsPresent type = ″boolean″ **outputResultSequences** 0 or more occurrences of db2WebRowSet	If the **execute** Web service operation is invoked with a query string that returns result sets, the boolean indicates that this, and outputResultSequences will each contain one of those result sets.

Example (note that such things as the namespace definitions are not shown here for simplicity):

```
<executeOutputParameter>
   <dqs:executeOutputData
      xmlns:dqs="http://schemas.ibm.com/db2/dqs/types/soap">
      <resultsPresent>true</resultsPresent>
      <dqs:outputResultSequences>
         <db2WebRowSet
           xmlns="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <metadata>
             ....
         </db2WebRowSet>
      </dqs:outputResultSequences>
   </dqs:executeOutputData>
</executeOutputParameter>
```

db2WebRowSet

The dynamic query service type, db2WebRowSet, describes a generic way for generating an XML document from an SQL result set.

The schema document, `db2WebRowSet.xsd` does not contain any metadata information about a particular result set. It contains generic metadata information about result set metadata. The actual result set metadata and the result set data is in the XML instance document. An instance document contains a metadata section and a data section.

Metadata section

The metadata section contains metadata information about all of the columns that are in the result. The first element is a column count element. It contains the number of columns in the result set. Then there is a column definition element for every column. The column definition contains the following metadata information:

Table 11. Column definition metadata

Element name	Description
<column-index>	The position of the column in the result set, starting with 1.
<nullable>	If the column can be NULL, then the value is 1. If the column cannot be NULL, then the value is 0.
<column-name>	The name of the column.
<column-precision>	The description of this element depends on the SQL data type. For example, if the SQL data type is a character, then the column-precision is length. If the SQL data type is a decimal, then the column- precision is precision.
<column-scale>	The column-scale is a decimal data type.
<column-type>	The column-type corresponds to the Java database connectivity type, such as BINARY, VARBINARY, CHAR, and VARCHAR.
<column-type-name>	The DB2 data type name, such as CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, CHAR, and VARCHAR.
<xml-type>	The XML data type, such as base64binary, int, string, and dateTime.

Data section

The data section contains the actual data. Each row is mapped to a row element. A row element contains as many column elements as there are columns in the result set, and ordered by the column index. The row element contains the actual data as an XML data type.

Table 12. Data type mapping conventions

DB2 data type <column-type-name>	JDBC data type <column-type>	XML data type <xml-type>
BLOB	BLOB	base64Binary
CLOB	CLOB	string
LONGVARCHAR	LONGVARCHAR	string
VARCHAR	VARCHAR	string
CHAR	CHAR	string
CHAR FOR BIT DATA	BINARY	base64Binary

Table 12. Data type mapping conventions (continued)

DB2 data type <column-type-name>	**JDBC data type <column-type>**	**XML data type <xml-type>**
VARCHAR FOR BIT DATA	VARBINARY	base64Binary
LONGVARCHAR FOR BIT DATA	LONGVARBINARY	base64Binary
DATE	DATE	date
TIME	TIME	time
TIMESTAMP	TIMESTAMP	dateTime
-	BOOLEAN	boolean
-	BIT	boolean
TINYINT	TINYINT	int
SMALLINT	SMALLINT	int
INTEGER	INTEGER	int
BIGINT	BIGINT	int
DOUBLE	DOUBLE	double
FLOAT	FLOAT	double
REAL	REAL	float
DECIMAL	DECIMAL	decimal
NUMERIC	NUMERIC	decimal
-	ARRAY	anyType
DISTINCT	DISTINCT	string
-	JAVA_OBJECT	string
-	NULL	string
-	OTHER	string
-	STRUCT	string
-	REF	string
	other number of the type	string

The following is the db2WebRowSet.xsd file. The default location of this file is the <contextRoot> directory.

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema targetNamespace="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:db2wrs="http://schemas.ibm.com/db2/dqs/db2WebRowSet"
  elementFormDefault="qualified">
   <xs:element name="db2WebRowSet">
  <xs:complexType>
   <xs:sequence>
      <xs:element ref="db2wrs:metadata"/>
    <xs:element name="data">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="row"
                              minOccurs="0"
                              maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element ref="db2wrs:column"
                                    minOccurs="0"
                                    maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
<xs:element name="column"
               type="xs:anyType"
               nillable="true"/>
 <xs:element name="metadata">
  <xs:complexType>
   <xs:sequence>
    <xs:element
                name="column-count"
                type="xs:string" />
    <xs:choice>
     <xs:element name="column-definition"
                      minOccurs="0"
                      maxOccurs="unbounded">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="column-index"
                             type="xs:string" />
        <xs:element name="nullable"
                             type="xs:string" />
        <xs:element name="column-name"
                             type="xs:string" />
        <xs:element name="column-precision"
                             type="xs:string" />
        <xs:element name="column-scale"
                             type="xs:string" />
        <xs:element name="column-type"
                             type="xs:string" />
        <xs:element name="column-type-name"
                             type="xs:string" />
        <xs:element name="xml-type"
                             type="xs:string"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 34. db2WebRowSet.xsd

Chapter 4. Document type definition repository table

When you enable a database for XML operations, the DB2 XML Extender creates a document type definition (DTD) repository table named DTD_REF. The schema for the DTD_REF table is DB2XML.

Validating the DTD

Each DTD in the DTD_REF table has a unique identifier, which is named in the DTDID. The DTDID can be a string identifier or it can be the path that specifies the location of the DTD on the local system. The DTDID must match the value that is specified in the DAD file for the <DTDID> element.

Use a DTD to validate the XML data in an XML column or an XML collection. Insert a DTD that you want to validate into the DTD_REF table by issuing the following statement from the command line:

```
INSERT INTO db2xml.dtd_ref VALUES
  ('resume.dtd',db2xml.XMLClobFromFile
      ('%PATH_DEMO%\resume.dtd'),0,
      'user1','user1','user1')
```

The INSERT statement uses the following values:

`db2xml.dtd_ref`: the DB2 XML Extender table that stores DTDs.

`resume.dtd`: a DTD that is validated.

`XMLClobFromFile`:

Verifying the DTD registration

Verify that the DTD is registered correctly by issuing the following statement:

```
SELECT dtdid, content FROM db2xml.dtd_ref;
```

The value of content in this statement is the content of the DTD.

DTD definitions for XML Extender

The DB2 XML Extender provides sample DTD definitions for various platforms.

Ensure that the database administrator has set up any databases or subsystems required for the application, and enables them for use by DB2 XML Extender (if you use XML Extender). The following table lists the default locations that the XML Extender samples reference.

Table 13. XML Extender samples reference the following document type definitions (DTDs)

Platform	Default location of DTDs

DB2 UDB Version 7.2 FixPak 7 or later Windows	`c:\dxx\samples\dtd\ getstart.dtd`
	 `c:\dxx\dtd\dad.dtd`

Table 13. *XML Extender samples reference the following document type definitions (DTDs) (continued)*

Platform	Default location of DTDs
DB2 UDB Version 8 Windows	`c:\<DB2 installed location>\samples\ db2xml\dtd\getstart.dtd` `c:\<DB2 installed location>\samples\ db2xml\dtd\dad.dtd`
DB2 UDB Version 8 on Solaris Operating Environment	`/opt/IBMdb2/V8.1/samples/ db2xml/dtd/dad.dtd`
DB2 UDB Version 8 on AIX	`/usr/opt/db2_08_01/ samples/db2xml/dtd/dad.dtd`
DB2 UDB Version 8 on Linux	`/usr/IBMdb2/V8.1/samples/ db2xml/dtd/dad.dtd`
DB2 UDB Version 7 on OS/390 and z/OS or DB2 UDB Version 8 on OS/390 and z/OS	`/u/USER/dxx/dtd/dad.dtd`

The following is a list of some of the files that reference dad.dtd:

department.dad
department2.dad
departmentStd.dad
order.dad
order-public.dad
getstart.xml
order-10.xml
sales_db.nst

XML collection operations

You can generate or store XML documents with the <retrieveXML> or <storeXML> operations. These operations call XML Extender stored procedures and require a DAD file or an XML collection reference.

The retrieveXML or storeXML stored procedures generate or store XML documents by using the mapping in a DAD file, or by referring to an enabled XML collection. See *DB2 XML Extender Administration and Programming* to learn how to create a DAD file.

The following example shows a more complex DADX file that generates an XML document from a DAD file. It references a stored procedure by using the <RetrieveXML> element. The <DAD_ref> element specifies the name of a DAD file.

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <documentation>
   Provides queries for part order information at myco.com.
   See
   <xhtml:a href="../documentation/PartOrders.html"
      target="_top">PartOrders.html
  </xhtml:a>
   for more information.
  </documentation>
  <operation name="findAll">
    <documentation>
      Returns all the orders with their complete details.
    </documentation>
    <retrieveXML>
       <DAD_ref>getstart_xcollection.dad</DAD_ref>
       <no_override/>
    </retrieveXML>
 </operation>
</DADX>
```

Figure 35. DADX file that generates an XML document

The Web service generated from this DADX file calls the dxxGenXML stored procedure and generates XML documents. The stored procedure refers to the `getstart_xcollection.dad` file to determine which tables to use when generating the XML documents, and the XML document structure.

Converting a document type definition to an XML schema

Web services description language uses XML schemas (XSD files) to define document structure. XML Extender uses document type definition files (DTDs) to define document structure. Web services object runtime framework (WORF) automatically creates an XML Schema (XSD) file.

Purpose

To use your XML Extender DTD files with WSDL, you must convert the DTD files to XML schemas. You must add an entry to the namespace table (NST file) to define the namespace associated with a DTD. This also enables conversion of the DTD to XSD. Request an XSD file by using the following uniform resource locator (URL) syntax:

http://*host*/*path*/*dtd_file*.dtd/XSD

Example request for an XSD file

In this example, the order.dtd file must be in WEB-INF\groups\dxx_sample.

```
http://host_name:port/services/sample/order.dtd/XSD
```

DTD_REF table

WORF and the XML Extender locate DTDs through their document type definition identifier (DTDID). The DTDID is either a file name or the key value in the DTD_REF table of your database. XML Extender creates the DTD_REF table when you enable your database. The best practice is to store DTDs in the DTD_REF table since file locations might change when you move your Web application to another machine.

How to insert DTDs in the DTD_REF table

The following extract from the Windows 2000 setup-xcollection.cmd file in the SALES_DB example shows how to insert DTDs into the DTD_REF table:

```
db2 "connect to SALES_DB"
rem Insert DTDs
db2 "insert into db2xml.dtd_ref values('getstart.dtd',
 db2xml.XMLClobFromFile('%CD%\getstart.dtd'),
 0, 'user1', 'user1', 'user1')"
db2 "insert into db2xml.dtd_ref
  values('order.dtd', db2xml.XMLClobFromFile('%CD%\order.dtd'),
         0, 'user1', 'user1', 'user1')"
```

Chapter 5. Testing Web services applications

Verifying and testing Web services provider (WORF)

You can verify the Web service by using the DADX test page that is available if you deploy the WORF examples that are shipped with the federated server. You can copy the Java Server Pages from the WORF directory in the websphere-services.war file or the axis-services.war file to test some of the WORF functionality in your application.

You are now ready to create a Web service that accesses the SAMPLE database that comes with DB2. This scenario assumes that you installed the WORF samples as a Web application named *services*. The scenario also assumes that you configured *services* on your application server.

Testing Web services applications – a scenario

WORF supports the creation of Web services by using the document access definition extension (DADX) file. The DADX file contains necessary information to create a Web service and can reference the DAD file.

This scenario uses a simple DADX file, called HelloSample.dadx:

```
<?xml version="1.0" encoding="UTF-8"?>
 <DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx">

    <operation name="listDepartments">
      <query>
        <SQL_query>SELECT * FROM DEPARTMENT</SQL_query>
      </query>
    </operation>
 </DADX>
```

Figure 36. Simple DADX file: HelloSample.dadx

For the OS/390® and z/OS™ platforms, you might need to modify the name of the table to correspond with the sample DEPARTMENT table that is installed. This table has a default name of DSN8710.DEPT.

To deploy the Web service defined in the DADX file, copy it to the application server in the directory defined by the db2sample group in the dxx_sample directory.

When you add a new DADX file, WORF redeploys the Web service. You must restart the Web application to enable the new DADX file.

HelloSample.dadx defines a Web service with a single operation named `listDepartment`, which lists the contents of the DEPARTMENT table. The child tag <query> specifies the type of operation.

Testing the Web service

You can test your Web service by using the samples provided by the Web services provider.

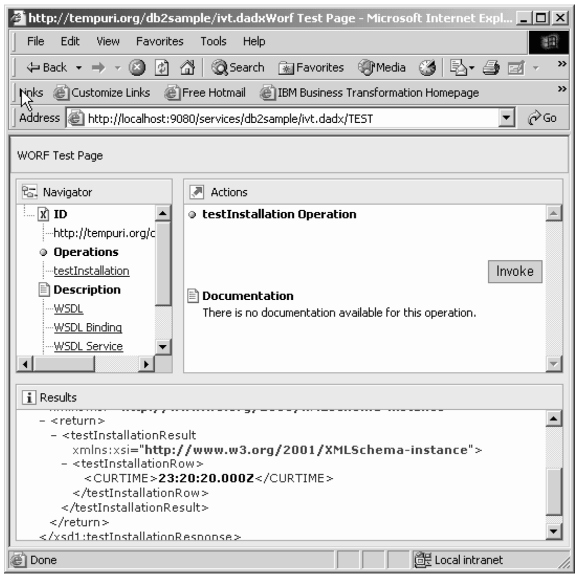
Procedure

To test your Web service:

1. Install the WORF samples in the directory \WEB-INF\classes\groups\dxx_sample.
2. Deploy the sample application in a server such as WebSphere Application Server (WAS).
3. Open a browser window and type the following uniform resource locator (URL) to begin the test:

   ```
   http://<your WebAppServer>/services/db2sample/ivt.dadx/TEST
   ```

   Remember that the identifier `<your WebAppServer>` depends on your Web server configuration. When you type the address, you see the following automatically generated documentation and test page



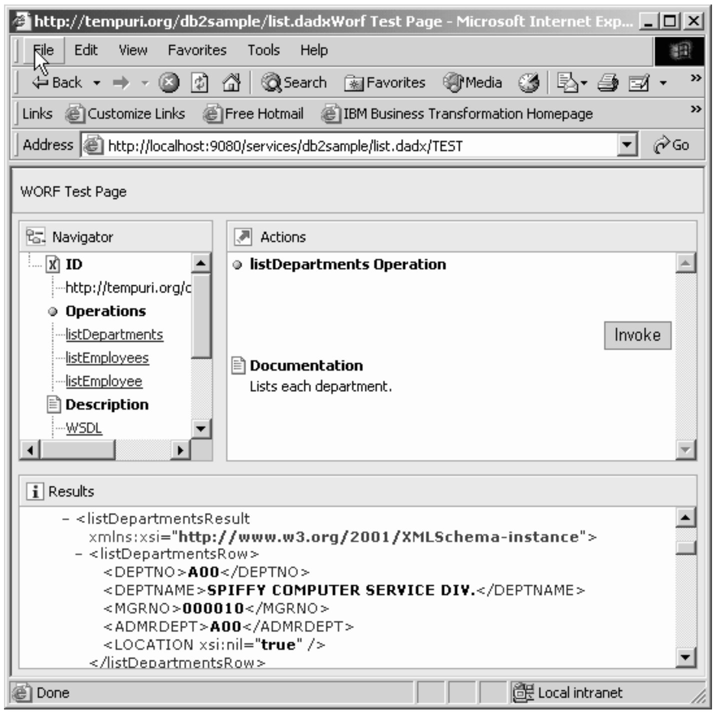
http://tempuri.org/db2sample/ivt.dadxWorf Test Page - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/db2sample/ivt.dadx/TEST Go
WORF Test Page
Navigator
ID
http://tempuri.org/c
Operations
testInstallation
Description
WSDL
WSDL Binding
WSDL Service
Actions
testInstallation Operation
Invoke
Documentation
Results
Done
Local intranet

There is no documentation available for this operation.

```
- <return>
  - <testInstallationResult
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    - <testInstallationRow>
        <CURTIME>23:20:20.000Z</CURTIME>
      </testInstallationRow>
    </testInstallationResult>
  </return>
</xsd1:testInstallationResponse>
```

Figure 37. The WORF test page

4. Test the `listDepartments` operation.
   a. Click the `listDepartments` link in the **Methods** pane.
   b. Click the **Invoke** push button in the **Inputs** pane. You can view the XML result of the operation in the Result pane:


http://tempuri.org/db2sample/list.dadxWorf Test Page - Microsoft Internet Exp...
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/db2sample/list.dadx/TEST
Go
WORF Test Page
Navigator
ID
http://tempuri.org/c
Operations
listDepartments
listEmployees
listEmployee
Description
WSDL
Actions
listDepartments Operation
Invoke
Documentation
Lists each department.
Results
Done
Local intranet

```
- <listDepartmentsResult
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <listDepartmentsRow>
      <DEPTNO>A00</DEPTNO>
      <DEPTNAME>SPIFFY COMPUTER SERVICE DIV.</DEPTNAME>
      <MGRNO>000010</MGRNO>
      <ADMRDEPT>A00</ADMRDEPT>
      <LOCATION xsi:nil="true" />
    </listDepartmentsRow>
```

Figure 38. Result of the query

Web services samples – PartOrders.dadx

The Web service example in this topic uses a database sample called dxx_sales_db. This is the sample database used in the documentation and samples shipped with DB2 XML Extender and with WORF.

The dxx_sales_db database stores information about part orders.

Suppose that you must provide a Web Service that retrieves orders that are based on the following conditions:

- Find all the orders
- Find all the orders for parts of a specified color
- Find all the orders whose price is greater than or equal to a minimum price

You create a DADX file named PartOrders.dadx that contains the following operations:

- findAll
- findByColor
- findByMinPrice

You create a Web Service by deploying the PartOrders.dadx file to the `services` Web application. This is configured with the dxx_sales_db instance of WORF. The deployment location of this file is WEB-INF/classes/groups/dxx_sales_db/PartOrders.dadx.

The Web Service supports access by the following protocols:

- Hypertext Transfer Protocol (HTTP) GET
- HTTP POST
- HTTP SOAP

HTTP GET and POST are useful for simple access from Web browsers. In this case, the request uses the content type of `application/x-www-form-urlencoded`.

For example, suppose that you deploy the Web services on the host www.mycompany.com. The following URLs would invoke the Web services using HTTP GET:

- `http://www.mycompany.com /services/sales/PartOrders.dadx /findAll`
- `http://www.mycompany.com /services/sales/PartOrders.dadx /findByColor?color=red`
- `http://www.mycompany.com /services/sales/PartOrders.dadx /findByMinPrice?minprice=20000`

This syntax encodes the method in the uniform resource locator (URL) as the extra path information and the parameters as the query string. The responses to these requests have a content type of `text/xml`. For HTTP POST, you send the query string in the body of the request instead of the URL, but its content type is still application/x-www-form-urlencoded. Here is an example of an HTTP POST request when captured with a Transmission Control Protocol (TCP) trace utility. The example shows both the HTTP header and body:

```
POST /services/sales/PartOrders.dadx/findByColor
HTTP/1.1
User-Agent: Java1.3.0
Host: localhost:9081
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
Content-type: application/x-www-form-urlencoded
Content-length: 12
color=red+++
```

A Web Service defined by a DADX file is self-describing. It dynamically generates a documentation and test page, WSDL documents, and XML Schema. The following HTTP GET URL requests the documentation and test page:

```
http://www.mycompany.com/services
/sales/PartOrders.dadx/TEST
```

The following HTTP GET URL requests the WSDL description of the service:

```
http://www.mycompany.com/services
/sales/PartOrders.dadx/WSDL
```

For HTTP SOAP, you invoke the services by sending SOAP envelopes using POST to the URL:

```
http://www.mycompany.com/services
/sales/PartOrders.dadx/SOAP
```

But with a request content type of `text/xml` instead of `application/x-www-form-urlencoded`. The following example is a SOAP request that is traced with a TCP monitor. It is like the one built into WebSphere Studio. This example includes the HTTP header information and the HTTP body:

```
POST /services/sales/PartOrders.dadx/SOAP
HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 547
SOAPAction: "http://tempuri.org/sales/PartOrders.dadx"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:findByColor xmlns:ns1="http://tempuri.org/sales/PartOrders.dadx" SOAP-
ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<color xsi:type="xsd:string" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">red </color>
</ns1:findByColor>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

.

PartOrder DADX file

PartOrders.dadx implements all three of its operations using the <retrieveXML> operator which uses the XML collection access method. In general, each operation can use a different operator and access method.

```
<?xml version="1.0"?>
  <DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <documentation>
      Provides queries for part order information at myco.com.
      See <xhtml:a href="../documentation/PartOrders.html" target="_top">
       PartOrders.html</xhtml:a> for more information.
    </documentation>

    <operation name="findAll">
      <documentation>

        Returns all the orders with their complete details.
      </documentation>
      <retrieveXML>
         <DAD_ref>getstart_xcollection.dad</DAD_ref>
         <SQL_override>
         select o.order_key, customer_name, customer_email,
           p.part_key, color, quantity, price, tax, ship_id, date, mode
         from order_tab o, part_tab p,
           table(select substr(char(timestamp(generate_unique())),16)
             as ship_id, date, mode, part_key from ship_tab) s
         where p.order_key = o.order_key and s.part_key = p.part_key
         order by order_key, part_key, ship_id
         </SQL_override>
      </retrieveXML>
    </operation>

<operation name="findByColor">
      <documentation>
Returns all  the orders that include one or
      more parts that have the specified
      color, and only shows the details for those parts.
</documentation>

      <retrieveXML>
        <DAD_ref>getstart_xcollection.dad</DAD_ref>
        <SQL_override>
         select o.order_key, customer_name, customer_email,
           p.part_key, color, quantity, price, tax, ship_id, date, mode
         from order_tab o, part_tab p,
           table(select substr(char(timestamp(generate_unique())),16)
             as ship_id, date, mode, part_key from ship_tab) s
         where p.order_key = o.order_key and s.part_key = p.part_key
           and color = :color
         order by order_key, part_key, ship_id
        </SQL_override>
        <parameter name="color" type="xsd:string"/>
      </retrieveXML>
    </operation>

 <operation name="findByMinPrice">
      <:documentation>
Returns all the orders that include one or more
   parts that have a price greater than
  or equal to the specified minimum price,
and only shows the details for
  those parts.
</documentation >
      <retrieveXML>
        <DAD_ref>
           getstart_xcollection.dad
       </DAD_ref>
        <SQL_override>
         select o.order_key, customer_name, customer_email,
           p.part_key, color, quantity, price, tax, ship_id, date, mode
         from order_tab o, part_tab p,
           table(select substr(char(timestamp(generate_unique())),16)
             as ship_id, date, mode, part_key from ship_tab) s
         where p.order_key = o.order_key and s.part_key = p.part_key
           and p.price >= :minprice
         order by order_key, part_key, ship_id
```

Installing a Web application that is used with the IBM Web Service SOAP provider engine

When you install an application with the Apache Axis SOAP engine, selecting the defaults from the WebSphere Application Server is all that you need to do. When you install with the IBM Web Service SOAP provider engine, you must prepare the WebSphere Application Server environment.

About this task

You can use Web archives files (WAR) to package, distribute, and install Web applications.

You generally package the files that make up your Web application in a single WAR file for deployment. A WAR file might contain the web.xml server configuration files, the group.properties configuration files, and the DAD and DADX files. The WORF samples that you deployed contain two examples of WAR files: websphere-services.war and axis-services.war.

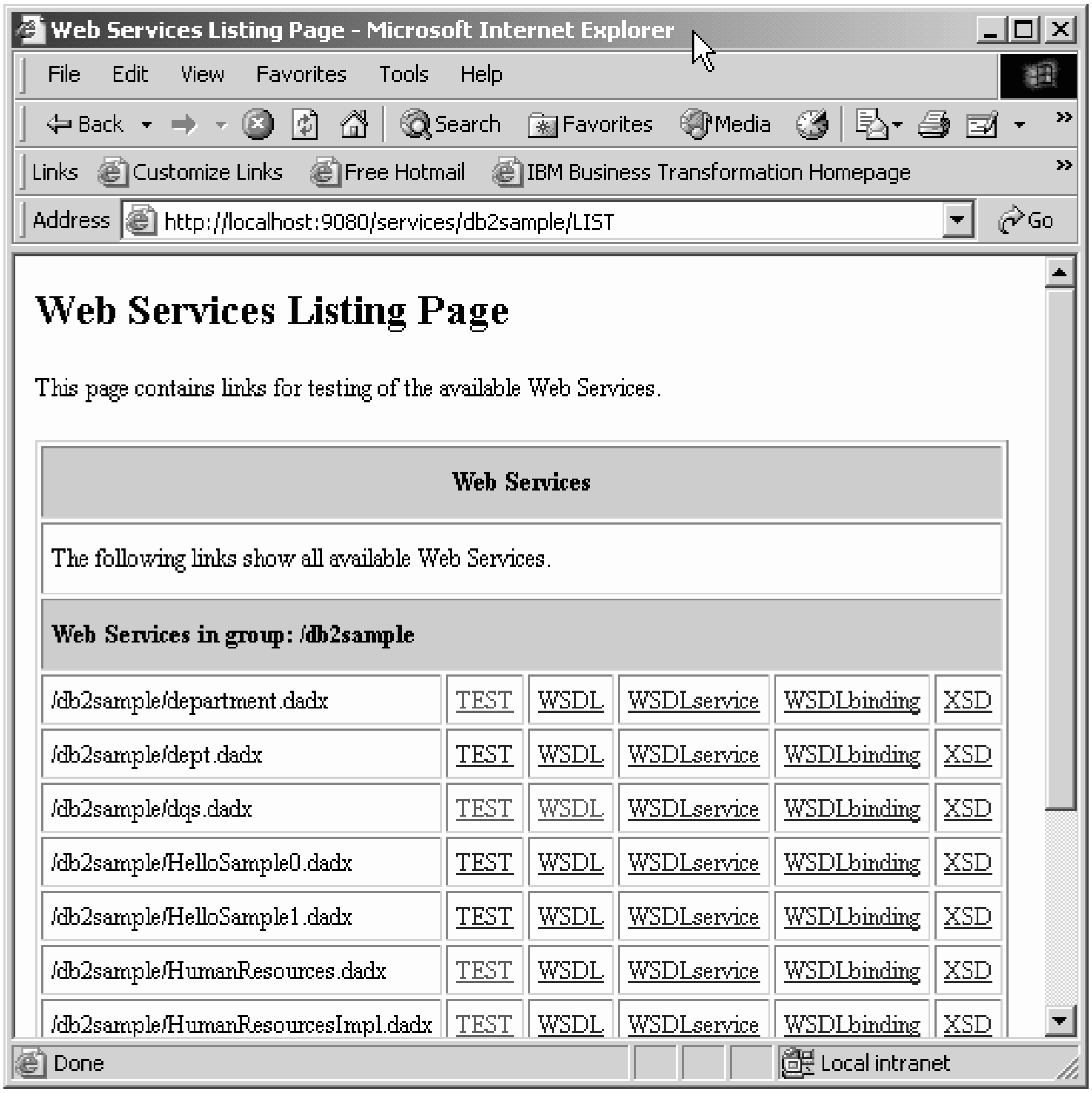
Procedure

To install a Web application that uses the IBM Web Service SOAP provider engine by installing websphere-services.war as an enterprise application:

1. Install the Web application with the following options on the WebSphere Application Server administration console:
   - Generate Default Bindings
   - Use Binary Configuration
   - Enable Class Reloading
2. Start the Web application.
3. Access the Web application by using the context root name from your browser with either the LIST or the WSDL function for each DADX file so that a deploying descriptor is generated.

   ```
   your-Web-server:9080/context_root_name/LIST
   ```
4. Re-start the Web application from the administration console of the WebSphere Application Server.

A list page at the group level looks like this:


Web Services Listing Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links Customize Links Free Hotmail IBM Business Transformation Homepage
Address http://localhost:9080/services/db2sample/LIST
Go
Done
Local intranet

Web Services Listing Page

This page contains links for testing of the available Web Services.

Web Services					
The following links show all available Web Services.					
Web Services in group: /db2sample					
/db2sample/department.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/dept.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/dqs.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HelloSample0.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HelloSample1.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HumanResources.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD
/db2sample/HumanResourcesImpl.dadx	TEST	WSDL	WSDLservice	WSDLbinding	XSD

Figure 40. Web services list page

Java 2 Enterprise Edition applications

You can use *.war files, *.jar files, and *.ear files in Java 2 Enterprise Edition applications.

When you deploy a Java™ 2 Enterprise Edition (J2EE) application, the following components must be built for an e-business application:

Web archive (WAR)
: The Web-related components (HTML, JavaScript™, JavaServer Pages)

Java archive (JAR)
The Java classes that make up the business logic components

Enterprise archive (EAR)
The Java archive files plus Web archive files that make up an enterprise solution

The minimum deployable unit in WebSphere® Application Server 5.0 is a Web archive file. If the application creates Enterprise JavaBeans™, then a Java archive file and an Enterprise archive file are required.

Preparing and creating the Web archive file

You can use Web archive files to package, distribute, and install Web applications.

Procedure

To create a Web archive (WAR) file:

1. Create the basic directory structure for the WAR file such as: WEB-INF\lib\worf-servlets.jar and WEB-INF\web.xml. These files are from the WORF dxxworf.zip file. The WORF directory hierarchy is in the websphere-services.war file or the axis-services.war file. You use these files when you run the TEST page. The files in the worf subdirectory are not necessary if you do not plan to use the built-in test facility of WORF. The worf-servlets.jar file is in the lib subdirectory where WORF is installed. The web.xml is the standard J2EE web.xml.
2. For each group:
   a. Create your group subdirectory, for example WEB-INF\classes\groups\myGroup, and include the group.properties and your DADX file in the subdirectory.
   b. Edit the WEB-INF\web.xml file to add the servlet and the servlet-mapping. For example, add a servlet name called myGroup and a URL-mapping called myURLPath.
3. Create the WAR file with either of the following methods:

Command line	Rational Web Developer for WebSphere Software
`jar -cvf minWORFwar.war WEB-INF worf`	Select **File** → **Export** → **WAR file**. Select the project name that your Web application is in and specify a file name.

4. Deploy the WAR file as described in the sample installations for WebSphere Application Server or Apache Jakarta Tomcat, for example, with myContext as the Web application context. For each new DADX file that you add to the application, WORF redeploys the Web service. You must restart the Web application to enable the additional DADX files. When you use the IBM Web Service SOAP provider engine perform the following steps:
   a. Install the Web application with the following WebSphere Application Server options: default binding, binary configuration and class reloading.
   a. Access the application with the Web services provider WSDL or LIST function.
   b. Restart the Web application.
5. Verify that you created the WAR file correctly by running the TEST page. For example, the URL for your TEST might look similar to the following:

   ```
   http://your WebAppServer/myContext/myURLPath/ivt.dadx/TEST
   ```

Note: The ivt.dadx file is a specific sample that is shipped. You might not have this file in your new WAR file

Examples of empty web.xml and dds.xml files

An empty web.xml file looks like the example in Figure 41.

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

Figure 41. Empty web.xml file

An empty dds.xml file looks like the example in Figure 42.

```
<?xml version='1.0'?>
<dds>
</dds>
```

Figure 42. Empty dds.xml file

Web services provider tracing

After you deploy your Web service, you might need to get information about run-time events and diagnostics from the Web service provider. To debug and troubleshoot your Web services application, DB2® Web services uses the trace facility of the Web application server on which your application runs. The trace information that you receive from the Web application server includes messages and event activity.

The DB2 Web services provider supports two tracing systems:

log4j Jakarta-log4j-1.2.8 and commons-logging-1.0.3 on Apache Jakarta Tomcat 4.0.6 and later

JRas The tracing and logging system that is used by WebSphere® Application Server, Version 5 and later

With these tracing systems you can incorporate message logging and trace facilities into your Java™ applications.

The output that is generated from a trace that you enable within your application appears in the root directory of the Web application server that you use. Table 14 shows the location of the output log file:

Table 14. Trace output location

Server	**Output log file location**
WebSphere Application Server	${SERVER_LOG_ROOT}/trace.log
Apache Jakarta Tomcat	<tomcat>/logs/worf_log4j.log

All trace messages and events are identified by the operation name of the Web service, the name of the servlet, or the name of the DADX file.

DB2 Web services provider can trace the following types of events:

Informational messages
: Messages that indicate when a Web service request event or a Web service response event completes successfully, such as when a DADX file is parsed successfully.

Warning messages
: Messages that indicate when a warning condition is detected during processing of the Web service request or the Web service response, such as a warning message from the XML parser for a DADX file.

Error messages
: Messages that indicate when an error is detected during the processing of the Web service request or the Web service response, such as when the application produces an exception.

Trace events
: Events that indicate when the application enters or exits a method, an exception, a call stack, or value of a variable.

Enabling tracing for the DB2 Web services provider-Apache Tomcat Version 4.0 or later Web application server

You can configure the Apache Tomcat server to trace your DB2 Web services.

Before you begin

You need authorization to modify the configuration of the server that you use.

Procedure

To enable tracing for the DB2 Web services provider:

1. Modify the default `log4j` tracing for the DB2 Web services provider.
2. View a log of your trace events at <installed Web server location>\AppServer\logs\<local server name>.
3. Create a configuration file with the name `log4j.configuration`.
4. Modify the settings in the configuration file to display only certain types of messages. For example: log4j.logger.com.ibm.etools.rt.webservice.*=INFO
5. Place the configuration file, `log4j.configuration`, in the WEB-INF/classes directory of the Web application.

Example of log4j.configuration

```
log4j.rootCategory=INFO, console, rollingFile
log4j.logger.com.ibm.etools.rt.webservice.*=INFO
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
log4j.appender.rollingFile=org.apache.log4j.RollingFileAppender");
log4j.appender.rollingFile.File=<servletContext>\..\..\logs\worf_log4j.log
log4j.appender.rollingFile.MaxFileSize=100KB
log4j.appender.rollingFile.layout=org.apache.log4j.TTCCLayout
log4j.appender.rollingFile.layout.layout.ConversionPattern=%p %t %c - %m%n
```

The following is a guide for message types in the log4j.configuration file.

Table 15. Message settings for log4j configuration file

Message type	Configuration setting
log4j warning messages or higher	log4j.logger.com.ibm.etools.webservice.*
log4j informational messages	log4j.logger.com.ibm.etools.webservice.*=INFO

Table 15. Message settings for log4j configuration file (continued)

Message type	Configuration setting
log4j error messages	log4j.logger.com.ibm.etools.webservice.*=ERROR

Enabling tracing for the DB2 Web services provider–WebSphere application server

You can configure the WebSphere application server to trace the DB2 Web services from the administrative console.

Before you begin

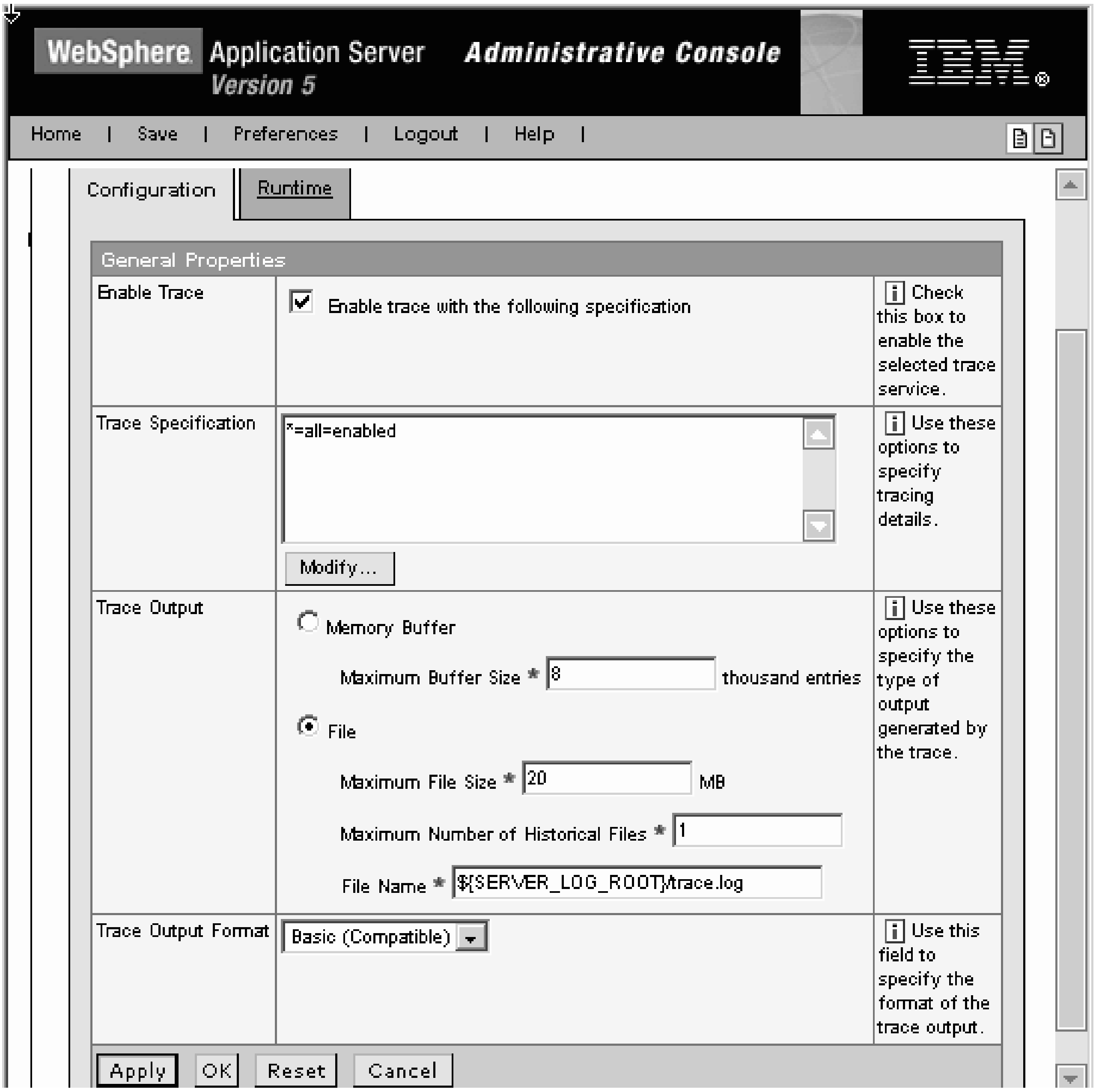
You need authorization to modify the configuration of the server that you use.

Procedure

To enable tracing for the DB2 Web services provider:

1. Modify the default jRAS tracing for the DB2 Web services provider.
2. View a log of your trace events at <installed Web server location>\AppServer\logs\<local server name>.
3. Start the WebSphere application server administrative console.
4. In the Navigation tree, click **Troubleshooting** → **Log and Trace** to open the Logging and Tracing window.
5. In the Logging and Tracing window, click the server name and then select **Diagnostic Trace**.
6. In the **Trace Specification** field, type the trace string:
   ```
   com.ibm.etools.webservice.*=all=enabled
   ```
7. If the server is stopped, go to the Configuration page. If the server is running, go to the Runtime page.
8. Optional: From the Runtime page, select the **Save trace** check box to write your changes to the server configuration. If the **Save trace** check box is cleared, the changes that you make apply only for the life of the server process that is currently running.
9. Optional: From the Configuration page, select the **Enable Trace** check box.
10. Save your changes and restart the server.

An example of enabling the trace when the server is not running.


WebSphere Application Server Version 5 Administrative Console
Home | Save | Preferences | Logout | Help |
Configuration
Runtime
General Properties

Enable Trace	✓ Enable trace with the following specification	Check this box to enable the selected trace service.
Trace Specification	*=all=enabled Modify...	Use these options to specify tracing details.
Trace Output	○ Memory Buffer Maximum Buffer Size * 8 thousand entries ● File Maximum File Size * 20 MB Maximum Number of Historical Files * 1 File Name * ${SERVER_LOG_ROOT}/trace.log	Use these options to specify the type of output generated by the trace.
Trace Output Format	Basic (Compatible)	Use this field to specify the format of the trace output.

Figure 43. Enabling the Web services provider trace

Enabling tracing for the DB2 Web services provider-Rational Web Developer

You can configure the WebSphere Studio to trace the DB2 Web services from the administrative console.

Before you begin

You need authorization to modify the configuration of the server that you use.

Procedure

To enable tracing for the DB2 Web services provider::

1. Modify the default jRAS tracing for the DB2 Web services provider.
2. View a log of your trace events at <installed Web server location>\AppServer\logs\<local server name>.
3. Start the WebSphere Studio administrative console.
4. From the main menu, click **Window** → **Show View** → **Server Configuration** to open the Server Configuration view.
5. On the **Servers** menu, double-click **WebSphere v5.0 Test Environment** to open the server editor.
6. Go to the Trace page.
7. In the **Trace Specification** field, type the following trace string:
   ```
   com.ibm.etools.webservice.*=all=enabled
   ```
8. Select the **Enable Trace** check box.
9. Save your changes and restart the server.

Publishing your Web services

Web service providers publish their Web services so that clients can access them using SOAP over HTTP.

This publication method contrasts with Enterprise Java™ Bean (EJB) clients that access beans by using remote method invocation (RMI) over Internet Inter-Orb Protocol (IIOP). Web services process requests from Web clients by invoking the appropriate business function and typically returning a response. The Web service description language (WSDL) document describes the Web service. You store the WSDL in a repository, such as a UDDI registry or on the server of the Web service provider. Storing the Web service description in an appropriate repository offers the potential for interested customers to discover its existence, potentially generating new business for the Web service provider.

Administering and troubleshooting the Web services provider

Using connection pooling to improve performance

You can use IBM WebSphere Application Server to help create and maintain a pool of database connections.

Before you begin

1. Create the JDBC provider if one does not exist that you want to use.
2. Create a data source.

About this task

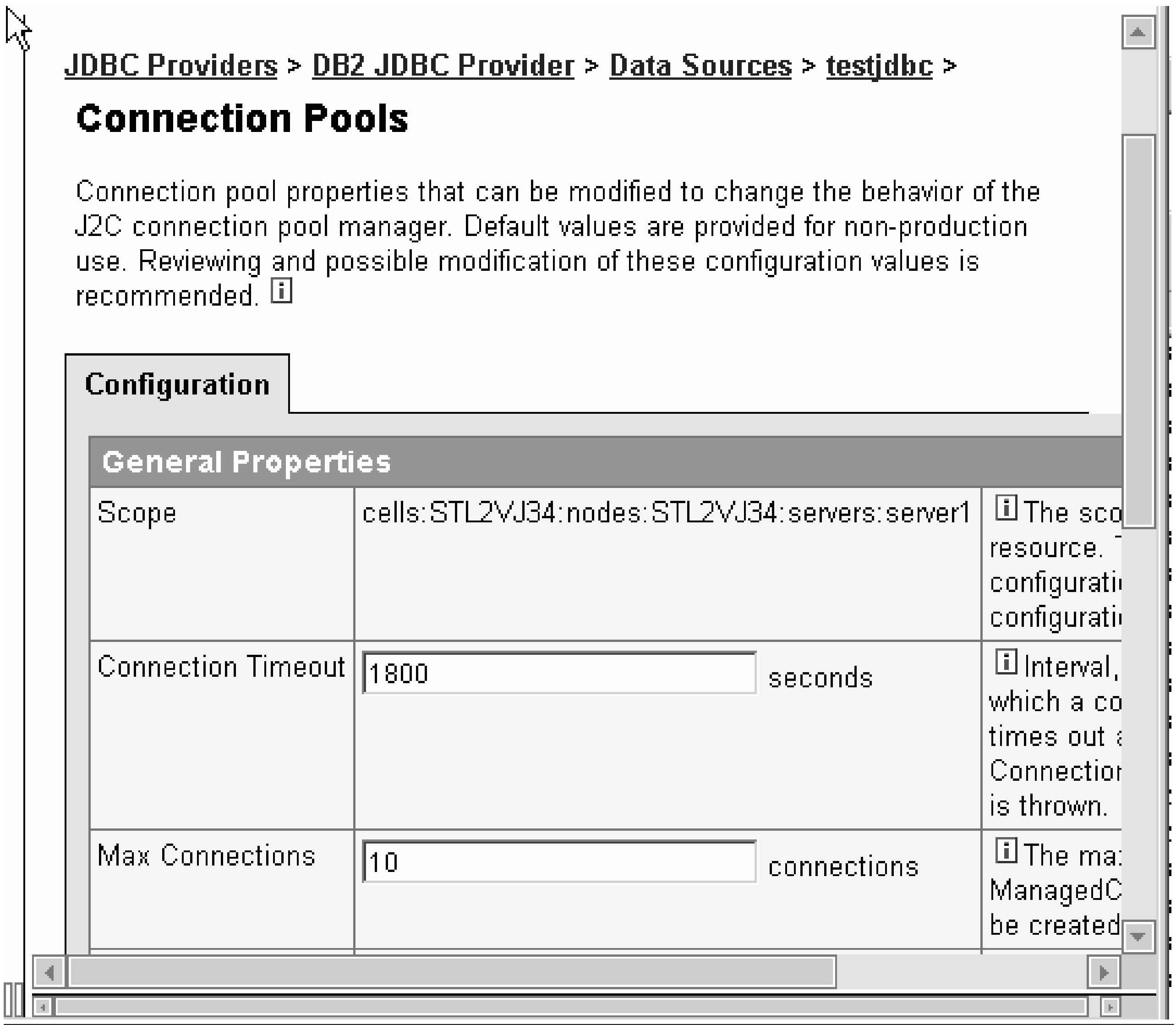
Each time a resource attempts to access a database, it must connect to that database. A database connection requires resources to create the connection, maintain it, and then release it when it is no longer required. The database resources required for a Web-based application can be high because Web users connect and disconnect more frequently. These database connections can be shared by applications on an application server to address the resource problems.

Installed applications use JDBC providers to access data from databases. By using a pool of database connections, you can spread the connection overhead across several user requests, thereby conserving resources for future requests and improving performance. You can configure a pool for each unique data source.

Procedure

To adjust some of the connection pooling parameters for a particular data source within a JDBC provider from WebSphere Application Server, Version 5, perform the following steps:

1. Configure the data source parameters.
2. Update the connection pooling information according to the guidelines in WebSphere handbook. Figure 44 shows the specific information that you should update.


JDBC Providers > DB2 JDBC Provider > Data Sources > testjdbc >

Connection Pools

Connection pool properties that can be modified to change the behavior of the J2C connection pool manager. Default values are provided for non-production use. Reviewing and possible modification of these configuration values is recommended.

Configuration

General Properties

Scope	cells:STL2VJ34:nodes:STL2VJ34:servers:server1	The sco resource. configurati configurati
Connection Timeout	1800 seconds	Interval, which a co times out Connection is thrown.
Max Connections	10 connections	The ma ManagedC be created

Figure 44. Connection pools

3. Edit the group.properties file in the groups subdirectory and add the following lines of text:

```
initialContextFactory=<your context factory>
datasourceJNDI=<your DataSource>
```

4. Restart the Web application if you have made any changes to the group.properties file so they will take effect.

The following example shows part of a group.properties file with the connection pool information:

```
initialContextFactory=com.ibm.websphere.naming.WsnInitialContextFactory
datasourceJNDI=jdbc/sampleDataSource
```

WebSphere Application Server provides a Java Naming service (JNDI) to facilitate the connection to DB2. The pool is shared by all applications connecting to the same data source.

Troubleshooting Web services

This topic describes some typical problems that might occur when you use WORF on WebSphere Application Server 5.1 and the recommended solutions.

Table 16 describes problems that can occur when you use WORF on WebSphere Application Server 5.1. The table provides recommended solutions.

Table 16. Errors and solutions

Problem	Solution
Error 500: Server caught unhandled exception from servlet [isd_demos]: org.apache.soap.rpc. SOAPContext: method setClassLoader (java\lang\ClassLoader;) not found	SOAP 2.2 or later (soap.jar) is missing.
Clicking on the **Invoke** button from the Web services test page in Internet Explorer results in a ’The page cannot be found’ error.	To view a more helpful error message use Netscape to debug the problem. Or, edit the Internet Explorer environment by doing the following steps: 1. Open the **Tools** menu from the Internet Explorer menu bar. 2. Select **Internet Options** from the menu to open the Internet Options window. 3. Click on the **Advanced** tab. 4. Clear the check box next to **Show friendly HTTP error messages**
Error 400: service ’http://tempuri.org /***/***.dadx’ unknown	You have to generate a deployment descriptor from the DADX file and restart your Web application before invoking the service.
Error 400: Unable to get DAD;unable to get Input Stream for: xxxxxx	There is no access to the specified XML Extender DAD file (for example, the DAD that is specified in the *.nst file)
Error 400: database connection error	• Database is not started. • The database objects that are referenced in the DADX file do not exist. • The JDBC driver is not found.
Error 400: unable to get input stream for /groups/xxx/ yyy.dadx	The DADX file is not in the group folder, or it is not accessible.

Table 16. Errors and solutions (continued)

Problem	Solution
Error 404: File not found: aaaa/abc.dadx	The servlet mapping 'aaaa' does not exist in the web.xml file.
blank page results	If you are using a version of WebSphere Application Server that is earlier than Version 5.0.2, you might be missing `jaas.jar`. Open SystemErr.log in the server directory to determine if other JAR files are missing.

To obtain information about runtime events and diagnostics from the Web service provider to troubleshoot your Web service after it is deployed, you can use the trace facility of the Web application server on which your application runs. The trace information that you receive from the Web application server includes messages and event activity. Even if the tracing is not enabled, errors are captured in the application server error logs.

Security in DADX Web services

You can secure Web services by using the security mechanisms of your application server. The mechanisms discussed here are authentication, encryption, and securing the database user ID.

Authentication

You can secure the DADX Web service endpoints by enabling authentication. When you enable authentication, you ensure that only those people who are authenticated can call your Web service. Access control in Java™ 2 Enterprise Edition (J2EE) is specified by the URL. Each DADX Web service uses URLs for different parts of the Web service, such as the endpoint and during the generation of the actual WSDL, and in the test page. For each URL that DADX uses, you can specify which roles (and as part of that definition you can specify which users) are allowed to access that URL. The process of setting authorizations for authenticated users of Web services is similar to granting SELECT privileges on a DB2 table.

The following table shows examples of URLs and URL patterns that you can use to protect Web services, specific operations, or services provided by WORF .

Table 17. Examples of URLs and patterns

Description	URL pattern
Enable access control for all URLs of a DADX, the test pages, and the WSDL generation by using the following URL:	`http://hostname:port/myContext/myGroup/myDadx.dadx/*`
Enable access control for only the test page by using the following URL	`http://hostname:port/myContext/myGroup/myDadx.dadx/TEST`
Enable access control for all of the DADX Web services in a group by using the following URL	`http://hostname:port/myContext/myGroup/*`

You can define security constraints in the Application Server Toolkit or Application Assembly tool of WebSphere® Application Server Version 5 or later. You can also define security constraints in the Web perspective of WebSphere Studio Application Developer Version 5 or later. The constraints that you define can include role names, so that anyone with that particular role name can access the Web area.

The DADX Web services support the Web services security that is part of the IBM Web Service SOAP provider.

Encryption

You can secure the DADX Web services by encrypting messages through HTTPS. Encryption ensures that nobody can read the messages that are exchanged between the Web service client and the Web service provider. See the documentation that is part of your application server to determine how to enable HTTPS.

Database security

In WebSphere Application Server Version 5, you can specify the database user ID on the application server for a JNDI data source. In the group.properties file, you can refer to that JNDI data source so that the DB2 Web service provider uses the user ID that you specify in WebSphere. Your database user password is encrypted on the application server.

For more information on authentication, encryption and data source authentication, see the information related to your particular application server. Also see the following WebSphere documentation for specific information on security:

- *IBM WebSphere Application Server, Version 5: Security*
- *IBM WebSphere V5.0 Security WebSphere Handbook Series*

Chapter 6. Web service consumer functions

The tools for the Web services consumer help you access Web services data by using SQL. The tools convert existing Web services description language (WSDL) interfaces into DB2 table or scalar functions.

You can invoke a set of user-defined functions (UDFs) that provide a client simple object access protocol (SOAP) over Hypertext Transfer Protocol (HTTP) interface to accessible Web services. You can call these functions directly from SQL statements.

You can construct the SOAP body according to the WSDL of a Web service. You can also use the Web service User-Defined Function (UDF) tool in Rational Application Developer to automatically generate specific UDFs. These UDFs can invoke operations that are defined by a user-specified Web services description language file. The generated UDFs are DB2 functions that do the following:
- Provide the parameters for the Web service request
- Invoke the SOAP client functions
- Map the result of the Web service invocation to the return types that are specified by the user

Firewalls

On some networks, access to the internet must go through a firewall. The traffic might be restricted to certain systems and certain ports that are allowed to send network traffic. Some systems allow applications to tunnel through the firewall. The SOAP UDFs support tunneling with SOCKS clients and HTTP proxies. To use a SOCKS server to tunnel through the firewall, you must install SOCKS client software on your system. To use HTTP proxies, you must set DB2 environment variables. Set **DB2SOAP_PROXY** to include the host name of the system with the HTTP proxy. Set **DB2SOAP_PORT** to the port of the HTTP proxy, such as 8080. In both cases, the SOAP traffic goes through the system that tunnels through the firewall.

Secure sockets layer (SSL) environment variables

The following environment variables for the SOAP UDFs and some non-relational UDFs support encryption between endpoints:

DB2SOAP_SSL_KEYSTORE_FILE
: Identifies the certificate storage file for SSL or transport layer security (TLS) communications. The value must be a fully qualified path name that is accessible by the DB2 agent or Fenced-Mode processes (FMP). The value GSK_MS_CERTIFICATE_STORE designates the native Microsoft® certificate storage.

DB2SOAP_SSL_KEYSTORE_PASSWORD
: Specifies the password for access to the SSL certificate storage file.

DB2SOAP_SSL_CLIENT_CERTIFICATE_LABEL
: Identifies the client certificate that is sent during an SSL authentication. If this value is not specified, the current DB2 authorization ID is used to locate the certificate.

DB2SOAP_SSL_VERIFY_SERVER_CERTIFICATE

Indicates whether the server certificate is verified during the SSL authentication. The value is case-insensitive and can be one of the following strings:

- Y
- YES
- T
- TRUE
- N
- NO
- F
- FALSE

The default value is NO.

Running the sample applications

You can test the sample federated applications with the following steps:

1. Start the database manager with the db2start command.
2. Create the sample database with the db2sampl command.
3. Establish a connection with the sample database.
4. Invoke the example files (in a Windows® environment, these example files are in <DB2 installed path>\samples\soap) with the following command: db2 -vf filename -t.

Installation of the Web services consumer user-defined functions

The db2enable_soap_udf command enables your database to use the SOAP requester functions.

Before you begin

You should install the following software before running the SOAP UDFs:

- DB2
  - Version 9 which includes Xerces parser and XML Extender
- Optional: Rational Application Developer for WebSphere Software Version 6, which provides wizards for exposing a variety of resources as Web Services.
- You must also enable DB2 XML Extender with the dxxadm enable_db sample command. See the*DB2 XML Extender Administration and Programming* for more options on the DB2 XML Extender commands.

Restrictions

The Web services consumer user-defined functions (UDFs) are available on the following platforms (all platforms are 32 bit):

- Windows 2000
- Linux
- AIX
- Solaris Operating Environment (with DB2 for Universal Database Version 8, Fix Pack 2)

Procedure

To enable, or install, and disable the Web service consumer:

1. Run the following command to register five user-defined functions:
   ```
   db2enable_soap_udf  -n dbName [-u uID] [-p password] [-force]
   ```
2. When you disable the Web service consumer, you drop the functions. Run the following command:
   ```
   db2disable_soap_udf  -n dbName [-u uID] [-p password]
   ```
3. You can also create and delete the user-defined functions by using the DB2 CLP (command line processor).
   ```
   CREATE SCHEMA db2xml;

      CREATE FUNCTION db2xml.soaphttpv (
                  endpoint_url VARCHAR(256),
                  soap_action VARCHAR(256),
                  soap_body VARCHAR(3072))
            RETURNS VARCHAR(3072)
         LANGUAGE C PARAMETER STYLE DB2SQL
         SPECIFIC soaphttpvivo EXTERNAL NAME 'db2soapudf!soaphttpvivo'
         SCRATCHPAD FINAL CALL FENCED
         NOT DETERMINISTIC CALLED ON NULL INPUT
         NO SQL EXTERNAL ACTION DBINFO;

      CREATE FUNCTION db2xml.soaphttpv (
                  endpoint_url VARCHAR(256),
                  soapaction VARCHAR(256),
                  input_message CLOB(1M))
            RETURNS VARCHAR(3072)
         LANGUAGE C PARAMETER STYLE DB2SQL
         SPECIFIC soaphttpcivo EXTERNAL NAME 'db2soapudf!soaphttpcivo'
         SCRATCHPAD FINAL CALL FENCED
         NOT DETERMINISTIC CALLED ON NULL INPUT
         NO SQL EXTERNAL ACTION DBINFO;

      CREATE FUNCTION db2xml.soaphttpc (
                  endpoint_url VARCHAR(256),
                  soapaction VARCHAR(256),
                  input_message CLOB(1M))
            RETURNS clob(1M)
         LANGUAGE C PARAMETER STYLE DB2SQL
         SPECIFIC soaphttpcico EXTERNAL NAME 'db2soapudf!soaphttpcico'
         SCRATCHPAD FINAL CALL FENCED
         NOT DETERMINISTIC CALLED ON NULL INPUT
         NO SQL EXTERNAL ACTION DBINFO;

      CREATE FUNCTION db2xml.soaphttpc (
                  endpoint_url VARCHAR(256),
                  soapaction VARCHAR(256),
                  soap_body varchar(3072))
            RETURNS clob(1M)
         LANGUAGE C PARAMETER STYLE DB2SQL
         SPECIFIC soaphttpvico EXTERNAL NAME 'db2soapudf!soaphttpvico'
         SCRATCHPAD FINAL CALL FENCED
         NOT DETERMINISTIC CALLED ON NULL INPUT
         NO SQL EXTERNAL ACTION DBINFO;

      CREATE FUNCTION db2xml.soaphttpcl (
                  endpoint_url VARCHAR(256),
                  soapaction VARCHAR(256),
                  soap_body varchar(3072))
            RETURNS CLOB(1M) as locator
         LANGUAGE C PARAMETER STYLE DB2SQL
   ```

```
        SPECIFIC soaphttpviclo EXTERNAL NAME 'db2soapudf!soaphttpviclo'
        SCRATCHPAD FINAL CALL NOT FENCED
        NOT DETERMINISTIC CALLED ON NULL INPUT
        NO SQL EXTERNAL ACTION DBINFO;
```

The Web service consumer WebSphere Studio plug-in is a component of WebSphere Studio Application Developer (WSAD) Version 5.1.1.

Parameters used in the enable and disable command

dbName
A database name

uID
Optional: User ID

password
Optional: The password associated with the user ID

–force
Attempts to drop any existing functions.

Web services consumer user-defined functions

A Web services consumer consists of SOAP requests and responses.

Simple Object Access Protocol (SOAP) is an XML protocol consisting of the following characteristics:

- An envelope that defines a framework for describing the contents of a message and how to process the message
- A set of encoding rules for expressing instances of application-defined data types
- A convention for representing SOAP requests and responses

DB2 needs the following information to build a SOAP request and receive a SOAP response.

- A service endpoint, for example, *http://services.xmethods.net/soap/servlet/rpcrouter*
- Some XML content of the SOAP body, which includes the name of an operation with requested namespace URI, an encoding style, and input arguments.
- Optional: A SOAP action URI reference. The reference can be empty, as shown in the following example, *http://tempuri.org/* or just ''.

The DB2 function db2xml.soaphttp() does the following actions:

1. It composes a SOAP request
2. It posts the request to the service endpoint
3. It receives the SOAP response
4. It returns the content of the SOAP body

This is an overloaded function that is used for VARCHAR() or CLOB(), depending on the SOAP body.

```
db2xml.soaphttpv returns VARCHAR():
    db2xml.soaphttpv (endpoint_url VARCHAR(256),
                     soap_action VARCHAR(256),
                     soap_body VARCHAR(3072))
                 RETURNS VARCHAR(3072)
db2xml.soaphttpv returns VARCHAR():
    db2xml.soaphttpv (endpoint_url VARCHAR(256),
                     soap_action VARCHAR(256),
                     soap_body CLOB(1M))
```

```
                             RETURNS VARCHAR(3072)

db2xml.soaphttpc returns CLOB():
       db2xml.soaphttpc (endpoint_url VARCHAR(256),
                       soapaction VARCHAR(256),
                       soap_body VARCHAR(3072))
                     RETURNS CLOB(1M)
db2xml.soaphttpc returns CLOB():
       db2xml.soaphttpc (endpoint_url VARCHAR(256),
                       soapaction VARCHAR(256),
                       soap_body CLOB(1M))
                     RETURNS CLOB(1M)

db2xml.soaphttpcl returns CLOB() as locator:
       db2xml.soaphttpcl(endpoint_url VARCHAR(256),
                        soapaction VARCHAR(256),
                        soap_body varchar(3072))
                     RETURNS CLOB(1M) as locator
```

Example of a DB2 constructed SOAP request envelope

The example in Figure 45 shows an Hypertext Transfer Protocol (HTTP) post header to post a SOAP request envelope to a host. The bold areas show the web service endpoint (post path and host) and the content of the SOAP body. The SOAP body shows a temperature request for zip code 95120.

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: services.xmethods.net
Connection: Keep-Alive User-Agent: DB2SOAP/1.0
Content-Type: text/xml; charset="UTF-8"
SOAPAction: ""
Content-Length: 410

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
                   xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
                   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
                   xmlns:xsd=http://www.w3.org/2001/XMLSchema >
   <SOAP-ENV:Body>
       <ns:getTemp xmlns:ns="urn:xmethods-Temperature">
         <zipcode>95120</zipcode>

       </ns:getTemp>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 45. A DB2 constructed SOAP request envelope

Example of using DB2 to extract the content of the SOAP response envelope

The example in Figure 46 on page 130 shows the HTTP response header with the SOAP response envelope. The **bold** content of the SOAP body shows the result of the temperature request. The namespace definitions from the SOAP envelope are not shown here, but they would also be included.

```
HTTP/1.1 200 OK
Date: Wed, 31 Jul 2002 22:06:41 GMT
Server: Enhydra-MultiServer/3.5.2
Status: 200
Content-Type: text/xml; charset=utf-8
Servlet-Engine: Lutris Enhydra Application Server/3.5.2
  (JSP 1.1; Servlet 2.2; Java™ 1.3.1_04;
   Linux 2.4.7-10smp i386; java.vendor=Sun Microsystems Inc.)
Content-Length: 467
Set-Cookie:JSESSIONID=JLEcR34rBc2GTIkn-0F51ZDk;Path=/soap
X-Cache: MISS from www.xmethods.net
Keep-Alive: timeout=15, max=10
Connection: Keep-Alive
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
                   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
                   xmlns:xsd=http://www.w3.org/2001/XMLSchema >
     <SOAP-ENV:Body>
           <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
           SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ >
              <return xsi:type="xsd:float">85<return>
           </ns1:getTempResponse>     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 46. Using DB2 to extract the content of the SOAP response envelope

Tracing Web services consumer events

The Web services consumer user-defined function can be traced by using the DB2 trace utility. In addition, when you use a Windows platform, you can trace HTTP SOAP requests and responses into a file.

Procedure

To trace the SOAP component in DB2:

Specify the following trace mask:

```
db2trc on -m *.*.147.*.*
```

Web services consumer—using the WebSphere Studio User-Defined Function tool

The Web services consumer User-Defined Function wizard generates and tests user-defined functions in WebSphere® Studio Version 5 or later.

The wizard used with WebSphere Studio reads the Web Services Definition Language (WSDL) file. It then generates the user-defined functions (UDFs) that provide easy access to Web services from database applications. You can use the generated UDFs in SQL statements to combine relational data with dynamic data that are retrieved from a Web service. You can invoke the Web service consumer functions directly in SQL. However, the task can require some advanced programming skills, and it can be time-consuming. After you generate and deploy the UDFs, you can use the functions in SQL to combine relational data with dynamic data that you retrieve from Web services. The generated UDFs are structured as follows:

1. Construct the SOAP body
2. Invoke the SOAP consumer (submit the SOAP request envelope)

3. Extract values from the SOAP response

How to generate the user-defined functions from WebSphere Studio

You can generate user-defined functions from the WebSphere Studio application.

Before you begin

1. Enable the DB2 XML Extender database
2. Enable the Web services consumer UDFs for the database
3. Create a project that you want to use with the Web service UDF
4. Create a connection to the database that you just enabled
5. Import the database to your WebSphere Studio, Version 5 project. See *WebSphere Studio Application Developer Programming Guide* for more information.

About this task

Within the WebSphere Studio, you can launch the wizard that generates the user-defined functions (UDFs) in three different ways.

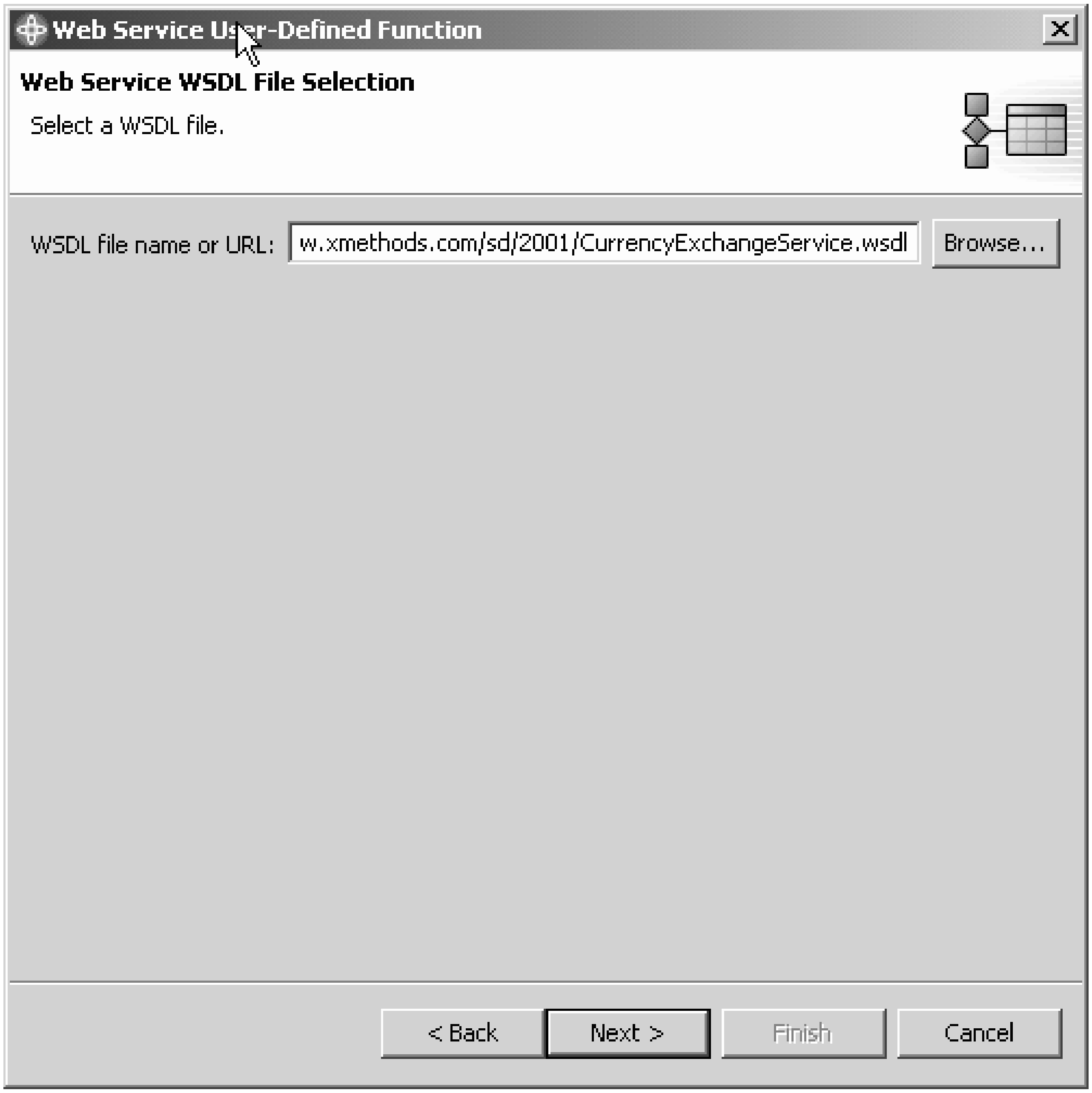
- You can invoke the wizard from the *File* > *New* > *Other*> menu. Then select **Data**. The folder expands and you select **Web Service User-Defined Function** from the menu. Click the **Next** push button to proceed.
- You can start the wizard in the Web service client wizard where it appears as an option in addition to generating a Java proxy.
- You can start the wizard as an option in the Web service wizard when generating a test client.

Procedure

To generate the user-defined functions from WebSphere Studio:

1. Specify the WSDL file from the first page of the wizard (see Figure 47 on page 133). You use this WSDL file to generate the UDF.
   a. Select a WSDL file from the work space or specify an appropriate uniform resource locator (URL). For example, the currency exchange rate Web service takes two countries as input parameters and returns the currency exchange rate between them. The WSDL file is at www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl.
2. Select the database. In Figure 48 on page 134, you see the database connection and a schema for which the UDF is generated. Click the **Browse** push button to select a database schema from the WebSphere Studio work space. The wizard requires that the database is enabled for the Web service consumer UDFs, and the DB2 XML Extender. If there is currently no connection to the specified database available, an additional message window asks for connection information. You can choose to immediately deploy the generated UDF into the database or generate a UDF in the WebSphere Studio work space only. You can deploy the UDF later.
3. Select the UDFs that you want to create. From the list of operations that are described in Figure 49 on page 135, select the one that you want to create. The wizard generates one UDF for every operation that is selected. Since the Web service that is used for this example provides only one operation, the wizard selects it automatically. Proceed to the next page.

4. Select options for the UDF. For each operation selected in the previous step, you can define options for those UDFs, such as changing the function name, or providing comments on the function. See Figure 50 on page 136 and Figure 51 on page 137.
   a. You can choose to build a scalar or a table function. Switching from a table function to a scalar function makes sense when the wizard should not automatically map the returned types. In this case, the wizard should return them as an XML fragment. Being able to switch from a scalar to a table function allows you to use the UDF in a FROM clause.
      - The wizard generates a scalar function when the Web service returns a simple XML type.
      - The wizard generates a table function when it returns a complex XML type. The table function automatically maps the complex XML type into multiple columns.
   b. You can include the input parameters as columns in the output table by selecting the **Echo the input parameters into the output table** check box.
   c. You can choose to generate a UDF with dynamic access to the Web service. When you do not specify the service location (the location attribute of the soap:address element) in the WSDL document, you generate a dynamic function. You can select the **Create a UDF with dynamic accesses to the service** check box even when the service location is specified to make use of late binding. When you generate a dynamic function, specify the service location at runtime as a parameter of the UDF.
   d. When using a Web service that can return responses of more than 3000 characters, specify **The Web service response message can be a big SOAP envelope** radio button. By default, **The Web service response message is always a small SOAP envelope** radio button is specified because this results in better performance for most Web services. If you specify the small SOAP response option and the wizard returns a SOAP envelope with more than 3000 characters, the generated Web Service UDF returns a descriptive error message.
   e. Select the **Return the whole SOAP envelope without parsing it** check box to help in debugging the Web services consumer UDFs.
5. From the Parameter page of the options window, you can review, and change the parameter mappings from WSDL types to SQL types (see Figure 51 on page 137).
6. From the Advanced Options page, you can specify the name for the UDF. If you do not specify a name, then a unique name is automatically generated by the database when you deploy the UDF.
7. Review the settings for generating the UDFs. Examine the database and schema, and the CREATE statement that will be issued on the database (see Figure 52 on page 138).
8. Click the **Next** or **Finish** push button. This generates the UDF and deploys it into the database, because of the earlier selections to generate and deploy.
9. You can run the Web service consumer UDF directly from the work space. To run the deployed UDF:
   a. Right-click on the UDF.
   b. Select **Run** (see Figure 53 on page 139). The Run Settings window opens (Figure 54 on page 140).
   c. From the Run Settings window, specify the parameter values.
   d. Click the **OK** push button to see the results of your test (Figure 55 on page 140).

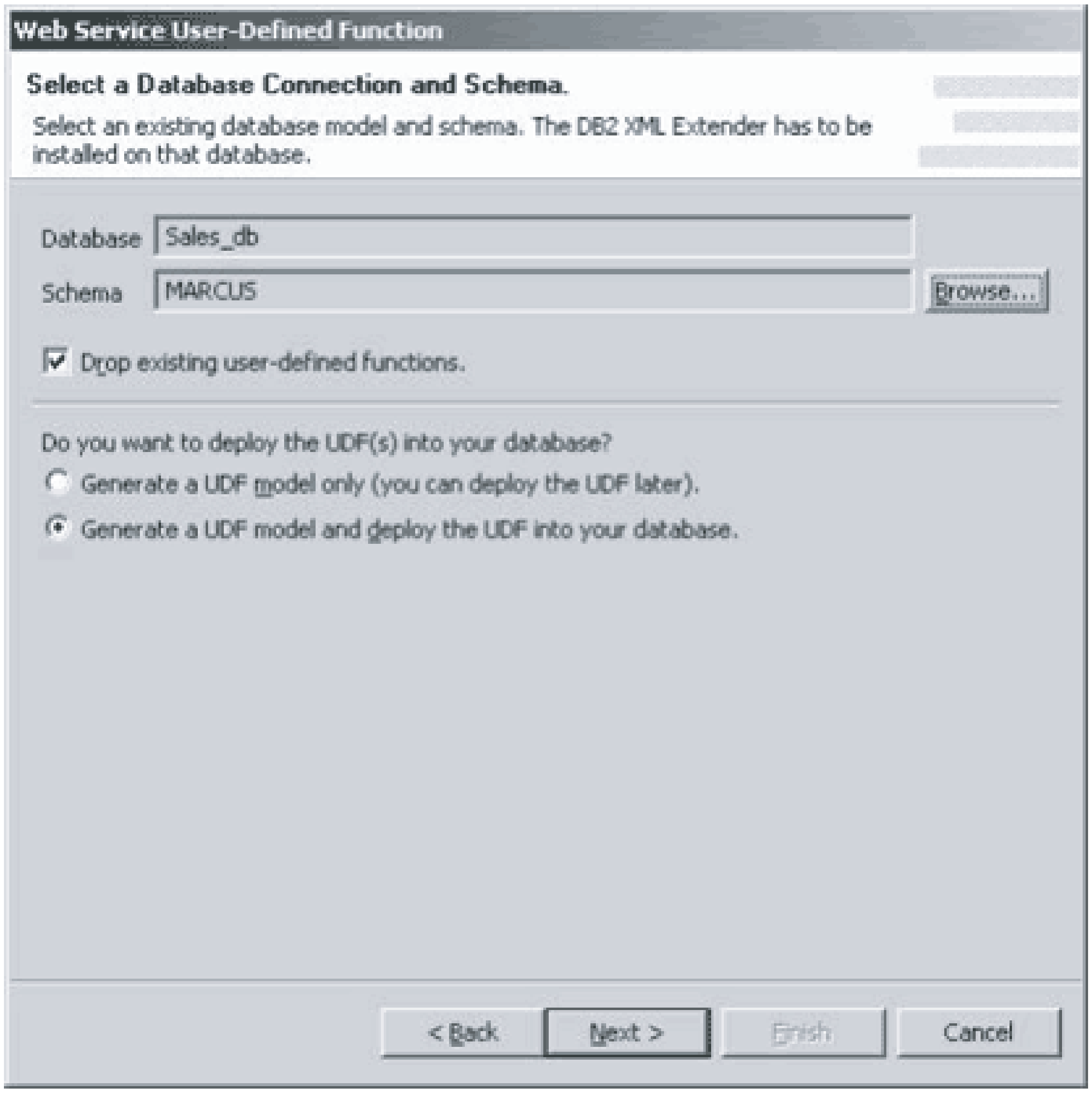


Web Service WSDL File Selection

Select a WSDL file.

WSDL file name or URL: w.xmethods.com/sd/2001/CurrencyExchangeService.wsdl

Figure 47. Select the WSDL file



Select a Database Connection and Schema.

Select an existing database model and schema. The DB2 XML Extender has to be installed on that database.

Database: Sales_db

Schema: MARCUS

[✓] Drop existing user-defined functions.

Do you want to deploy the UDF(s) into your database?

( ) Generate a UDF model only (you can deploy the UDF later).

(●) Generate a UDF model and deploy the UDF into your database.

Figure 48. WSDL page 2

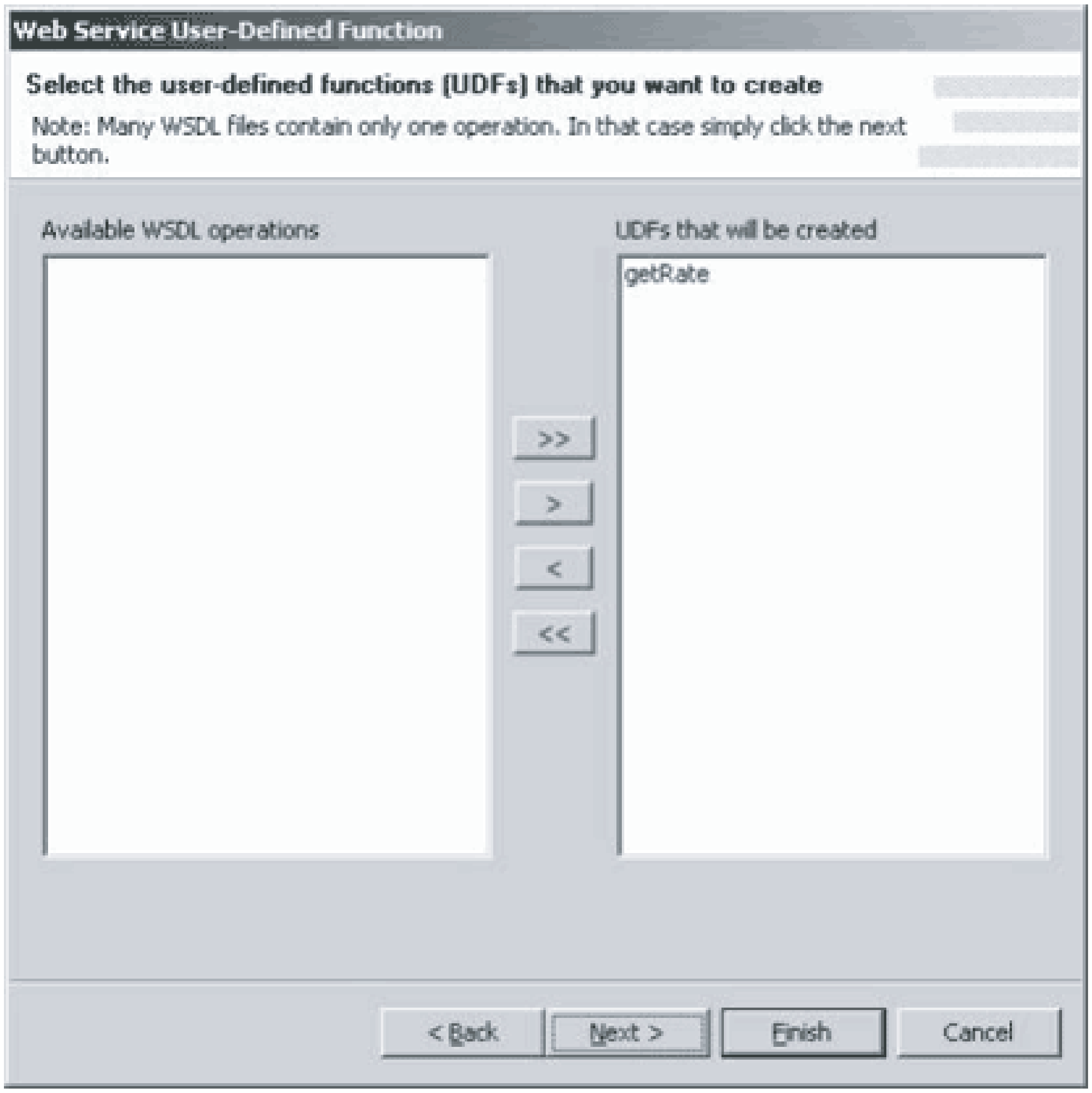
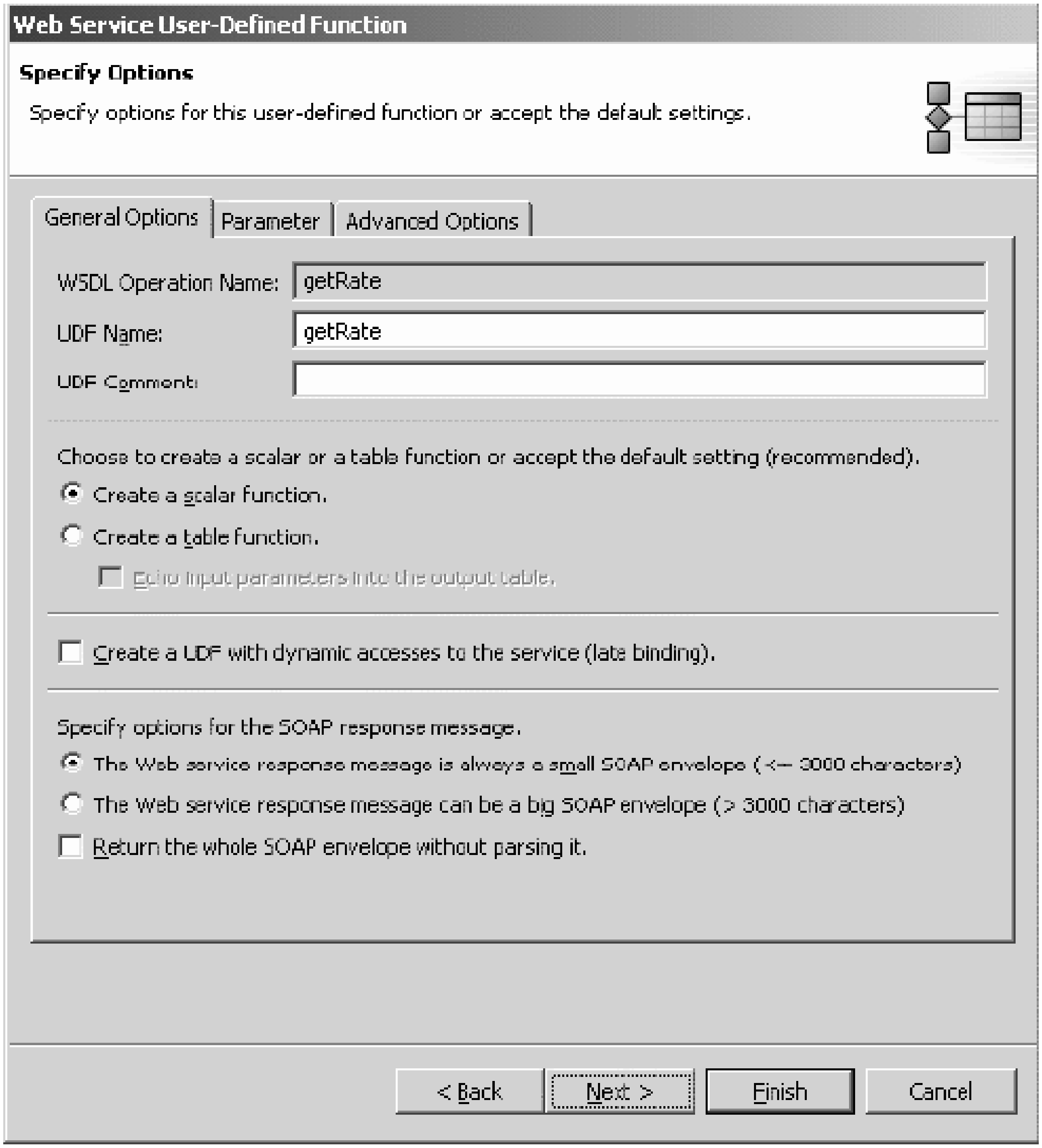

Web Service User-Defined Function
Select the user-defined functions (UDFs) that you want to create
Note: Many WSDL files contain only one operation. In that case simply click the next button.
Available WSDL operations
UDFs that will be created
getRate
>>
>
<
<<
< Back
Next >
Finish
Cancel

Figure 49. Wizard page 3


Web Service User-Defined Function

Specify Options

Specify options for this user-defined function or accept the default settings.

WSDL Operation Name:	getRate
UDF Name:	getRate
UDF Comment:	

Choose to create a scalar or a table function or accept the default setting (recommended).

[✓ Create a scalar function.]

Create a table function.

☐ Echo input parameters into the output table.

☐ Create a UDF with dynamic accesses to the service (late binding).

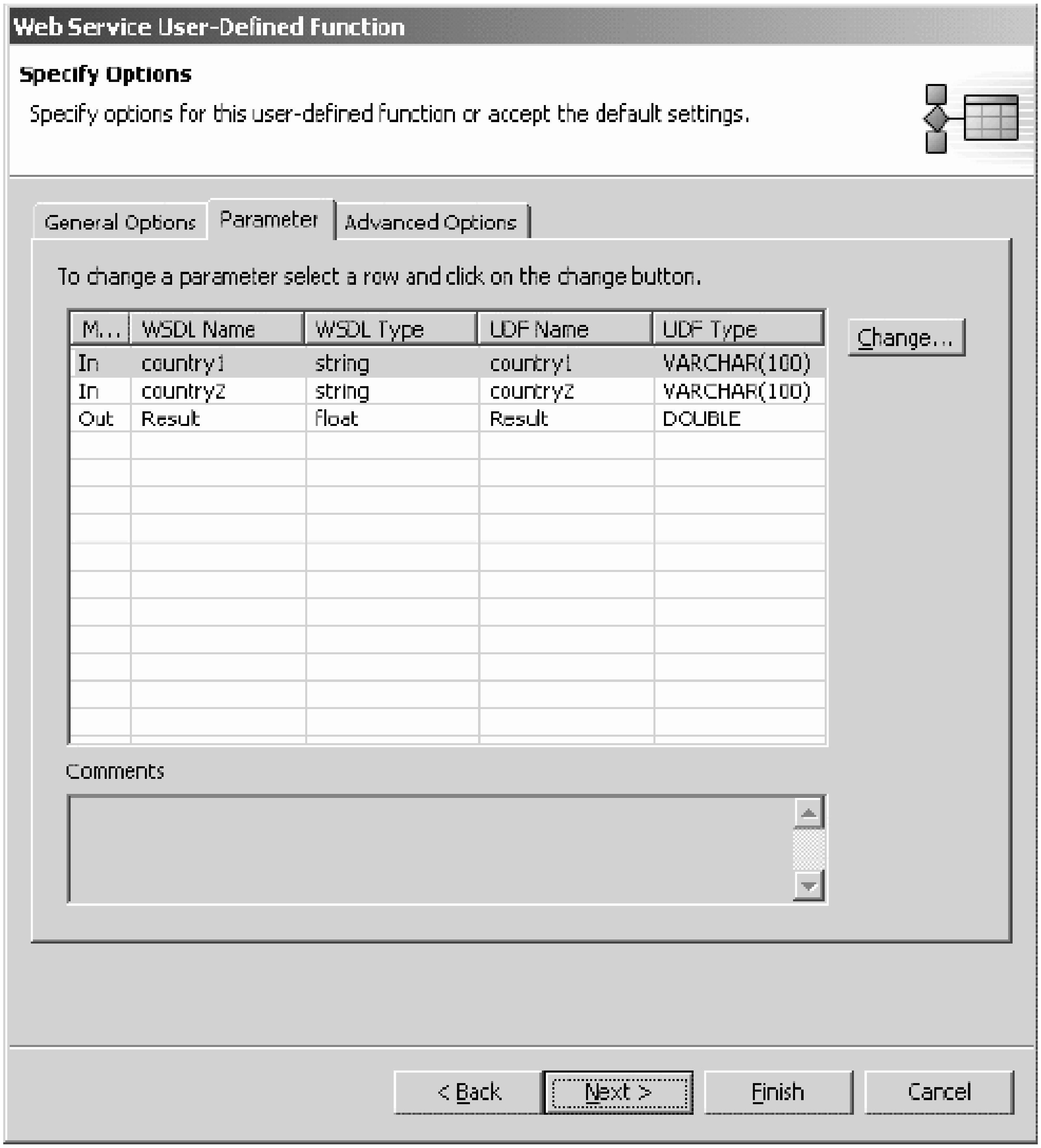
Specify options for the SOAP response message.

[✓ The Web service response message is always a small SOAP envelope (<= 3000 characters)]

The Web service response message can be a big SOAP envelope (> 3000 characters)

☐ Return the whole SOAP envelope without parsing it.

Figure 50. Select options page 1


Web Service User-Defined Function

Specify Options

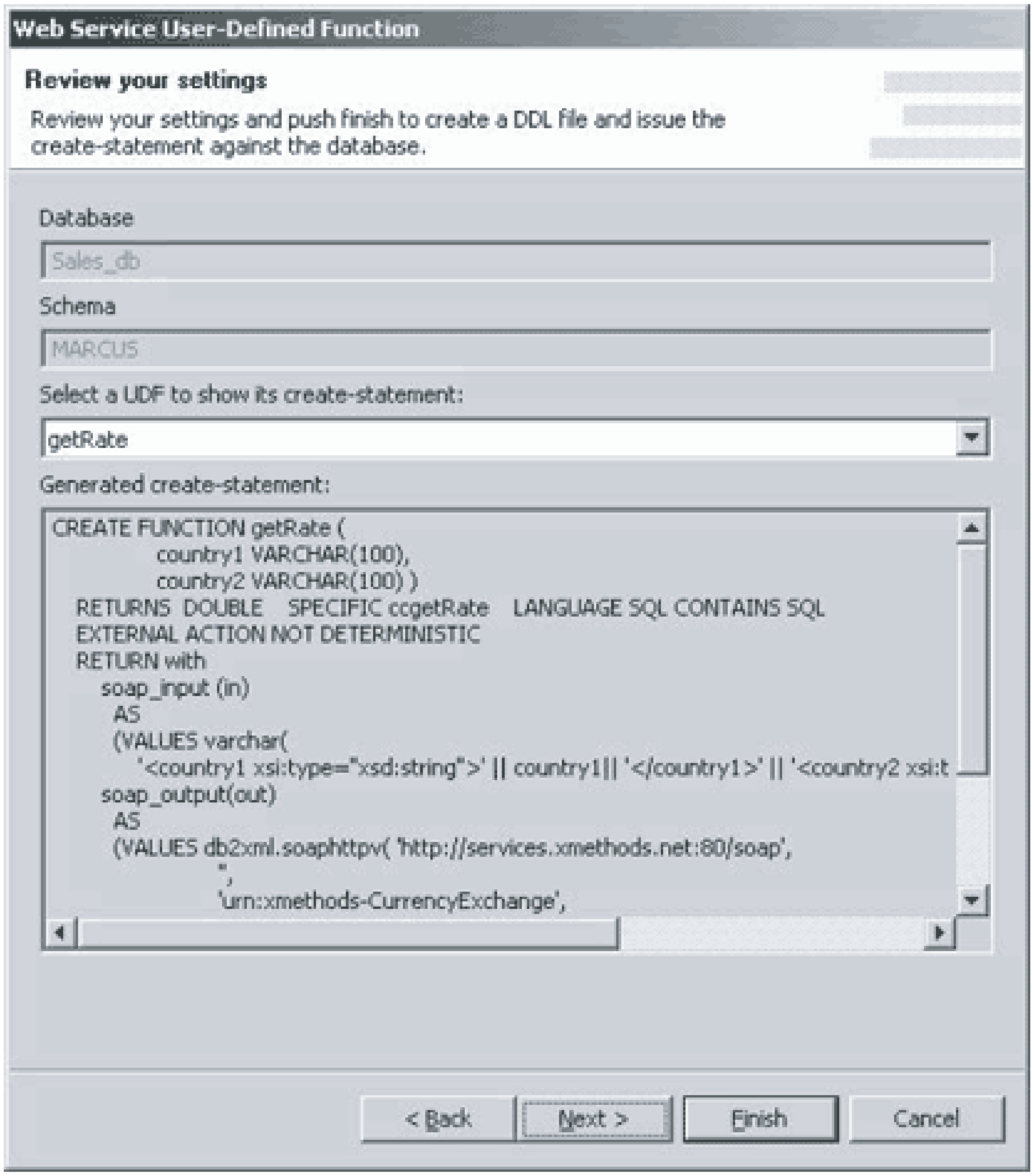
Specify options for this user-defined function or accept the default settings.

To change a parameter select a row and click on the change button.

M...	WSDL Name	WSDL Type	UDF Name	UDF Type
In	country1	string	country1	VARCHAR(100)
In	country2	string	country2	VARCHAR(100)
Out	Result	float	Result	DOUBLE

Comments

Figure 51. Select options page 2


Web Service User-Defined Function

Review your settings

Review your settings and push finish to create a DDL file and issue the create-statement against the database.

Database

Sales_db

Schema

MARCUS

Select a UDF to show its create-statement:

getRate

Generated create-statement:

```
CREATE FUNCTION getRate (
            country1 VARCHAR(100),
            country2 VARCHAR(100) )
  RETURNS  DOUBLE   SPECIFIC ccgetRate   LANGUAGE SQL CONTAINS SQL
  EXTERNAL ACTION NOT DETERMINISTIC
  RETURN with
    soap_input (in)
     AS
     (VALUES varchar(
       '<country1 xsi:type="xsd:string">' || country1|| '</country1>' || '<country2 xsi:t
    soap_output(out)
     AS
     (VALUES db2xml.soaphttpv( 'http://services.xmethods.net:80/soap',
                 '',
                 'urn:xmethods-CurrencyExchange',
```

Figure 52. Review

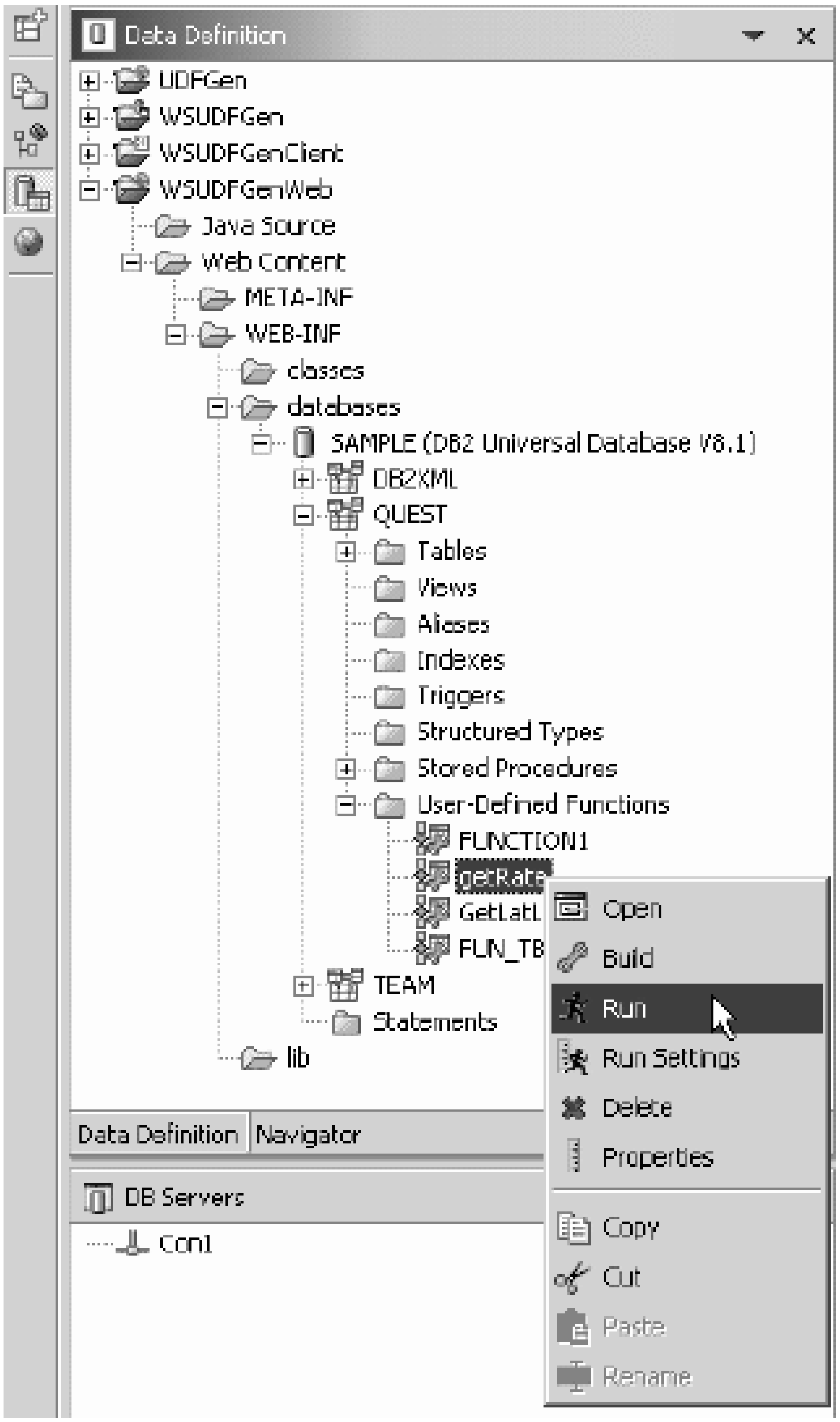
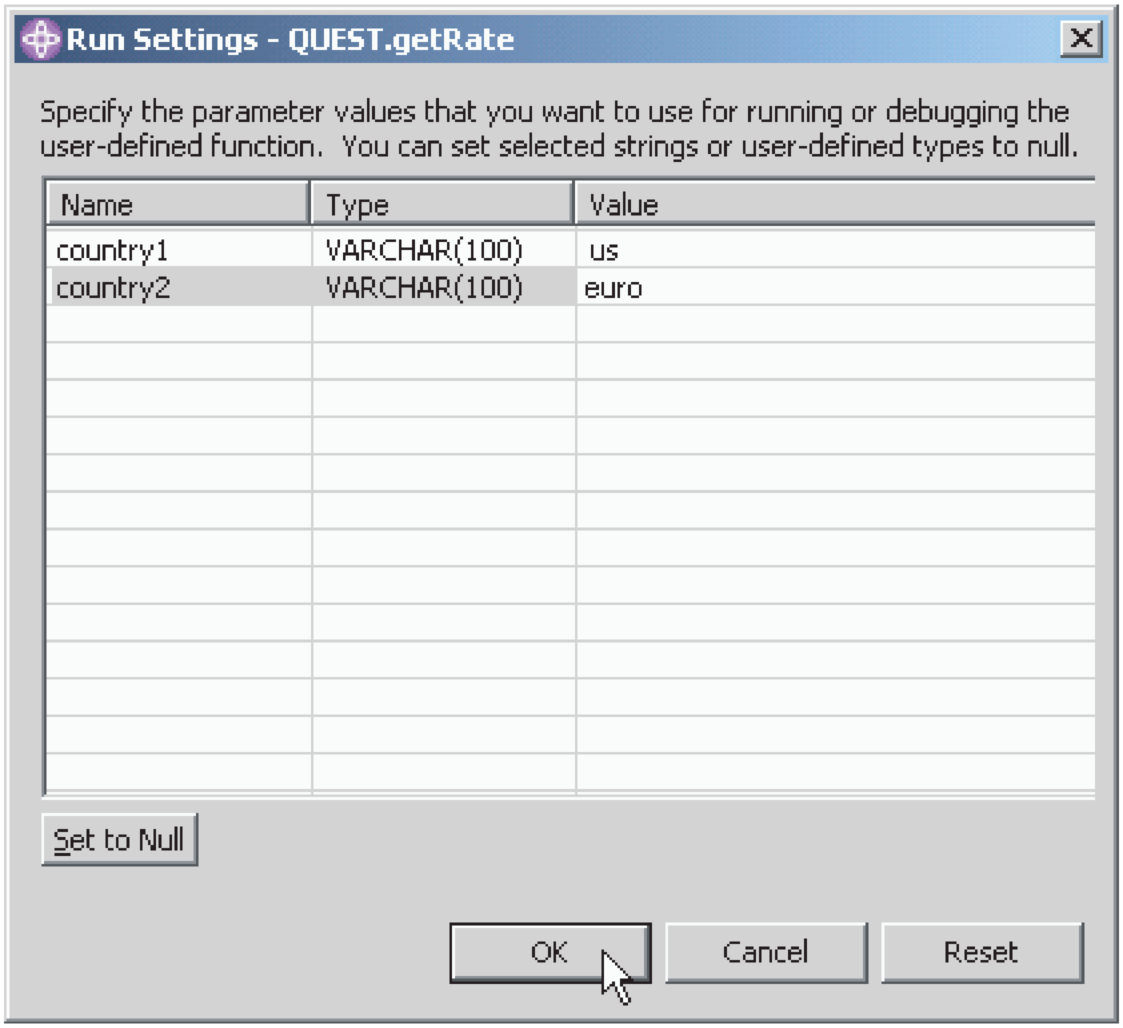


Figure 53. Test

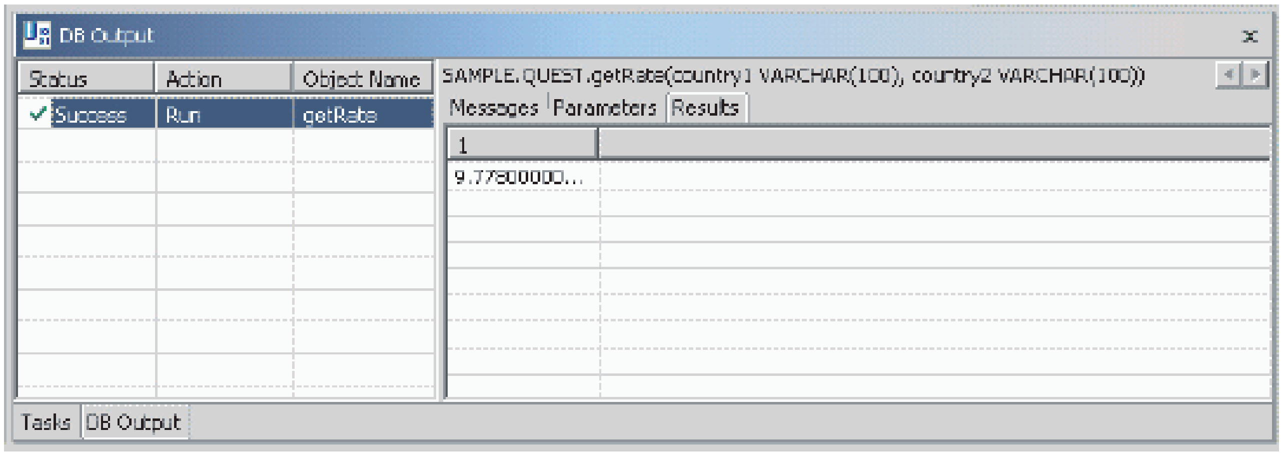


Run Settings - QUEST.getRate

Specify the parameter values that you want to use for running or debugging the user-defined function. You can set selected strings or user-defined types to null.

Name	Type	Value
country1	VARCHAR(100)	us
country2	VARCHAR(100)	euro

Figure 54. Run Settings



DB Output

Status	Action	Object Name
✓ Success	Run	getRate

SAMPLE.QUEST.getRate(country1 VARCHAR(100), country2 VARCHAR(100))

1
9.77800000...

Figure 55. Results

Using the Web services consumer UDFs

Use the User Defined Functions to share information between your relational tables and your Web services.

Assume that there is a table in a relational database with the following data:

Table 18. Products table

Product	Price
Gear	950.00
Nut	25.00
Bolt	35.00

And assume that there is information about currency types in a remote table.

Table 19. Currency table

Area
US
EURO
UK

Use the following SQL statement to determine how you can use the currency exchange rate function to display price information in Euros instead of US dollars. This accesses real-time exchange rates. Note that the statement uses a built-in decimal function to cast the price information.

```
SELECT product, decimal(getRate('us', 'euro') * price, 10, 2)
  as 'EUR_Price'
  FROM products
```

The result of the statement is:

Table 20. Using real-time exchange rates

Product	EUR_Price
Gear	1019.82
Nut	26.84
Bolt	37.57

Use the following SQL statement to show how you can use relational data as input to the Web service. The example shows how the currency exchange rate function can display price information in different currencies.

```
SELECT p.product, c.area,
    decimal(getRate('us', c.area) * price, 10, 2)
  as Price
  FROM products, areas
```

The result is:

Table 21. Displaying price information in different currencies

Product	Area	Price
Gear	us	950.00
Nut	us	25.00
Bolt	us	35.00
Gear	euro	1019.82

Table 21. Displaying price information in different currencies (continued)

Product	Area	Price
Nut	euro	26.84
Bolt	euro	37.57
Gear	uk	650.84
Nut	uk	17.12
Bolt	uk	23.97

If you use this query often, you might want to define a view to provide a simpler interface. An example of this view would be:

```
CREATE VIEW prices AS
  SELECT p.product, c.area,
      decimal(getRate('us', c.area) * price, 10, 2)
      as Price
  FROM p.products, c.areas
```

By using the view, you can code the following simpler query:

```
SELECT * FROM prices
```

Use the following SQL statement to show how you can use a UDF that is generated as a table function in a FROM clause. This example regenerates the getRate-UDF as a table function. The input parameters are echoed into the output table.

```
SELECT t.*
FROM countries c,
table( getRate('us', c.countries) ) t
```

The result is:

Table 22. Using getRate as a table function

AREA1	AREA2	RESULT
us	us	+1.00000000000000E+000
us	euro	+1.07280000000000E+000
us	uk	+6.84800000000000E-001

Web services consumer examples

This topic lists the SOAP samples and how to access them.

The examples that are referred to here work with DB2 Version 9. The file <DB2_installed path>/samples/soapsample.sql describes how to run the samples. The file soapsample.sql contains the following list of examples and sample queries:

- getTemp - Retrieves a temperature in Fahrenheit
- getRate - Returns the exchange rate between any two currencies

Chapter 7. DADX environment checker

The DADX environment checker performs different syntax and semantic checks on the NST, DAD and DADX files used to create and run Web services with WORF. Use the DADX environment checker to help minimize the number of errors that occur when deploying Web services with WORF.

The DADX environment checker is a Java application that is called from the command line. When invoked, it produces an output file that contains errors, warnings, and success indicators. The name of the output text file is user-defined. If no name is specified, the standard output is used.

The DADX environment checker is included in the WORF installation, in the tools\lib subdirectory. The JAR files containing the code for this tool are CheckersCommon.jar and DADXEnvChecker.jar. Make sure that you have a JRE or JDK Version 1.3.1 or later, installed on your system. Update your CLASSPATH to include all of the following archives:

- CheckersCommon.jar, DADXEnvChecker.jar and worf.jar, included in the tools\lib directory where WORF is installed
- xerces.jar . For UNIX and Windows, these files are included in the binary distribution for Xerces-J 2.0.2 downloadable at http://xml.apache.org/. For OS/390 and z/OS, these files are included in the IBM XML Toolkit Version 1 Release 4 with PTF UW95866
- soap.jar, included in the binary distribution for SOAP 2.3 downloadable at http://xml.apache.org, or included in the WebSphere Application Server installation.
- j2ee.jar, version 1.3 or later. You can download this file from java.sun.com
- qname.jar . You can download this file from java.sun.com
- wsdl4j.jar. You can download this file from http://oss.software.ibm.com/developerworks/projects/wsdl4j.
- activation.jar, included in the binary distribution for JavaBeans Activation Framework 1.0.1, downloadable at http://java.sun.com
- mail.jar, included in the binary distribution for JavaMail 1.2 downloadable at http://java.sun.com
- servlet.jar, included in the WebSphere Application Server installation, or in the distribution for Jakarta Tomcat Version 3.2.x through 4.0.3 or later downloadable at http://www.apache.org/
- For UNIX and Windows: db2java.zip, included in the /java directory located where you installed DB2. For OS/390 and z/OS: db2j2classes.zip, included in the classes/ subdirectory where you installed DB2 in HFS. You can also use jcc.jar. The `dbDriver` parameter in the group.properties files determines the driver package that you use.

For example, if you are running in the Windows environment, you must set your CLASSPATH to find the following files:

```
CheckersCommon.jar;
DADXEnvChecker.jar;
worf.jar;
xerces.jar;
j2ee.jar
```

```
qname.jar
wsdl4j.jar
soap.jar;
db2java.zip;
```

Running the DADX environment checker

The DADX environment checker is a Java program that can run on JDK version 1.3.1 and later.

You run the DADX environment checker by running the following command written on a single line:

```
 java com.ibm.etools.webservice.util.Check_install
    [-srv] [-schdir pathToSchemasDir]
    [-sch schemaLocations] [-out outputFile]
   fileToCheck
```

For example, the following command assumes that you extracted the dxxworf.zip file to directory c:\dxxworf. This command runs the DADX checker on the resource files contained by the c:\tomcat\webapps\services directory. The command then sends the output to myOutputFile.txt in the current directory:

```
 java com.ibm.etools.webservice.util.Check_install
 -srv -schdir c:\dxxworf\schemas
-out myOutputFile.txt c:\tomcat\webapps\services
```

DADX environment checker parameters

The parameters that can be used to run the DADX environment checker.

Parameters

-schdir *pathToSchemasDir*
: Specifies the absolute path to the directory where the schemas that are used for validating NST and DADX files are stored.

-sch *schemaLocations*
: Specifies a list of schemas for the parser to validate the files. The DADX checker allows the user to specify the value of a property of the Xerces parser. This property can be used to specify the location of XML schemas that perform the validation of the files that are being parsed. You specify the location of a schema by providing the name of the target namespace of the schema (for example: http://myschema) that is followed by the actual location of the schema. The path could be a path in the file system (for example, c:\dir\schema1.xsd) or a valid URL. But, the XML documents themselves can contain declarations of schema locations. The schemaLocation attribute is used in an XML document to provide this information. Here is an example of the beginning of an XML document:

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/Report
http://www.example.com/Report.xsd">
```

For a particular namespace, the parser uses the schema location that is defined by using the property of the parser, even if the schemaLocation attribute defines another schema location for the same namespace. The syntax for schemaLocations is the same as for schemaLocation attributes in instance documents. For example, http://www.example.com file_name.xsd is the syntax for schemaLocations. The user can specify more than one XML Schema, for example, -sch http://www.example_1.com file_name_1.xsd http://www.example_2.com file_name_2.xsd

-out *outputFile*
Specifies the output text file name. If omitted, the standard output is used.

-srv
Indicates that the checks must be performed on all of the NST, DAD and DADX files that are found under the Web services module directory (for example c:\tomcat\webapps\services) that are passed as the fileToCheck. If this option is not used, then the checks are performed only on the DADX file that is passed as the file to check and on the related data that is contained in other resource files. For example, the DAD files that are referred to in this DADX file are checked and then the DTDIDs that are referred to in these DAD files are checked in the NST file. Only the data that is related to the DADX file is checked in the NST file and in the web.xml file.

fileToCheck*path*
If parameter -srv is not used, then the value of fileToCheck is the DADX file that is checked. If parameter -srv is used then the fileToCheck value is the root directory of the Web services module. For example, the root directory of an unzipped .war file is services for either the websphere-services.war or axis-services.war module.

-help
Displays command line option information.

-version
Displays version information.

DADXEnvChecker_sample.txt

A sample file is found in the tools\samples directory in the dxxworf.zip file. DADXEnvChecker_sample.txt is an output text file that shows the results of the checks that are performed on a Web services module. The DADX environment checker generates this file. The checker uses the file name DADXEnvChecker_sample.txt that is specified in the **-out** parameter.

Indicating errors and warnings in the output text file

When the `-srv` parameter is used, errors, warnings, and success indicators are grouped together in paragraphs. Each paragraph is associated with a checked file.

The results of checking each file are displayed in the output file if you indicated a file name, or in the standard output device if no filename is indicated.

The paragraphs are grouped together according to the path, or subdirectories, in directory **groups**. Here is an excerpt of an output text file showing the error messages corresponding to the checks performed in files sales_db.nst and getstart_xcollection.dad belonging to group /groups/dxx_sales_db:

```
## Checking group: c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sales_db
## Checking NST file: c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sales_db\sales_db.nst
INFO. Line 5: file "c:\dxx\samples\dtd\getstart.dtd" is accessible.
ERROR. Line 12: file "wrongDtd.dtd" CANNOT be found
either in the file system or in the database.
INFO. Line 8: file "getstart.dtd" is accessible.
## Checking DAD file: c:\tomcat\webapps\services\WEB-
```

```
INF\classes\groups\dxx_sales_db\getstart_xcollection.dad
WARNING. Line 4: DTDID "dtd_.dtd" CANNOT be found in the DTD_REF table.
INFO. Line 9: the DTDID "c:\dxx\samples\dtd\getstart.dtd"
has been declared in the NST file.
```

Errors, warnings and success messages can begin with a line number if the error or warning or success event is related to a specific line. The line numbers in the output text indicate the line numbers where the checked elements associated with the messages were found in the files. There is no order related to the output within a paragraph.

Error checking by the DADX environment checker

The DADX environment checker performs checks on web.xml, NST, DAD, and DADX files.

When you invoke the DADX environment checker with the `-srv` parameter, the first check that is made is on the web.xml file within directory WEB-INF. Then, the DADX environment checker performs checks on the NST, DAD, and DADX files found in each group directory in the WEB-INF\classes\groups directory.

When you invoke the DADX environment checker without the `-srv` parameter, the first check that is made is on the DADX file that is passed as the file to check. Then, the DADX environment checker checks the DAD files that are referenced in this DADX file. It also performs checks on the NST file of the group to which the DADX file belongs. The DADX environment checker eventually checks the web.xml file within the WEB-INF directory containing the DADX file.

Database error message

For some checks on NST and DADX files, the DADX environment checker performs the following actions:

1. Attempts to establish a connection to the database by using data contained in the file `group.properties`
2. Queries the database with which the group is associated
3. Checks the files of a group for errors

If the connection to the database fails, the DADX environment checker issues an error message. The following example shows a typical error message:

```
Checking group: c:\test\jakarta-tomcat-3.2.2
##Checking group: c:\tomcat\webapps\services
\WEB-INF\classes\groups\dxx_travel
WARNING. Connection error [IBM][CLI Driver]
SQL1013N The database alias name or database name
"TRAVELLL" could not be found.
SQLSTATE=42705
```

Checking errors in the web.xml file

The DADX environment checker checks the WEB-INF\web.xml file that is located in the root directory of the Web Service module, which is services in this example.

An excerpt of a Web.xml file

```
<servlet>
<servlet-name>dxx_sales_db</servlet-name>
<servlet-class>com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker
</servlet-class>
<init-param>
<param-name>faultListener</param-name>
<param-value>org.apache.soap.server.DOMFaultListener
```

```
</param-value>
</init-param>
<load-on-startup>-1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>dxx_sales_db</servlet-name>
<url-pattern>/sales/*</url-pattern>
</servlet-mapping>
```

The <servlet-class> tags, which are direct children of the <servlet> tags must have a value of either com.ibm.etools.webservice.rt.isd.servlet.IsdInvoker or com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker. When their values are different, the checker provides an error message.

Results of a check on <servlet-class> tags

The following example shows the results of the checks performed on <servlet-class> tags in a web.xml document:

```
INFO. Line 21: servlet class for
servlet "dxx_sales_db" is a correct servlet class.
ERROR. Line 31: servlet class
"com.ibm.etools.webservice.rt.dxx.servlet.OtherInvoker"
for servlet "dxx_sample"
is NOT a correct servlet class.
INFO. Line 41: servlet class
for servlet "dxx_travel"
is a correct servlet class.
```

Each <servlet-mapping> tag contains a <servlet-name> tag with a value that must be the same as the value of the <servlet-name> tag of a <servlet> tag. If this is not the case the checker provides an error message as shown in the following example:

```
ERROR. There is no <servlet>
tag declaring servlet
"isd_demos" mapped at line 50.
```

Each <servlet> tag must have a corresponding <servlet-mapping> tag with the same servlet name. If a <servlet> tag has no corresponding <servlet-mapping> tag, the checker provides the following kind of message:

```
ERROR. There is no
<servlet-mapping> tag
for servlet "dxx_travel" declared
at line 40.
```

Each <servlet-mapping> tag also contains a <url-pattern> tag with a value that must be unique. If two <url-pattern> tags have the same value, the checker provides an error message as shown in the following example:

```
ERROR. Line 56: "/sales/*" is already
declared as the URL pattern for servlet "isd_demos"
(see line 50).
```

Checking errors in the NST files

NST files declare the namespace table of the group. They contain mappings between DTD identifiers and the namespace and location of the XML schema that is automatically generated from the DTD.

Each group directory might contain an NST file.

An excerpt of an NST file

```
<namespaceTable
xmlns="http://schemas.ibm.com/db2/dxx/nst">
<mapping dtdid="c:\dxx\samples\dtd\getstart.dtd"
    namespace="http://schemas.ibm.com/db2/dxx/samples/dtd/getstart.dtd"
    location="/dxx/samples/dtd/getstart.dtd/XSD"/>
<mapping dtdid="getstart.dtd"
    namespace="http://schemas.myco.com/sales/getstart.dtd"
    location="/getstart.dtd/XSD"/>
```

The DADX environment checker first validates NST files for correct schema in the nst.xsd file. Here is an example of a validation error reported by the checker:

```
ERROR. Validation error, in
"file:///c:/tomcat/webapps/services/WEB-
INF/classes/groups/dxx_sales_db/sales_db.nst",
line 8, column 35. cvc-complex-type.2.4.a:
Invalid content starting with element 'mappin'.
The content must match
'("http://schemas.ibm.com/db2/dxx/nst":mapping){0-UNBOUNDED}'.
ERROR. Validation error, in
"file:///c:/tomcat/webapps/services/WEB-
INF/classes/groups/dxx_sales_db/sales_db.nst",
line 17, column 32. cvc-complex-type.4: Attribute 'dtdid'
must appear on element 'mapping'.
ERROR. Validation error, in
"file:///c:/tomcat/webapps/services/WEB-
INF/classes/groups/dxx_sales_db/sales_db.nst",
line 17, column 32. Duplicate unique value
[ID Value: /order.dtd/XSD] declared for identity constraint
of element "namespaceTable".
```

The checker checks that the <mapping> elements have the following dtdid attributes:

- A correct path in the file system.
- A value stored in column DTDID in the db2xml.DTD_REF table.

The following example shows the results of the checks on the <mapping> elements of an NST file:

```
INFO. Line 5: file
"c:\dxx\samples\dtd\getstart.dtd" is accessible.
ERROR. Line 14: file
"wrongDtd.dtd" CANNOT be found either in
the file system or in the database.
```

Checking errors in the DAD files

The Document Access Definition (DAD) file is an XML file that is supported in DB2 XML Extender. The DAD associates XML documents to DB2 tables through two alternative access and storage methods: XML columns and XML collections.

An excerpt of the start of a DAD file

```
<?xml
version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd">
<DAD>
<dtdid>c:\dxx\samples\dtd\getstart.dtd</dtdid>
<validation>NO</validation>
<Xcollection>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE Order SYSTEM
 "c:\dxx\samples\dtd\getstart.dtd"
```

```
</doctype>
<root_node>
<element_node name="Order">
...
```

Process of checking the DAD

1. The DADX environment checker checks that the DAD file is valid according to its DTD dad.dtd. You must ensure that the path to dad.dtd specified in the DOCTYPE declaration of the DAD is correct.
2. The checker gets the value of the <dtdid> tag if it is present. If the value of this tag does not match a value stored in column DTDID in the db2xml.DTD_REF table, then the checker issues a warning. If the <validation> tag in the DAD contains a value of YES, then the checker issues an error message:

   ```
   ## Checking DAD file:
   c:\tomcat\webapps\services\WEB-INF
   \classes\groups\dxx_sales_db\order.dad
   ERROR. Line 4: DTDID "wrongDtd.dtd"
   CANNOT be found in the DTD_REF table.
   ```
3. The checker determines whether the DAD file declares an Xcollection or an Xcolumn. If it declares an Xcollection, the DTD specified in the <doctype> element is extracted. The DADX environment checker checks that this DTD is declared in the NST file.

Results of checking a DAD

The following example shows the results of the checks of an Xcolumn and an Xcollection DAD belonging to the same group:

```
## Checking DAD file:
c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sales_db\getstart_xcolumn.dad
INFO. Line 4: DTDID "getstart.dtd" was found in the DTD_REF table.

## Checking DAD file: c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sales_db\order-public.dad
INFO. Line 4: DTDID "order.dtd" was found in the DTD_REF table.
ERROR. Line 8: the DTDID "order.dtd" has NOT been
declared in the NST file.
```

You can perform other checks on the DAD files by using the DAD checker. The DAD checker is a separate tool that is also contained in the tools\lib directory in dxxworf.zip. For more information, see the documentation on dadchecker tool at the WebSphere Application Development Web site.

Checking errors in the DADX files

Document Access Definition Extension (DADX) is a technology for rapidly creating Web services that access databases. DADX lets you define Web service operations using the standard SQL statements SELECT, INSERT, UPDATE, DELETE, and CALL, and the DB2 XML Extender stored procedures.

An excerpt of a DADX file

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<operation name="find">
<documentation >
Returns the parts from order #1 with price > 20000.
    </documentation>
<retrieveXML>
<DAD_ref>getstart_xcollection.dad</DAD_ref>
```

```
<no_override/>
</retrieveXML>
</operation>
<operation name="findByMinPrice">
<retrieveXML>
<collection_name>
getstart_xcollection.dad
</collection_name>
<no_override/>
<parameter name="minprice"
type="xsd:decrimal"/>
</retrieveXML>
</operation>
```

Process of checking a DADX file

1. The DADX environment checker validates the DADX file according to its schema, dadx.xsd.
2. The checker gets the values of the <DAD_REF> or <collection_name> tags and it checks that the values of these tags are:
   - For <DAD_REF> tags, a correct path to a DAD file in the file system.
   - For <collection_name>, the name of an enabled collection, which is a value stored in column COL_NAME from table db2xml.xml_usage.

Results of checking a DADX file

The following example shows the results of the checks performed on a DADX file:

```
## Checking DADX file: c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sales_db\PartOrders.dadx
ERROR. Validation error, in "file:///c:/tomcat/webapps/services/WEB-
INF/classes/groups/dxx_sales_db/PartOrders.dadx",
line 8, column 67. cvc-complex-type.2.4.c:
The matching wildcard is strict, but no declaration
can be found for element 'as'.
INFO. Line 16: for operation "find",
DAD "getstart_xcollection.dad" was found.
ERROR. Line 26: for operation "findAll",
DAD "non_existing_dad.dad" was NOT found.
INFO. Line 44: for operation "findByColor",
DAD "getstart_xcollection.dad" was found.
INFO. Line 65: for operation "findByMinPrice",
DAD "getstart_xcollection.dad" was found.
```

If an <operation> tag has no <DAD_REF> or <collection_name> tag as a child, the checker issues a message indicating that no check was performed for this particular operation, as shown in the following example:

```
##
Checking DADX file: c:\tomcat\webapps\services\WEB-
INF\classes\groups\dxx_sample\HelloSample.dadx
INFO. Line 10: no <DAD_ref> or <collection_name>
elements to check for operation "listDepartments".
```

Checks for deserializer

The DADX environment checker also checks if WORF will be able to find a deserializer for the parameters declared in the DADX file. A deserializer reconstructs XML messages received across a network connection into the specified variable or object. For every <parameter> tag, the value of its type attribute must be a type that can be deserialized. If no deserializer can be found for a particular type, the checker provides an error message as shown in the following example:

```
ERROR. Line 13: no deserializer was found
to deserialize a
 "http://www.w3.org/2001/XMLSchema:ssstring", using encoding
"http://schemas.xmlsoap.org/soap/encoding/".
```

Chapter 8. WebSphere MQ and DB2 User Defined Functions

Use DB2® and WebSphere® MQ to create SQL requests, develop stored procedures, extend the database with user-defined functions, and turn database requests into Web services.

A set of DB2 User Defined Functions (UDFs) allows direct integration with WebSphere MQ messaging functions. Users can imbed MQ UDFs in their SQL statements to send, receive, and publish or subscribe messages from or to columns in tables. The IBM® WebSphere® MQ messaging functions speed implementation of distributed applications by simplifying application development and testing, and by using a consistent interface across all platforms.

IBM® WebSphere MQ is used for dynamic integration. It connects applications through a simple consistent interface or noninvasive adapters on a variety of platforms across all of the major networking systems. WebSphere MQ allows systems to operate independently, but assures delivery of information. It includes message encryption through a Secure Sockets Layer (SSL) for extra security and enhanced performance. Because of its reliability and robustness, you can use WebSphere MQ in mission-critical, high-value solutions across all industries today.

Messaging, queuing, and publishing and subscribing are common technologies within database application environments. These techniques help link together disparate applications, disseminate real-time information and integrate data and communication within the enterprise.

WebSphere MQ is a message handling system that enables applications to communicate in a distributed environment across different operating systems and networks. WebSphere MQ handles the communication from one program to another by using application programming interfaces (APIs).

Use the WebSphere messaging facilities to receive, process, and store information. Then use DB2 to coordinate the collection and process the message data. WebSphere Federation Server acts as the source and destination for XML information that is processed by WebSphere MQ Integrator.

WebSphere MQ provides support for applications with a number of application programming interfaces:

Message Queuing Interface
Message queuing is one method of program-to-program communication. Message Queuing allows programs to send and receive application-specific data without direct connections. Programs communicate by sending or retrieving messages to or from named queues. Programs do not need to know the location of the named queues. You can replicate programs for availability or performance. You can relocate programs or queues.

Java™ Messaging Services
The Java Messaging Services application programming interfaces allow applications to create, send, receive, and read messages. They enable asynchronous and reliable communications.

WebSphere MQ messaging interface

The WebSphere MQ messaging interface helps you establish and maintain services and policies and the related attributes of messages.

You can connect to WebSphere MQ, the transport layer, by using a WebSphere MQ server. The messaging interface communicates with the application programs by using MQ user-defined functions. The messaging interface provides access to the WebSphere MQ server and the messaging functions through the DB2 database objects. The functions that are required in a particular installation are defined by using services and policies.

Sender, receiver, publisher, and subscriber objects are services of the MQ messaging interface.

The DB2MQ configuration tables for the MQ messaging interface are loaded by a system administrator. The configuration tables define various types of MQ services and policies, as well as other messaging objects.

Sending and receiving messages

You can use the MQ messaging interface to send and receive messages in a number of different ways:

- Send and forget (datagram), where no reply is needed
- Request and response, where a sending application needs a response to the request message
- Publish and subscribe, where a broker manages the distribution of messages

Interoperability

The MQ messaging interface is interoperable with other WebSphere MQ interfaces. By using the MQ messaging interface, you can exchange messages with one or more of the following:

- Another application that is using the messaging interface.
- Any application that is using the message queue interface (MQI).
- A message broker (such as WebSphere MQ Publish/Subscribe or WebSphere MQ Integrator).

Message handling and the MQ messaging interface

The MQ messaging interface is used with the user defined functions in SQL statements to allow you to combine DB2 access with WebSphere MQ message handling.

Message handling

The WebSphere MQ message handling system takes a piece of information (the message) and sends it to its destination. WebSphere MQ guarantees delivery despite any network disruptions that might occur.

Applications programmers use the MQ messaging interface to send messages and to receive messages. The three components in the MQ messaging interface are the message, the service, and the policy:

- The *message* defines **what** one program sends to another

- The *service* defines **where** the message is going to or coming from
- The *policy* defines **how** to handle the message

To send a message that uses the MQ messaging interface, an application must specify the message data, the service, and the policy. A system administrator defines the WebSphere MQ configuration that is required for a particular installation. DB2 provides the default service and default policy, DB2.DEFAULT.SERVICE and DB2.DEFAULT.POLICY, that application programmers can use to simplify their programs.

WebSphere MQ messages

WebSphere MQ uses messages to pass information between applications. Messages consist of the following parts:

- The message attributes, which identify the message and its properties. The MQ messaging interface uses the attributes and the policy to interpret and construct WebSphere MQ headers and message descriptors.
- The message data, which is the application data that is carried in the message. The messaging interface does not act on this data.

Attributes are properties of a WebSphere MQ message. With the MQ messaging interface, the message can contain the attributes, or a system administrator can define the attributes in a default policy. The application programmer is not concerned with the details of message attributes.

WebSphere MQ services

A service describes a destination to which an application sends messages or from which an application receives messages. WebSphere MQ calls a destination a message queue, and a queue resides in a queue manager.

Applications can put messages on queues or get messages from them by using the services and policies defined by the configuration tables. A system administrator sets up the parameters for managing a queue, which the service defines. Therefore, the WebSphere Federation Server messaging interface hides the complexity from the application programmer. An application program selects a service by specifying it as a parameter for WebSphere MQ function calls.

WebSphere MQ policies

A policy controls how the MQ messaging functions handle messages. Policies control such items as:

- The attributes of the message, for example, the priority
- Options for send and receive operations, for example, whether an operation is part of a unit of work

DB2 and WebSphere Federation Server provide the default policies. Alternatively, a system administrator can define customized policies and store them in a set of DB2 configuration tables. An application program can specify a policy as a parameter for WebSphere MQ function calls.

Installing and using the DB2 WebSphere MQ functions

The DB2 WebSphere MQ functions are available in DB2 Version 9.1 for Linux, UNIX, and Windows as user-defined functions. By using these functions, users can access the WebSphere MQ queues from DB2 database objects.

Before you begin

1. Install DB2 Version 9.1.
2. Install WebSphere MQ Server Version 6 or a later edition on the same machine as the DB2 server.
3. Ensure that the owner of the DB2 instance and the owner of the db2fmp process are in the mqm group.
4. On Solaris, open a command prompt, and type: ulimit -n 1024. This command sets the limit for the number of files that are opened and allows you to create a queue manager for Solaris.
5. If you use the WebSphere MQ user defined functions with the db2mq1c schema and you want to enable the functions for transactional context, you must issue the following command:
   ```
   update dbm cfg using federated yes
   ```
6. Install DB2 XML Extender, a component of DB2 Version 9.1, if you want to use the MQ XML functions.
7. If you choose not to use the DB2 default queue manager and queue, create the message queues and WebSphere MQ objects by using the WebSphere MQ script commands (MQSC) that are provided by WebSphere MQ and DB2.
8. Edit the queue information in the amtsamp.tst and amtsdfts.tst sample files to help you create the appropriate objects.
9. Define the queues for the target queue Manager with the runmqsc command, as in the following example:
   ```
   runmqsc QMName <amtsamp.tst
   ```

   You can use the message queues and objects in SQL statements only after they are created.

Restrictions

- The DB2 Version 9.1 MQ transactional functions that exist under the schema db2mq1c do not support CLOB type messages.
- The enable utility of the transactional MQ user-defined functions allows only 40 corresponding policies to exist in the MQPolicy table for the queue manager specified with the **-q** option. The **-q** option only applies to MQ user defined functions under schema db2mq1c. If you want to use a queue manager other than DB2MQ_DEFAULT_MQM, then you must create the queue manager.
- The transactional MQ user-defined functions support only one Queue Manager within a single transaction. The queue manager that you specify in Service and Policy must match. If you leave the Queue Manager blank in the service point, WebSphere MQ defaults to the manager designated by Policy. There is a default set of MQ queues and a default Queue Manager that is normally created during the MQ installation and the enable_MQFunctions processes.
- The Queue name and the Queue Manager name in the configuration tables do not support a double byte character set or a Unicode character set.
- MQ user-defined functions are not supported on a multiple processor partition (MPP) or DB2 Database Partitioning Feature (DPF) environment.

About this task

You use the commands, enable_MQFunctions and disable_MQFunctions, for transactional and nontransactional MQ user-defined functions. The MQ user-defined functions are defined as a group or set under different schema names. The groups that do not support transactions have the schema db2mq. The groups that do support transactions have the schema db2mq1c. The enable_MQFunctions command with the options that support transactions, allows you to select a set of MQ user-defined functions to install or uninstall for transactional support. If you set a value in the **-v** parameter, you specify the type of schema that the enable_MQFunctions command creates. The possible values are **all**, **0pc**, or **1pc**. If you specify **all**, then the enablement creates all schemas under user-defined functions (db2mq, db2mq1c).

If you do not specify a value in **-v** parameter, the enablement defaults to the **all** option.

Procedure

To configure the DB2 WebSphere MQ functions:

1. Connect to the database on which you want to enable MQ functions. For example, if you are working in the SAMPLE database, issue the following command:
   ```
   db2 connect to sample
   ```
2. Run the amtsetup.sql script on the database if this is the first time you are trying to enable the database, or if you want to drop the existing configuration tables and start a new configuration. If this is the first time you are enabling the database there are no configuration tables that exist for this database, so you can ignore the errors from the drop table statements.
   a. Change your current directory to sqllib/cfg/mq.
   b. From the mq directory type `db2 -tvf amtsetup.sql`.
3. Configure and enable a database for the WebSphere MQ functions. The enable_MQFunctions command checks that you have properly set up the WebSphere MQ environment. It then installs and creates a default configuration for the WebSphere MQ functions. Then, it enables the specified database with these functions, and confirms that the configuration works.
   a. Enable the transactional and nontransactional user-defined functions. This example assumes that the user is connected to the SAMPLE database.
      ```
      enable_MQFunctions -n sample -u user1 -p password1
      ```
   b. Create DB2MQ1C functions under the schema DB2MQ1C. This example assumes that the user is connected to the SAMPLE database. The value **1pc** in the -v parameter means that you want to create the db2mq1c schema:
      ```
      enable_MQFunctions -n sample -u user1 -p password1 -v 1pc
      ```
4. Test the MQ functions by using the command line processor on a Windows environment. Issue the following commands after you connect to the currently enabled database:

Command	Description
values DB2MQ1C.MQSEND('a test')	Sends the message `a test` to the DB2MQ_DEFAULT_Q queue. You can use this statement in a DB2 transaction that you can commit or roll back as part of the unit of work.

Command	Description
values DB2MQ1C.MQRECEIVE()	Receives the message back. The statement assumes that you have used some default configuration. You can use this statement in a DB2 transaction that you can commit or roll back as part of the unit of work.

Capabilities of DB2 WebSphere MQ functions

Use the DB2 WebSphere MQ functions to send messages to a message queue or to receive messages from the message queue. In addition, you can send a request to a message queue and receive a response.

You can use WebSphere MQ functions with DB2:
- User-defined functions with no transactional semantics (schema name is DB2MQ)
- User-defined functions that use single-phase commit semantics (schema name is DB2MQ1C)

The schema name indicates the type of user-defined function.

Operations

The DB2 WebSphere MQ functions support the following types of operations:

Send and forget
The messages do not need reply.

Read
The application can read one or all messages without removing them from the queue.

Receive
The application can receive and remove one or all messages from the queue.

Request and response
A sending application needs a response to a request.

Using the messaging operations in SQL statements

By using DB2 scalar and table functions along with views with the federated server, you can incorporate message handling operations in SQL queries from any environment. If you have a WebSphere MQ client or server, you can use the messaging operations within SQL statements. For example:

```
SELECT DB2MQ1C.MQSend ('MyAddress'|| firstname ||' '|| lastname)
  FROM employee
```

DB2 WebSphere MQ scalar functions

The DB2 WebSphere MQ scalar include the following functions:

MQREAD
Returns a message in a VARCHAR variable from the WebSphere MQ location that is specified by *receive-service*, by using the policy that is defined in *service-policy*. This operation does not remove the message from the head of the queue but instead returns it. If no messages are available to be returned, a null value is returned.

```
►►──MQREAD──(──┬──────────────────────────────────────┬──)──────────►◄
               └─receive-service──┬──────────────────┬┘
                                  └─,──service-policy─┘
```

MQRECEIVE

Returns a message in a VARCHAR variable from the WebSphere MQ location that is specified by *receive-service*, by using the policy that is defined in *service-policy*. This operation removes the message from the queue. If *correlation-id* is specified, the first message with a matching correlation identifier is returned; if *correlation-id* is not specified, the message at the head of queue is returned. If no messages are available to be returned, a null value is returned.

```
►►──MQRECEIVE──────────────────────────────────────────────────────►

►─(──┬────────────────────────────────────────────────────────┬──)──────►◄
     └─receive-service──┬────────────────────────────────────┬┘
                        └─,──service-policy──┬──────────────┬┘
                                             └─,──correl-id─┘
```

MQSEND

Sends the data in a VARCHAR variable *msg-data* to the WebSphere MQ location that is specified by *send-service*, by using the policy that is defined in *service-policy*. An optional user-defined message correlation identifier can be specified by *correlation-id*. The return value is 1 if successful or 0 if not successful.

```
►►──MQSEND──(──┬──────────────────────────────────────────┬──msg-data──────►
               └─send-service──,──┬──────────────────────┬┘
                                  └─service-policy──,────┘

►──┬────────────────────┬──)────────────────────────────────────────►◄
   │              (1)   │
   └─,──correl-id───────┘
```

Notes:

1 The *correl-id* cannot be specified unless a *service* and a *policy* are also specified. It is used to associate a request message with a response message.

You can send or receive messages in VARCHAR variables for the schemas DB2MQ and DB2MQ1C. The maximum length for a DB2MQ message and a DB2MQ1C message in a VARCHAR variable is 32000 bytes long.

DB2 WebSphere MQ table functions

The WebSphere MQ table functions includes the following functions:

MQREADALL

Returns a table that contains the messages and message metadata in VARCHAR variables from the WebSphere MQ location that is specified by *receive-service*, by using the policy that is defined in *service-policy*. This operation does not remove the messages from the queue. If *num-rows* is specified, a maximum of *num-rows* messages is returned; if *num-rows* is not specified, all available messages are returned.

```
►►──MQREADALL──(──┬────────────────────────────────────┬──────────────►
                  └─receive-service─┬─────────────────┬┘
                                    └─,──service-policy─┘

►──┬──────────┬─)─────────────────────────────────────────────────────►◄
   └─num-rows─┘
```

MQRECEIVEALL
: Returns a table that contains the messages and message metadata in VARCHAR variables from the WebSphere MQ location that is specified by *receive-service*, by using the policy that is defined in *service-policy*. This operation removes the messages from the queue. If *correlation-id* is specified, only those messages with a matching correlation identifier are returned; if *correlation-id* is not specified, all available messages are returned. If *num-rows* is specified, a maximum of *num-rows* messages is returned; if *num-rows* is not specified, all available messages are returned.

```
►►──MQRECEIVEALL──(───────────────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────┬──►
   └─receive-service─┬──────────────────────────────────────────┬─┘
                     └─,──service-policy─┬────────────────────┬─┘
                                         └─,──correl-id───────┘

►──┬───────────────────┬─)────────────────────────────────────────────►◄
   └─┬───┬──num-rows───┘
     └─,─┘
```

Sends or receive messages in VARCHAR variables. The maximum length for a message in a DB2MQ variable or a DB2MQ1C variable is a VARCHAR 32 000 bytes. The first column of the result table of a DB2 WebSphere MQ table function contains the message.

Publish and subscribe messages

Publishing and subscribing messages gives you more control over which services can receive messages. Publish and subscribe systems provide a scalable, secure environment in which many subscribers can register to receive messages from multiple publishers. You can use the trigger facility within DB2 to automatically publish messages as part of a trigger invocation.

MQPUBLISH
: Publishes data to WebSphere MQ. This function requires the installation of either WebSphere MQ Publish/Subscribe or WebSphere MQ Integrator.

```
►►──MQPUBLISH──(──┬───────────────────────────────────────────────┬───►
                  └─publisher-service──,──┬─────────────────────┬─┘
                                          └─service-policy──,───┘

►─msg-data──┬────────────────────────────────┬─)──────────────────────►◄
            └─,──topic──┬──────────────────┬─┘
                        │            (1)   │
                        └─,──correl-id─────┘
```

Notes:

1 The *correl-id* cannot be specified unless a *service* and a *policy* are also specified.

MQSUBSCRIBE

Registers interest in WebSphere MQ messages that are published on a specified topic. The subscriber-service specifies a logical destination for messages that match the specified topic. Messages that match a topic are placed on the queue that is defined by subscriber-service. Messages can be read or received through a subsequent call to MQREAD, MQRECEIVE, MQREADALL, or MQRECEIVEALL. This function requires the installation and configuration of an WebSphere MQ-based publish and subscribe system, such as WebSphere MQ Integrator or WebSphere MQ Publish/Subscribe.

```
►►──MQSUBSCRIBE──(──┬──────────────────────────────────────────────┬──────►
                    └─subscriber-service──,──┬──────────────────┬─┘
                                             └─service-policy──,─┘

►─topic──)────────────────────────────────────────────────────────────────►◄
```

MQUNSUBSCRIBE

Unregisters an existing message subscription. The subscriber-service, service-policy, and topic are used to identify which subscription is canceled. This function requires the installation and configuration of an WebSphere MQ-based publish and subscribe system, such as WebSphere MQ Integrator or WebSphere MQ Publish/Subscribe.

```
►►──MQUNSUBSCRIBE──(──┬──────────────────────────────────────────────┬────►
                      └─subscriber-service──,──┬──────────────────┬─┘
                                               └─service-policy──,─┘

►─topic──)────────────────────────────────────────────────────────────────►◄
```

Commit environment for DB2 WebSphere MQ functions

The commit environment for DB2 WebSphere MQ user-defined functions depends on the schema name and the type of connection.

Schema name

DB2 provides these versions of commit when you use WebSphere MQ user-defined functions:

- A non-transactional user-defined function with a schema name of DB2MQ
- A single-phase commit with a schema name of DB2MQ1C

If your application uses non-transactional user-defined functions, any DB2 commit or rollback operations are independent of the WebSphere MQ operations. If you roll back a transaction, the MQ functions do not discard the messages that you sent to a queue within the current unit of work.

In this environment, WebSphere MQ controls its own queue operations. A DB2 commit or rollback does not affect when or if your application adds or deletes messages to or from an WebSphere MQ queue.

A transaction is commonly referred to in DB2 as a unit of work. A unit of work is a recoverable sequence of operations within an application process. It is used by the database manager to ensure that a database is in a consistent state. Any reading from or writing to the database is done within a unit of work. A unit of work starts when the first SQL statement is issued on the database. The application must end the unit of work by issuing either a commit or rollback statement.

Connection type

The commit environment for MQ user-defined functions is also dependent on the type of connection that your application includes. The CONNECT statement establishes a connection between an application process and its server. A type 1 CONNECT statement supports the single database per unit of work (Remote Unit of Work) semantics. A type 2 CONNECT statement supports the multiple databases per unit of work (Application-Directed Distributed Unit of Work) semantics. You can also specify a SYNCPOINT of ONEPHASE. A SYNCPOINT defines how commit or rollbacks are coordinated among multiple database connections. With a SYNCPOINT of ONEPHASE, updates can only occur against one database in the unit of work, and all other databases are read-only.

If your application uses single-phase commit with your data sources, and a transaction is rolled back, the application might discard the message or produce an error. This action can result in an inconsistent state. The rules for single-phase commit with SYNCPOINT=ONEPHASE are as follows:

- Updates are allowed only for one data source
- Messaging functions cannot be combined with other updates

The following table shows the DB2 MQ user-defined function semantics:

Table 23. DB2 MQ user-defined function semantics

Connection type	Single-phase commit (schema name=DB2MQ1c)

Type 1 (ONEPHASE)	`select db2mq1c.mqsend`
	` (e.LASTNAME || ' ' || d.DEPTNAME)`
	` from EMPLOYEE e,`
	`     DEPT d`
	`where e.DEPARTMENT =`
	`   d.DEPTNAME`
	
	An application can select and send; it can update only one data source.

Type 2 (TWOPHASE)	Message functions not allowed.

Configuring the MQ messaging interface

The messaging interface that is used by WebSphere MQ user defined functions is a simple programming interface that provides support for point-to-point messaging and publish and subscribe messaging.

The messaging interface simplifies application development by moving function from the application program into a database structure. The three essential parts of the messaging interface syntax are the service, the policy, and the message. The service defines where to send the message. The policy defines how to send the message. The message is what is sent. The service encapsulates local or remote queues. The policy encapsulates options for the message such as *priority* or *retry*. The message part might contain application message data and attributes such as format or correlation identifiers.

WebSphere MQ configuration parameters

When you configure your MQ user defined functions, the information is maintained in tables that are available in DB2 Version 9.1.

The WebSphere MQ messaging interface is available to your application by using the column values in the DB2 configuration tables. Some of the parameters that are available in the tables including the following parameters:

Sender service
Represents a destination such as a WebSphere MQ queue to which messages are sent.

Receiver service
Represents a source from which messages are received.

Publisher
Contains a sender service where the destination is a publish/subscribe broker.

Subscriber
Contains a sender service to send subscribe and unsubscribe messages to a publish/subscribe broker and a receiver service to receive publications from the broker.

Policy Defines how to handle the message, including items such as priority, persistence, and whether it is included in a unit of work.

All WebSphere MQ user defined functions are implemented by using the configuration table parameters. When you use the user defined functions, the parameters are created automatically and populated as needed with values from the configuration tables. When you configure the message environment, the information is stored in the configuration tables. The configuration tables are part of schema DB2MQ.

MQService

The MQService table maintains various types of service point entries and the associated attributes. There are sample values for most of the attributes.

Table 24. The MQService table with pre-defined values in the table

Column name	Type	Default Value	Allowable Value	Sample value
serviceName	varchar(48) not null, primary key	Not applicable	Not applicable	'DB2.DEFAULT.SERVICE'
queueName	varchar(48) NOT NULL	Not applicable	Not applicable	'DB2MQ_DEFAULT_Q'
queueMgrName	varchar(48) NOT NULL	''	Not applicable	'DB2MQ_DEFAULT_MQM'
ccsid	varchar(6) NOT NULL	''	Not applicable	''
description	varchar(400)	''	Not applicable	'DB2 MQ UDFs default service'

MQPubSub

This table defines the Publisher/Subscriber Service Points for the MQ Publish/Subscribe functions.

Table 25. MQPubSub: pre-defined Publisher/Subscriber Service Points

Column name	Type	Default value	Allowable value	Sample value
PubSubName	varchar(48) not null unique	Not applicable	Not applicable	`'DB2.DEFAULT.SUBSCRIBER'`
broker	varchar(48) not null	Not applicable	Not applicable	`'AMT.SAMPLE.SUBSCRIBER'`
receiver	varchar(48) NOT NULL	`''`	Not applicable	`'AMT.SAMPLE.SUBSCRIBER.RECEIVER'`
type	char(1) NOT NULL	Not applicable	• S (Subscriber Service • P (Publisher Service)	`'S'`
description	varchar(400®)	`''`	Not applicable	`'DB2 MQPublish default Publisher service'`

Note:
- If the **type** is *S*, then a value is required in the **receiver** column.
- The values in the **broker** and the **receiver** columns must refer to an entry in the MQService table.

MQPolicy

The MQPolicy table maintains various policy entries and the associated attributes of the policies.

Table 26. MQPolicy

Column name	Type	Default value	Allowable value	Sample value
policyName	varchar(48) not null, primary key	Not applicable	Not applicable	`'AMT.SAMPLE.POLICY'`
connectionName	varchar(48) NOT NULL	`''.`	Not applicable	`'defaultConnection'`
connectionMode	char(1) NOT NULL	`'L'`	• R (Real) • L (Logical)	`'L'`

Table 26. MQPolicy (continued)

Column name	Type	Default value	Allowable value	Sample value
snd_priority	char(1) NOT NULL	`'T'` (AsTransport)	• ’0’ • ’1’ • ’2’ • ’3’ • ’4’ • ’5’ • ’6’ • ’7’ • ’8’ • ’9’ • ’T’	`'T'`
snd_persistent	char(1) NOT NULL	`'T'` (AsTransport)	• ’Y’ • ’N’ • ’T’	`'T'`
snd_expiry	integer NOT NULL	0 = unlimited.	`> = 0`	0
snd_retrycount	integer NOT NULL	0	`> = 0`	0
snd_retry_interval	integer NOT NULL	1000 milliseconds	`> = 0 milliseconds`	1000 milliseconds
snd_newCorrelID	char(1) NOT NULL	`'N'`	• ’Y’ • ’N’	`'N'`
snd_responseCorrelID	char(1) NOT NULL	`'M'`	• ’M’ (MessageID) • ’C’ (CorrelID)	`'M'`
snd_exceptionAction	char(1) NOT NULL	`'D'`	• ’Q’ (DLQ) • ’D’ (Discard)	`'D'`
snd_reportData	char(1) NOT NULL	`'R'`	• ’R’ (Report) • ’D’ (Report_With_Data) • ’F’ (Report_With_Full_Data)	`'R'`
snd_rtException	char(1) NOT NULL	`'N'`	• ’Y’ • ’N’	`'N'`
snd_rtCOA	char(1) NOT NULL	`'N'`	• ’Y’ • ’N’	`'N'`
snd_rtCOD	char(1) NOT NULL	`'N'`	• ’Y’ • ’N’	`'N'`

Table 26. MQPolicy (continued)

Column name	Type	Default value	Allowable value	Sample value
snd_rtExpiry	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
rcv_waitInterval	integer NOT NULL	30 milliseconds	`> = -1 milliseconds (-1=unlimited)`	30 milliseconds
rcv_convert	char(1 NOT NULL	'Y'	• 'Y' • 'N'	'Y'
rcv_handlePoisonMsg	char(1) NOT NULL	'Y'	• 'Y' • 'N'	'Y'
rcv_rcvTruncatedMsg	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
rcv_openShared	char(1) NOT NULL	'Y'	• 'Y' • 'N'	'Y'
pub_retain	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
pub_othersOnly	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
pub_suppressReg	char(1) NOT NULL	'Y'	• 'Y' • 'N'	'Y'
pub_pubLocal	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
pub_direct	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
pub_anonymous	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
pub_correlasID	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
sub_subLocal	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
sub_NewPubsOnly	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
sub_PubOnReqOnly	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'

Table 26. MQPolicy (continued)

Column name	Type	Default value	Allowable value	Sample value
sub_informIfRet	char(1) NOT NULL	'Y'	• 'Y' • 'N'	'Y'
sub_unsubAll	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
sub_anonymous	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
sub_correlAsID	char(1) NOT NULL	'N'	• 'Y' • 'N'	'N'
Description	varchar (400)	''	Not applicable	''

MQHost

This table links the connectionName value that is used in the MQPolicy table to the actual queue manager when the connectionMode has a value of "L", which means Logic.

Table 27. MQHost with pre-defined values

Column name	Type	Default value	Allowable value	Sample value
connectionName	varchar(48) not null unique	Not applicable	Not applicable	`'defaultConnection'`
queueMgrName	varchar(48) not null	''	Not applicable	`'DB2MQ_DEFAULT_MQM'`

WebSphere MQ function messages

WebSphere MQ provides the message transport. The messages define what is sent. Information is passed between communicating applications by using messages.

Messages consist of attributes and data:

- The message attributes identify the message and its properties. The message interface uses the attributes and the information in the policy to interpret and construct WebSphere MQ headers and message descriptors. Attributes are properties of the message object. An application can set the attributes before sending a message or access the attributes after receiving a message. The attributes can be defined in a policy that is created by the system administrator.
- The message data is carried in the message. The size of the message can be VARCHAR(32000) or CLOB(1 MB) depending on the function name or the data type that is provided as the function parameter.

Examples of message attributes

The following list contains message attribute examples:

MessageID
An identifier for the message. It is usually unique, and typically it is generated by WebSphere MQ acting as the message transport.

CorrelID
A correlation identifier that can be used as a key to correlate a response message to a request message.

Format
The structure of the message.

Topic Indicates the content of the message for publish or subscribe applications.

WebSphere MQ messaging Services

A service represents a destination that applications send messages to or receive messages from.

In WebSphere MQ a destination is called a message queue. A queue resides in a queue manager. The parameters that describe the messages on a queue are defined in a service by the systems administrator.

Point-to-point application

In a point-to-point application, the sending application knows the destination of the message. Point-to-point applications can be send and forget, or datagrams, in which a reply to the message is not required. Point-to-point applications can be request or response messages, where the message specifies the destination for the response message. For example, the correlation identifier in the Table 26 on page 164 configuration table, is used for a request/reply type of application in MQ user defined functions

Publish or subscribe application

In a publish/subscribe application, the providers of information (publishers) are separate from the consumers of that information (subscribers).

Publishers supply information about a subject by sending it to a broker. The subject is identified by a topic. A publisher can publish information on more than one topic, and many publishers can publish information on a particular topic.

Subscribers decide what information might be of interest, and subscribes to the relevant topics by sending a message to the broker. When information is published on one or the topics of interest, the broker sends it to the subscriber.

There can be many brokers in a publish or subscribe system. The brokers communicate with each other to exchange subscription requests and publications. A publication is propagated to another broker if a subscription to that topic exists on the other broker.

Types of service

Different types of service are defined to specify the mapping from the messaging interface to resources in the messaging network.

- Senders and receivers establish one-way communication pipes for sending and receiving messages.

- A publisher contains a sender that is used to publish messages to a publish or subscribe broker.
- A subscriber contains a sender, that is used to subscribe to a publish or subscribe broker, and a receiver, that is used to receive publications from the broker.

The messaging interface provides default services that are used unless otherwise specified by the application program. You can also define your own service that is customized and stored in the configuration tables when calling a function. Many of the options used by the services are contained in a policy.

Messaging Policies

A policy controls how the messaging interface functions operate.

Policies control some of the following items:

- The attributes of the message, such as the priority.
- Options for send and receive operations, such as whether an operation is part of a unit of work.
- Publish or subscribe options, such as whether a publication is retained.
- Added value functions that can be invoked as part of the call, such as retry.

The messaging interface provides default policies. A systems administrator can also define customized policies and store them in the configuration tables. An application program selects a policy by specifying it as a parameter on calls.

You can choose to use a different policy on each call and specify in the policy only those parameters that are relevant to the particular call. You can then have policies shared between applications, such as a `Transactional_Persistent_Put` policy. Another approach is to have policies that specify all the parameters for all the calls made in a particular application, such as a `Payroll_Client` policy. Both approaches are valid with the messaging interface, but a single policy for each application will simplify management of policies. The messaging interface will automatically retry when temporary errors are encountered on sending a message, if requested by the policy. (Examples of temporary errors are queue full, queue disabled, and queue in use.)

Service points

Sender and receiver definitions are represented in the messaging interface configuration by a single definition called a service point.

Table 28. Sender and receiver service points

Attribute	Comments
Queue Name	Mandatory name of the queue representing the service that messages are sent to or received from.
Queue Manager Name	Name of the queue manager that owns the Queue. If blank, the system default queue manager is used.

Table 28. Sender and receiver service points (continued)

Attribute	Comments
CCSID	Coded character set identifier of the destination application. Can be read by sending applications in order to prepare a message in the correct CCSID for the destination. It is not used by the messaging interface. Leave blank if the CCSID is unknown (the default), or set to the CCSID number.

Table 29. Subscriber service points

Attribute	Comments
Name	Mandatory name.
Sender Service	The name of the sender service that defines the publish/subscribe broker. It must be a valid service point name.
Receiver Service	The name of the receiver service that defines where publication messages are to be sent. It must be a valid service point name.

Table 30. Publisher service points

Attribute	Comments
Name	Mandatory name, specified on AMI calls.
Sender Service	The name of a sender service that defines the publish/subscribe broker. It must be a valid service point name.

Policy definitions

Policy definitions are available for attributes such as initialization, send, receive, subscribe, and publish.

Table 31. Initialization attributes

Attribute	Comments
Name	Mandatory policy name, specified on messaging interface calls.
Connection Name	If Connection Mode is set to 'Real', Connection Name is the name of the queue manager the application will connect to. If blank, the default local queue manager is used. If Connection Mode is 'Logical', the Connection Name attribute is required and is the name of the logical connection that is defined in the MQHost table to generate the queue manager to which connection is made.

Table 31. Initialization attributes (continued)

Attribute	Comments
Connection Mode	If Connection Mode is set to 'Real' (the default), Connection Name is used as the queue manager name for connection. If Connection Mode is set to 'Logical', Connection Name is used as a key to the host file on the system where the application is running that maps Connection Name to a queue manager name. This allows applications running on different systems in the network to use the same onnection name to connect to different local queue managers.

Table 32. Send attributes

Attribute	Values	Default	Comments
Priority	• 0-9 • T as Transport	T as Transport	The priority set in the message, where 0 is the lowest priority and 9 is the highest. When set to T as Transport, the value from the queue definition is used. You must deselect As Transport before you can set a priority value.
Persistence	• Yes • No • T as Transport	T as Transport	The persistence set in the message, where Yes is persistent and No is not persistent. When set to T as Transport, the value from the underlying queue definition is used.
Expiry Interval	• 0-999999999 • Unlimited	Unlimited	A period of time (in tenths of a second) after which the message will not be delivered.
Retry Count	0-999999999	0	The number of times a send will be retried if the return code gives a temporary error. Retry is attempted under the following conditions: Queue full, Queue disabled for put, Queue in use.
Retry Interval	0-999999999	1000	The interval (in milliseconds) between each retry.

Table 32. Send attributes (continued)

Attribute	Values	Default	Comments
New Correl Id	• Y • N	N	When yes, each message is sent with a new Correl Id (except for response messages, where this is set to the Message Id or Correl Id of the request message).
Response Correl Id	• ’M’ for Message Id • ’C’ for Correl Id	’M’ for Message Id	The identifier in the Correl Id of a response or report message. This is set to either the Message Id or the Correl Id of the request message, as specified.
Exception Action	• Discard • DLQ	DLQ	Action when a message cannot be delivered. When set to DLQ, the message is sent to the dead-letter queue. When set to Discard, it is discarded.
Report Data	• Report • With Data • With Full Data	Report	The amount of data included in a report message, where Report specifies no data, With Data specifies the first 100 bytes, and With Full Data specifies all data.
Report Type Exception	• Y • N	N	When yes, Exception reports are required.
Report Type COA	• Y • N	N	When yes, Confirm on Arrival reports are required.
Report Type COD	• Y • N	N	When yes, Confirm on Delivery reports are required.
Report Type Expiry	• Y • N	N	When yes, Expiry reports are required.

Table 33. Receive attributes

Attribute	Values	Default	Comments
Wait Interval	• 0-999999999 • Unlimited	Unlimited	A period of time (in milliseconds) that the receive waits for a message to be available.

Table 33. Receive attributes (continued)

Attribute	Values	Default	Comments
Convert	• Y • N	Y	When yes, the message is code page converted by the message transport when received.
Handle Poison Message	• Y • N	Y	When yes, poison message handling is enabled.
Accept Truncated Message	• Y • N	Y	When yes, truncated messages are accepted.
Open Shared	• Y • N	Y	When yes, the queue is opened as a shared queue.
Note: A poison message is one for which the count of the number of times it has been backed-out during a unit of work exceeds the maximum backout limit specified by the underlying WebSphere MQ transport queue object. If poison message handling is enabled during a receive request, the messaging interface handles it as follows: If a poison message is successfully requeued to the backout-requeue queue (specified by the underlying WebSphere MQ transport queue), the message is returned to the application with completion code MQCC_WARNING and reason code MQRC_BACKOUT_LIMIT_ERR. If a poison message requeue attempt (as described earlier) is unsuccessful, the message is returned to the application with completion code MQCC_WARNING and reason code MQRC_BACKOUT_REQUEUE_ERR. If a poison message is part of a message group (and not the only message in the group), no attempt is made to requeue the message. The message is returned to the application with completion code MQCC_WARNING and reason code MQRC_GROUP_BACKOUT_LIMIT_ERR. In all cases, a warning is returned and the message is returned to the application (even if it was successfully queued on the backout-requeue queue). Also, the message does not disappear from the original queue from where it is received, unless the application explicitly performs a commit			

Table 34. Subscribe attributes

Option	Values	Default	Comments
Subscribe Locally	• Y • N	N	When yes, the subscriber is sent publications that were published with the Publish Locally option, at the local broker
New Publications Only	• Y • N	N	When yes, the subscriber is not sent existing retained publications when it registers.
Publish On Request Only	• Y • N	N	When yes, the subscriber is not sent retained publications, unless it requests them by using Request Update.

Table 34. Subscribe attributes (continued)

Option	Values	Default	Comments
Inform If Retained	• Y • N	Y	When yes, the broker informs the subscriber if a publication is retained.
Unsubscribe All	• Y • N	N	When yes, all topics for this subscriber are to be deregistered.
Anonymous Registration	• Y • N	N	When yes, the subscriber registers anonymously.
Use Correl Id As Id	• Y • N	N	When yes, the Correl Id is used by the broker as part of the subscriber's identity.

Table 35. Publish attributes

Option	Values	Default	Comments
Retain	• Y • N	N	When yes, the publication is retained by the broker.
Publish To Others Only	• Y • N	N	When yes, the publication is not sent to the publisher if it has subscribed to the same topic (used for conference-type applications).
Suppress Registration	• Y • N	Y	When yes, implicit registration of the publisher is suppressed. (This attribute is ignored for WebSphere MQ Integrator Version 2.)
Publish Locally	• Y • N	N	When yes, the publication is only sent to subscribers that are local to the broker.
Accept Direct Requests	• Y • N	N	When yes, the publisher should accept direct requests from subscribers.
Anonymous Registration	• Y • N	N	When yes, the publisher registers anonymously.
Use Correl Id As Id	• Y • N	N	When yes, the Correl Id is used by the broker as part of the publisher's identity.

Migrating MQ user defined functions from the repository-based configuration to the table-based configuration

If you deployed MQ user defined functions with the application messaging interface from a release earlier than Version 9, you must convert the configuration to the DB2 table structures.

About this task

The WebSphere MQ user defined functions that are used in WebSphere Federation Server in Version 9 and later rely on a DB2 table structure for the messaging configuration. You must convert your existing configuration to the table-based structures.

Procedure

To migrate from a repository-based configuration to a table-based configuration:

1. Connect to the database. For example, if you are working in the SAMPLE database, issue the following command:
   ```
   db2 connect to sample
   ```
2. Establish the configuration tables with default entries by using the following command:
   ```
   db2 -tvf amtsetup.sql
   ```
3. Verify the values in the tables by opening the WebSphere Version 5 AMI Administration GUI tool, available on Windows, to view the current entries in the application messaging interface repository file amt.xml. You can enter the relevant fields from the Administration GUI tool as column or field values in the tables. Ignore the fields or attributes that are not support in MQ user defined functions.
4. Insert rows into the four MQ configuration tables that are based on the current repository-based configuration in the files amt.xml and amthost.xml.
   a. Insert rows in the MQHOST table from the amthost.xml file that corresponds to the connection name and queue manager name.
   b. Insert rows in the MQPolicy, MQService, and MQPubSub tables from the values in the amt.xml file.
   - Each entry in the Service Points category in the amt.xml file is mapped to a row in the MQService table.
   - Each entry in the Policies category in the amt.xml file is mapped to a row in the MQPolicy table.
   - Each entry in the Subscribers or the Publishers category in the amt.xml file is mapped to a row in the MQPubSub table. A Subscriber flag (S) or a Publisher flag (P) indicates the type of service point.

Examples of MQPUBLISH and MQSUBSCRIBE

If you need more control over which services can receive any particular message, then you need to use the publish and subscribe functions.

An example of simple data publication is when one application notifies other applications about events of interest. The application does this by sending a message to a queue that is monitored by another application. The contents of the message might be either a user-defined string, composed from database columns, or a string-valued function call, or any valid expression that yields a string of the correct type.

Many subscribers can register to receive messages from multiple publishers. You can specify a topic that you can associate with your message. For example, a DB2 application can publish a message to the service point *Weather*. The message is *Sleet*, and the topic is *Austin*.

```
values DB2MQ1C.MQPublish ('Weather Bulletins','Sleet','Austin')
```

This notifies the interested subscribers that the weather in Austin is sleet. Subscribers register an interest in receiving this kind of information with the following statement:

```
values DB2MQ1C.MQSUBSCRIBE('aSubscriber', 'Austin')
```

When the subscriber is no longer interested in subscribing to a particular topic, that subscriber must explicitly unsubscribe by using a statement such as:

```
values DB2MQ1C.MQUNSUBSCRIBE('aSubscriber','Austin')
```

DB2 WebSphere MQ functions as part of the DB2 transaction

You can use DB2 MQ UDFs as part of the DB2 unit of work or transaction in many kinds of DB2 operations.

Multiple connections

Multiple connections describe a scenario where two users connect to the same database. Both of the users execute the DB2 MQ user defined functions. One connection, or user, sends a message. The other connection, or user, receives a message. The second connection does not see the message of the first connection before the first connection commits. The second connection sees the messages of the first connection after the commit. If the first connection issues a roll back, the second connection does not see the message.

Table 36. Two users connecting to the same database

Connection 1	Connection 2
db2 +c // Turn auto commit off	No action
values db2mq1c.mqsend (“test message”)	No action

	`//The connection can not`
	`//see the`
	`//message yet:`
	`values db2mq1c.mqreceive();`

commit;	No action

No action	`//Now the connection`
	`//can see the message:`
	`values db2mq1c.mqreceive();`

Triggers

DB2 MQ user defined functions can be part of a single or a multiple statement BEFORE or AFTER trigger.

```
create table EMPLOYEE
  (NAME VARCHAR(30), LASTNAME VARCHAR(30) NOT NULL PRIMARY KEY);

create trigger AFTER_TEST
    after insert on EMPLOYEE
    referencing NEW as NEWEMP
    for each row mode DB2SQL
    VALUES db2mq.mqsend(newemp.lastname);

insert into EMPLOYEE values ('MORGAN', 'TONG');

create trigger BEFORE_TEST
    no cascade before update of NAME on EMPLOYEE
    referencing NEW as NEWNAME OLD as OLDNAME
    for each row mode db2sql
    values db2mq.mqsend (oldname.lastname);

update EMPLOYEE set NAME = 'RAY';
```

Examples of statements that cannot use DB2 MQ functions

You can integrate the messaging techniques with database operations on most DB2 SQL statements. If a DB2 MQ function results in an error, DB2 automatically rolls the transaction back. Here are some examples of statements that cannot use DB2 MQ functions:

- If a user issues an application savepoint
- If a user tries to use the DB2 MQ user defined functions from within an atomic compound SQL statement

WebSphere MQ functions within DB2

WebSphere® MQ and DB2® message operations combine database operations in a single unit of work as an atomic transaction.

The most basic form of messaging with the DB2 MQ functions occurs when all database applications connect to the same DB2 server. Clients can be local to the database server or distributed in a network environment.

In a simple scenario, client A invokes the MQSEND function to send a user-defined string to the location that is defined by the default service. DB2 runs the WebSphere MQ functions that perform this operation on the database server. At some later time, client B invokes the MQRECEIVE function. This removes the message at the head of the queue that is defined by the default service. The function then returns it to the client. DB2 runs the WebSphere MQ functions that perform this operation on the database server.

Simple messaging

Database clients can use simple messaging in the following ways:

Data collection
: The application receives information in the form of messages from one or more sources. An information source can be any application. The application receives the data from queues and stores the data in database tables for additional processing.

Workload distribution
: The application posts work requests to a queue that is shared by multiple instances of the same application. When an application instance is ready to

perform some work, it receives a message that contains a work request from the head of the queue. Multiple instances of the application can share the workload that is represented by a single queue of pooled requests.

Application signaling
In a situation where several processes collaborate, you can use messages to coordinate their efforts. These messages might contain commands or requests to perform work. For more information about this technique, see Application-to-application connectivity.

Messaging scenario

The following scenario extends basic messaging to incorporate remote messaging. Assume that computer A sends a message to computer B.

1. The DB2 client executes an MQSEND function call and specifies a target service that has been defined to be a remote queue on computer B.
2. The WebSphere MQ functions perform the work to send the message. The WebSphere MQ server on computer A accepts the message. The server guarantees that it will deliver the message to the destination. The service and the current configuration of computer A defines the destination. The server determines that the destination is a queue on computer B. The server then attempts to deliver the message to the WebSphere MQ server on computer B, retrying as needed.
3. The WebSphere MQ server on computer B accepts the message from the server on computer A and places it in the destination queue on computer B.
4. WebSphere MQ client on computer B requests the message at the head of the queue.

When you use MQSEND, you choose what data to send, where to send it, and when to send it. This type of messaging is called *send and forget*. The sender only sends a message, relying on WebSphere MQ to ensure that the message reaches its destination.

Example that uses the DB2MQ schema

The example assumes that automatic commit is off. Therefore, a commit is needed. Without the commit, you might still be holding locks until the end of the transaction. The following CREATE TRIGGER statement sends a message that consists of the first and last names that are inserted into table employees:

```
CREATE TRIGGER T1
AFTER INSERT ON employee REFERENCING new AS newemp
   FOR EACH ROW MODE DB2SQL
  VALUES DB2MQ.MQSEND(newemp.name)
```

Examples that use the DB2MQ1C schema

The following examples use the DB2MQ1C schema for single-phase commit with the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY. All of the examples assume that automatic commit is off. Therefore, a commit is needed. Without the commit, you might still be holding locks until the end of the transaction.

Assume that you have an EMPLOYEE table with VARCHAR columns LASTNAME, FIRSTNAME, and DEPARTMENT. Also assume that automatic commit is turned off. To send a message that contains this information for each employee in DEPARTMENT 5LGA, issue the following SQL SELECT statement:

```
SELECT DB2MQ1C.MQSEND (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT)
  FROM EMPLOYEE WHERE DEPARTMENT = '5LGA';
COMMIT;
```

Message content can be any combination of SQL statements, expressions, functions, and user-specified data. Because this MQSEND function uses DB2 MQ transactional user-defined functions with single-phase commit semantics, the COMMIT statement ensures that the message is added to the WebSphere MQ queue.

The DB2 WebSphere MQ functions allow an application to read or receive messages. The difference between reading and receiving is that reading returns the message at the head of a queue without removing it from the queue. Receiving causes the message to be removed from the queue. A message that is retrieved by using a receive operation can be retrieved only once. A message that is retrieved by using a read operation allows the same message to be retrieved many times.

The following examples use the DB2MQ1C schema for single-phase commit with the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY.

The following SQL SELECT statement reads the message at the head of the queue that is specified by the default service and policy. Assume that automatic commit is turned off.

```
SELECT DB2MQ1C.MQREAD() FROM SYSIBM.SYSDUMMY1;
COMMIT;
```

You invoke the MQREAD function once because SYSIBM.SYSDUMMY1 has only one row. The SELECT statement returns a VARCHAR(32000) string. If no messages are available to be read, the result of the statement is a null value.

The following SQL SELECT statement materializes the contents of a queue as a DB2 table:

```
SELECT T.* FROM TABLE(DB2MQ1C.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue and the metadata about those messages. The queue is defined by the default service. The first column of the result table is the message itself, and the remaining columns contain the metadata. The SELECT statement returns both the messages and the metadata.

To return only the messages, issue the following statement:

```
SELECT T.MSG FROM TABLE(DB2MQ1C.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue and the metadata about those messages. The queue is defined by the default service This SELECT statement returns only the messages.

The following SQL SELECT statement tries to send the message from the queue. Assume that automatic commit is turned off.

```
 SELECT DB2MQ1C.MQSEND(name) FROM employees e;
 ROLLBACK;
```

The ROLLBACK statement means that the message is not actually sent because it is in the same unit of work as the DB2 operation.

Application-to-application connectivity

You typically use application-to-application connectivity to solve the problem of putting together a diverse set of application subsystems.

To facilitate application integration, WebSphere MQ® provides the means to interconnect applications. One common scenario is called *request-and-reply* communication.

The request-and-reply method enables one application to request the services of another application. One way to do this is for the requester to send a message to the service provider to request that some work be performed. When the provider completes the work, the provider might decide to send results, or just a confirmation of completion, back to the requester. Unless the requester waits for a reply before continuing, WebSphere MQ must provide a way to associate the reply with its request.

WebSphere MQ provides a correlation identifier to correlate messages in an exchange between a requester and a provider. The requester marks a message with a known correlation identifier. The provider marks its reply with the same correlation identifier. To retrieve the associated reply, the requester provides that correlation identifier when receiving messages from the queue. The provider returns the first message with a matching correlation identifier to the requester.

Examples that use the DB2MQ1C schema

The following examples use the DB2MQ1C schema for single-phase commit.

The following SQL SELECT statement sends a message consisting of the string `Msg with corr id` to the service, MYSERVICE. The application uses the policy MYPOLICY with a correlation identifier CORRID1:

```
SELECT DB2MQ1C.MQSEND ('MYSERVICE', 'MYPOLICY', 'Msg with corr id', 'CORRID1')
  FROM SYSIBM.SYSDUMMY1;
COMMIT;
```

You invoke the MQSEND function once because SYSIBM.SYSDUMMY1 has only one row. Because this MQSEND uses the DB2MQ1C schema, which is the single-phase commit UDF, the message is part of the DB2® transaction.

The following SQL SELECT statement receives the first message that matches the identifier CORRID1. The application receives the message from the queue that is specified by the service MYSERVICE and uses the policy MYPOLICY:

```
SELECT DB2MQ1C.MQRECEIVE ('MYSERVICE', 'MYPOLICY', 'CORRID1')
  FROM SYSIBM.SYSDUMMY1;
```

The SELECT statement returns a VARCHAR(32000) string. If no messages are available with this correlation identifier, the result of the statement is a null value, and the queue does not change.

You can use WebSphere MQ user-defined functions that are available in XML Extender to pass only XML messages between DB2 and the various WebSphere MQ implementations. First, you enable the database for XML extender. Then, you enable the WebSphere MQ XML Extender functions in the following way:

```
enable_MQXML -n DATABASE -u USER -p PASSWORD
```

WebSphere MQ XML functions

The following table is a brief description of some of the WebSphere MQ XML functions. These functions have a DB2XML database schema. They are not under MQ user defined functional transactional control. These MQ XML functions require a separate enable_MQXML command before they are available.

Table 37. WebSphere MQ XML functions

WebSphere MQ XML Functions	Description
DB2XML.MQSendXML	Send an XML message to the queue.
DB2XML.MQReadXML	A nondestructive read of matching XML message(s) from the queue.
DB2XML.MQReadAllXML	A nondestructive read of all XML messages from the queue
DB2XML.MQReadXMLCLOB	A nondestructive read of matching XML CLOB message(s) from the queue.
DB2XML.MQReadAllXMLCLOB	A nondestructive read of all XML CLOB messages from the queue

Tracing WebSphere MQ problems

The WebSphere MQ includes a trace facility to help identify what is happening when you have a problem. It shows the paths taken when you run your messaging interface program. This trace is in addition to the base DB2 trace that you use to debug problems.

There are three environment variables that you set to control trace:

- DB2MQ_TRACE
- DB2MQ_TRACE_PATH
- DB2MQ_TRACE_LEVEL

If you have tracing switched on, it will slow down the running of your messaging interface program, but it will not affect the performance of your WebSphere MQ environment. When you no longer need a trace file, it is your responsibility to delete it. You must stop your messaging interface program running to change the status of the DB2MQ_TRACE variable. The messaging interface trace environment variable is different than the trace environment variable used within the WebSphere MQ range of products. Within the messaging interface, the trace environment variable turns tracing on. If you set the variable to a string of characters (any string of characters) tracing will remain switched on. It is not until you set the variable to NULL that tracing is turned off.

Commands on Windows

SET DB2MQ_TRACE_PATH=drive:\directory
Sets the trace directory where the trace file will be written.

SET DB2MQ_TRACE_PATH=
Removes the DB2MQ_TRACE_PATH environment variable; the trace file is written to the current working directory (when the messaging interface was started).

SET DB2MQ_TRACE_PATH
Displays the current setting of the trace directory.

SET DB2MQ_TRACE_LEVEL=n
Sets the trace level, where n is an integer from 0 through 9. 0 represents minimal tracing, and 9 represents a fully detailed trace. You can also suffix the value with a + (plus) or - (minus) sign. When the plus sign is suffixed, the trace includes all control block dump information and all informational messages. When the minus sign is suffixed, the trace includes only the entry and exit points in the trace, with no control block information or text output to the trace file.

SET DB2MQ_TRACE_LEVEL=
Removes the DB2MQ_TRACE_LEVEL environment variable. The trace level is set to its default value of 2.

SET DB2MQ_TRACE_LEVEL
Displays the current setting of the trace level.

SET DB2MQ_TRACE=xxxxxxxx
Sets tracing ON by putting one or more characters after the '=' sign. For example: SET DB2MQ_TRACE=yes SET DB2MQ_TRACE=no In both of these examples, tracing will be set ON.

SET DB2MQ_TRACE=
Sets tracing OFF.

SET DB2MQ_TRACE
Displays the contents of the environment variable.

Example of a WebSphere MQ messaging trace

The trace file will have names in the format of DB2Mnnnn.TRC under the directory specified in the DB2MQ_TRACE_PATH.

```
Trace for program ---- <<< DB2M trace >>> ---- started at Mon Feb 6 15:39:07 2006

@(!) <<< *** Code Level is DB2 mqint 2.0 *** >>>
! BuildDate Jan 31 2006
! Trace Level is 9
15:39:07.088
-->xmq_xxxInitialize
---->ObtainSystemCp
! About to go off to xmqGetCodeset
! .... back with .....
! ISO8859-1
! Code page is 819
<----ObtainSystemCp (rc = 0)
<--xmq_xxxInitialize (rc = 0)
-->amSessCreateX
---->amCheckAllBlanks()
<----amCheckAllBlanks() (rc = 0)
---->amCheckValidName()
<----amCheckValidName() (rc = 1)
! Session name is: DB2MQ_RCV_SESSION
---->amIdxTableAddEntry
------>amIdxTableCreate
! allocating 48, 8192
! amIdxTableCreate allocated structure 110139ff0
! amIdxTableCreate allocated array 11014edf0
<------amIdxTableCreate (rc = AM_ERR_OK)
------>amIdxTableLock
<------amIdxTableLock (rc = AM_ERR_OK)
------>amIdxTableUnlock
<------amIdxTableUnlock (rc = AM_ERR_OK)
! Added entry at index 0 to 110139ff0
<----amIdxTableAddEntry (rc = AM_ERR_OK)
---->amSesClearErrorCodes
```

```
------>amIdxTableGetEntry
-------->amIdxTableLock
<--------amIdxTableLock (rc = AM_ERR_OK)
-------->amIdxTableUnlock
<--------amIdxTableUnlock (rc = AM_ERR_OK)
```

Chapter 9. MQListener in WebSphere Federation Server

The WebSphere Federation Server provides an asynchronous listener, named MQListener. MQListener is a framework for tasks that read from WebSphere® MQ queues and call DB2 stored procedures with messages as they arrive.

MQListener combines messaging with database operations. You can configure the MQListener daemon to listen to the WebSphere MQ message queues that you specify in a configuration database. MQListener reads the messages that arrive from the queue and then calls DB2 stored procedures with the messages as input parameters. If the message requires a reply, MQListener creates a reply from the output that is generated by the stored procedure. The message retrieval order is fixed at the highest priority first, and then within the priority the first message in is the first message served.

MQListener runs as a single multi-threaded process. Each thread or task establishes a connection to its configured message queue for input. Each task also connects to a DB2 database on which to run the stored procedure. The information about the queue and the stored procedure is stored in a table in the configuration database. The combination of the queue and the stored procedure is a task.

MQListener tasks are grouped together into named configurations. By default, the configuration name is empty. If you do not specify the name of a configuration for a task, MQListener uses the configuration with an empty name.

MQListener can integrate the message queue read and write operations with the stored procedure into a single transaction. When you run transactional tasks a message cannot be lost even if your computer fails after you read the message from the queue, but before the stored procedure receives the message. By default, only the call to the stored procedure is transactional. If you want to combine into the same transaction the operations of removing the message from the queue and calling the stored procedure, configure the WebSphere MQ environment as a coordinator by using the *–mqcoordinated* parameter with the db2mqlsn command. You must configure the pertinent queue manager to coordinate with the proper resource according to WebSphere MQ guidelines. If you do not want to specify transactional queue operations, the queue manager should not be configured as a transaction manager. Do not run a nontransactional task with a queue manager that is configured as a transaction coordinator.

Configuration

As part of the MQListener configuration, you specify the configuration user (-configUser) and the run user (-dbUser). The **configuration user** and the **run user** can be separate users with different access rights. The run user does not inherit the privileges of the configuration user. In a normal MQListener scenario, a user runs the MQListener application. The only right that the user who runs MQListener requires is the ability to access WebSphere MQ functions, which generally means being a member of the *mqm* group in Windows® and UNIX® operating systems. The user who executes MQListener is typically the configuration user.

Stored procedure interface

The stored procedure interface for MQListener takes the incoming message as input and returns the reply, which might be NULL, as output:

```
schema.proc(in inMsg inMsgType, out outMsg outMsgType)
```

The data type for *inMsgType* and the data type for *outMsgType* can be VARCHAR, VARCHAR FOR BIT DATA, CLOB, or BLOB of any length. The input data type and output data type can be different data types. The number of parameters of the stored procedure is pre-defined, such that if there is one input parameter, there is one output parameter.

Asynchronous messaging in Information Integration

With asynchronous messaging, the program that sends the message proceeds with its processing after sending the message without waiting for a reply.

Programs can communicate with each other by sending data in messages rather than by using constructs like synchronous remote procedure calls. If the program needs information from the reply, the program suspends processing and waits for a reply message. If the messaging programs use an intermediate queue that holds messages, the requester program and the receiver program do not need to be running at the same time. The requester program places a request message on a queue and then exits. The receiver program retrieves the request from the queue and processes the request.

Asynchronous operations require that the service provider is capable of accepting requests from clients without notice. An asynchronous listener is a program that monitors message transporters, such as WebSphere® MQ, and performs actions based on the message type. An asynchronous listener can use WebSphere MQ to receive all messages that are sent to an endpoint. An asynchronous listener can also register a subscription with a publish or subscribe infrastructure to restrict the messages that are received to messages that satisfy specified constraints.

The following examples show some common uses of asynchronous messaging:

Message accumulator
: You can accumulate the messages that are sent asynchronously so that the listener checks for messages and stores those messages automatically in a database. This database, which acts as a message accumulator, can save all messages for a particular endpoint, such as an audit trail. The asynchronous listener can subscribe to a subset of messages, such as *save only high value stock trades*. The message accumulator stores entire messages. The message accumulator does not provide for selection, transformation, or mapping of message contents to database structures. The message accumulator also does not reply to messages.

Message event handler
: The asynchronous event handler listens for messages and invokes the appropriate handler, such as a stored procedure, for the message endpoint. You can call any arbitrary stored procedure. The asynchronous listener lets you select, map, or reformat message contents for insertion into one or more database structures.

Benefits of asynchronous message

The following lists some of the benefits of asynchronous messaging database interactions:

- The client and database do not need to be available at the same time. If the client is available intermittently, or if the client fails between the time that the request is issued and the response is sent, it is still possible for the client to receive the reply. Or, if the client is on a mobile computer and becomes disconnected from the database, and if a response is sent, the client can still receive the reply.
- The content of the messages in the database contain information about when to process particular requests. The messages in the database use priorities and the request contents to determine how to schedule the requests.
- An asynchronous message listener can delegate a request to a different node. It can forward the request to a second computer to complete the processing. When the request is complete, the second computer returns a response directly to the endpoint that is specified in the message.
- An asynchronous listener can respond to a message from a supplied client or from a user-defined application. The number of environments that can act as a database client is greatly expanded. Clients such as factory automation equipment, pervasive devices, or embedded controllers can communicate with DB2® either directly through WebSphere MQ or through some gateway that supports WebSphere MQ.

Configuring and running MQListener

Use this procedure to configure the environment for MQListener and to develop a simple application that receives a message, inserts the message in a table, and creates a simple response message.

Procedure

To configure and run MQListener:

Configuring MQListener to run in the DB2 environment

Configure your database environment so that your applications can use messaging with database operations.

Before you begin

Create a database for the MQListener configuration and a database for the stored procedures that you call when a message arrives (if valid databases are not already available). You can use the same database for the configuration and the stored procedures.

The configuration users must have the following privileges and authorizations:

- Read and write access to the DB2 table SYSMQL.LISTENERS. MQListener run users do not need access to SYSMQL.LISTENERS
- Authority to run the configuration package mqlCOnfig.bnd

The run users must have the authority to run the mqlRun.bnd package.

Procedure

To configure MQListener to run with federated databases:

1. Issue the following command to connect. Substitute the appropriate values for your database environment:
   ```
   db2 connect to ConfigDB user DBAdmin using DBAdminPwd
   ```
2. Run the MQLInstall.sql script, which creates a table that stores the MQListener configuration. The script is in the ...\sqllib\bin directory in a Windows environment:
   ```
   db2 -td; -f MQLInstall.sql
   ```
3. Issue the following commands to grant access to the configuration user. Substitute the appropriate values for your database environment:
   ```
   db2 grant all privileges on table SYSMQL.LISTENERS to ConfigUser
   db2 connect reset
   ```
4. Bind the MQListener packages and grant access to the packages. You must bind the MQLConfig package in the configuration database. Issue the following commands.
   ```
   db2 connect to ConfigDB user DBAdmin using DBAdminPwd
   db2 bind mqlConfig.bnd
   db2 grant execute on package mqlConfig.bnd to ConfigUser
   db2 connect reset
   ```
5. Bind the mqlRun.bnd package in each of the run databases. Issue the following commands for each run database and each run user in that database:
   ```
   db2 connect to RunDB user DBAdmin using DBAdminPwd
   db2 bind mqlRun.bnd
   db2 grant execute on package mqlRun to RunUser
   db2 connect reset
   ```

Configuring WebSphere MQ for MQListener

You can run a simple MQListener application with a simple WebSphere MQ configuration. More complex applications might need a more complex configuration. Configure at least two kinds of WebSphere MQ entities: the queue manager and some local queues. Configure these entities for use in such instances as transaction management, deadletter queue, backout requeue and backout retry threshold.

Before you begin

Issue the WebSphere MQ control commands while in the *mqm* group. The *mqm* group is used by the WebSphere MQ administrators and for internal MQ programs. All members of this group have access to all resources.

Procedure

To configure WebSphere MQ for a simple MQListener application:

1. Create a queue manager.
   ```
   crtmqm TransQM
   ```
2. Start the queue manager.
   ```
   strmqm TransQM
   ```
3. Optional: Configure the queue manager to coordinate transactions with DB2.
   a. Provide the name of a shared library, which is called a switch load file, that WebSphere MQ can use to find the DB2 X/Open resource manager functions, and an extended architecture open string (`xa_open`) that is specific to DB2.

b. Create the MQStart routine in the switch load file by compiling a small C program that returns the DB2 global variable `db2xa_switch`. See the *WebSphere MQ: System Administration Guide* for specific information on how to create the switch load file. MQStart returns a structure of pointers to the functions that implement the X/Open resource manager functions in DB2. The following example shows the required format for the extended architecture open string, with appropriate values that are substituted for MQListener configuration parameters:

```
DB=RunDB, UID=RunUser, PWD=RunUserPwd, TPM=MQ, TOC=P
```

If you use TPM=MQ in the extended architecture open string as in this example, you do not need to set the DB2 TP_MON_NAME instance variable. WebSphere MQ obtains the parameters that it needs based on the operating system environment. On Windows operating systems the parameters are in the Windows registry. You can specify the parameters by using the WebSphere MQ MQServices. On UNIX operating systems the parameters are in the queue manager configuration file. Use a valid text editor for your environment to specify the parameters to use.

4. Create your local queues by using the WebSphere MQ script facility.
   a. Create a file that contains the following commands (for this example the file is mqconfig.mqs):

```
define qlocal('DLQ')
alter qmgr deadq('DLQ')
define qlocal('Backout')
define qlocal('Admin')
define qlocal('In') boqname('Backout') bothresh(3)
define qlocal('SYSTEM.SAMPLE.REPLY')
```

   b. Redirect the mqconfig.mqs file into the script interpreter by issuing the following command:

```
runmqsc TransQM < mqconfig.mqs
```

If you configure the queue manager to coordinate transactions with DB2, then MQListener applications can remove a message and call a stored procedure in a single transaction.

Configuring MQListener

Use the MQListener command, db2mqlsn, to configure MQListener.

Restrictions

- Use the same queue manager for the request queue and the reply queue.
- On Windows systems, each thread can connect to one queue manager.
- On UNIX systems, each process can connect to one queue manager. If you specify different queue managers within the same MQListener configuration on a UNIX system, you receive a run-time error from WebSphere MQ.
- MQListener does not support logical messages that are composed of multiple physical messages. MQListener processes physical messages independently.

About this task

Issue the command db2mqlsn from a command line in any directory. On a Windows system, issue the command in a DB2 command line processor window to insure proper message display. The add parameter with the db2mqlsn command updates a row in the DB2 table SYSMQL.LISTENERS. .

Procedure

To specify MQListener configuration:

1. Add an MQListener configuration with the following command:
   ```
   db2mqlsn add
     -configDB ConfigDB
     -config aConfiguration
     -configUser ConfigUser
     -configPwd ConfigUserPwd
     -queueManager TransQM
     -inputQueue In
     -procSchema RunUser
     -procName aProc
     -dbName RunDB
     -dbUser RunUser
     -dbPwd RunUserPwd
     -waitMillis waitInMilliSeconds
     -mqCoordinated
   ```
2. Display all of the tasks in a configuration with the following command:
   ```
   db2mqlsn show
     -configDB ConfigDB
     -config aConfiguration
     -configUser ConfigUser
     -configPwd ConfigUserPwd
   ```
3. Remove the messaging tasks with the following command:
   ```
   db2mqlsn remove
     -configDB ConfigDB
     -config aConfiguration
     -configUser ConfigUser
     -configPwd ConfigUserPwd
     -queueManager TransQM
     -inputQueue In
   ```
4. Get help with the command and the valid parameters with the following command:
   ```
   db2mqlsn help
   ```
5. Get help for a particular parameter with the following command by using a specific parameter:
   ```
   db2mqlsn help <command>
   ```

Creating a stored procedure to use with MQListener

MQListener uses the stored procedure, aProc, to store a message in a table. The stored procedure returns the string OK if the message is successfully inserted into the table.

Before you begin

The stored procedure requires a C compiler.

Restrictions

About this task

The run database contains the stored procedure that is run when a message arrives. The run user is the user in whose name MQListener connects to the run database to run the stored procedure. Use the following parameters with the db2mqlsn add command to define the run database and the run user:

- -dbName

- -dbUser

The run user must be able to connect to the run database and run the stored procedure. The run user does not need to be the owner of the stored procedure. The run user also does not need access to the MQListener configuration.

Procedure

To create DB2 database objects that you can use with MQListener applications:

1. Create a simple table as the run user (you can use the DB2 command line processor):

   ```
   CREATE TABLE aTable (val VARCHAR(25) CHECK (val NOT LIKE 'fail%'))
   ```

2. Create the following stored procedure:

   ```
   CREATE PROCEDURE aProc (IN pin VARCHAR(25), OUT pout VARCHAR(2))
   BEGIN
           INSERT INTO aTable VALUES(pin);
           SET pout = 'OK';
   END
   ```

The table contains a check constraint so that messages that start with the characters `fail` cannot be inserted into the table. The check constraint is used to demonstrate the behavior of MQListener when the stored procedure fails.

MQListener examples

The examples in this topic show a simple MQListener application. The application receives a message, inserts the message in a table, and generates a simple response message.

To simulate a processing failure, the application includes a check constraint on the table that contains the message. The constraint prevents any string that begins with the characters `fail` from being inserted into the table. If you attempt to insert a message that violates the check constraint, the example application returns an error message and requeues the failing message to the backout queue.

Issue MQListener with all tasks configured

To run MQListener with all of the tasks specified in a configuration, issue the following command:

```
db2mqlsn run
  -configDB ConfigDB
  -config aConfiguration
  -configUser ConfigUser
  -configPwd ConfigUserPwd
  -adminQueue Admin
  -adminQMgr TransQM
```

Using MQListener to send simple messages

The following examples show how to use MQListener to send a simple message and then inspect the results of the message in the WebSphere MQ queue manager and the database. The examples include queries to determine if the input queue contains a message, or if a record is placed in the table by the stored procedure. Many tools support these operations, including the DB2 command line processor, DB2 Command Center, some WebSphere MQ command line utilities, sample programs, the MQ Explorer, and the MQ API exerciser. Consider using DB2 and WebSphere MQ tools for more complex applications.

MQListener example 1: Running a simple application

1. Start with a clean database table:
   ```
   db2 delete from aTable
   ```
2. Send a datagram to the input queue:
   a. Place the string `a sample message` in a file named sampleMsg1.txt
   b. Use the WebSphere MQ sample program amqsput to put the message on the queue:
      ```
      amqsput In TransQM < sampleMsg1.txt
      ```
3. Query the table to verify that the sample message is inserted:
   ```
   db2 select * from aTable
   ```
4. Display the number of messages that remain on the input queue to verify that the message has been removed:
   a. Place the following command in the file checkIn.mqs:
      ```
      display queue('In') curdepth
      ```
   b. Redirect the command into the script interpreter:
      ```
      runmqsc TransQM < checkIn.mqs
      ```

MQListener example 2: Sending requests to the input queue and inspecting the reply

The following example statements send a request to the input queue and inspect the reply:

1. Start with a clean database table:
   ```
   db2 delete from aTable
   ```
2. Send a request to the input queue:
   a. Place the string `another sample message` in a file named sampleMsg2.txt
   b. Use the WebSphere MQ sample program amqsreq to send the request to the input queue:
      ```
      amqsreq In TransQM < sampleMsg2.txt
      ```

      The amqsreq program sets the reply-to queue in the request to SYSTEM.SAMPLE.REPLY
3. Query the table to verify that the sample message is inserted:
   ```
   db2 select * from aTable
   ```
4. Display the number of messages that remain on the input queue to verify that the message is removed.
   ```
   display queue('In') curdepth
   ```
5. Look at the SYSTEM.SAMPLE.REPLY queue for the reply by using the WebSphere MQ sample program amqsget. Verify that the OK string is generated by the stored procedure:
   ```
   amqsget SYSTEM.SAMPLE.REPLY TransQM
   ```

MQListener example 3: Testing an unsuccessful insert operation

If you send a message that starts with the string `fail`, the constraint in the table definition is violated, and the stored procedure fails.

1. Start with a clean database table:
   ```
   db2 delete from aTable
   ```
2. Send a request to the input queue:
   a. Place the string `failing sample message` in a file named sampleMsg3.txt

b. Use the WebSphere MQ sample program amqsreq to send the request to the input queue:

```
amqsreq In TransQM < sampleMsg3.txt
```

The amqsreq program sets the reply-to queue in the request to SYSTEM.SAMPLE.REPLY

3. Query the table to verify that the sample message is not inserted:

```
db2 select * from aTable
```

4. Display the number of messages that remain on the input queue to verify that the message is removed:

```
display queue('In') curdepth
```

5. Read from the SYSTEM.SAMPLE.REPLY queue and find an exception report rather than an OK reply:

```
amqsget SYSTEM.SAMPLE.REPLY  TransQM
```

6. Read from the Backout queue and find the original message:

```
amqsget Backout TransQM
```

Parameters used in MQListener configuration

This topic describes the parameters that you can use in the MQListener configuration.

ConfigDB
: The configuration database, which can be any valid DB2 database, contains an MQListener configuration table. The configuration table contains information about the queues to which MQListener should listen and the stored procedures MQListener should call.

ConfigUser
: The user ID in whose name you access the configuration database. The configuration user does not need to be a database administrator. You can specify the configuration user and password in the MQListener command. If you do not specify a configuration user and password, and your database installation supports implicit connections, by default the configuration user is the user under whose account the MQListener is running.

ConfigUserPwd
: The password that is used with the configuration user ID.

RunDB
: The run database is the database that contains the stored procedures that are run when a message arrives. The stored procedures can be in different databases from the configuration database.

RunUser
: The user in whose name you access the run database to run the stored procedure. The run user does not need any privilege except the ability to connect to the run database and run the stored procedure.

RunUserPwd
: The password that is associated with the run user.

WebSphere MQ queues used in MQListener

This topic describes the typical WebSphere MQ queues that a simple MQListener application uses.

Deadletter queue
The deadletter queue (DLQ) in WebSphere MQ holds messages that cannot be processed. MQListener uses this queue to hold replies that cannot be delivered, for example, because the queue to which the replies should be sent is full. A deadletter queue is useful in any MQ installation especially for recovering messages that are not sent.

Backout queue
For MQListener tasks in which WebSphere MQ is the transaction coordinator, the Backout queue serves a similar purpose to the deadletter queue. MQListener places the original request in the Backout queue after the request is rolled back a specified number of times (called the backout threshold).

Administration queue
The administration queue is used for routing control messages such as *shutdown* and *restart* to MQListener. If you do not supply an administration queue, then the only way to shut down MQListener is to issue a kill command.

Application input and output queues
The application uses input queues and output queues. The application receives messages from the input queue. The application sends replies and exceptions to the output queue. For example, SYSTEM.SAMPLE.REPLY is used in the WebSphere MQ sample program amqsreq.

Chapter 10. Developing applications that use federation

By using the technologies of a federated database environment, you can create applications that solve the problems of data scalability, data accessibility, and data currency. The scenarios describe companies that are composites of several real companies and use actual customer experiences. All of the names in the scenarios are fictitious.

Developing federated application with Java technology

Web-enabled applications that are developed in a database environment can use several components of the Java™ 2 Enterprise Edition (J2EE) server environment. Some of the J2EE components include support for enterprise beans, connections to the database manager, and access to the database manager, which includes support for Java Database Connectivity code and Java transaction application programming interfaces.

Advantages of enterprise beans in a federated system

By using enterprise beans and federated system objects, programmers can perform database operations and access multiple data sources. Programmers can create applications that integrate disparate data through Enterprise JavaBeans technology.

Accessing disparate data sources

Without a federated system, accessing disparate data sources requires multiple steps:

1. You must connect to each data source individually.
2. You must extract the necessary data by using different native application programming interfaces.
3. You must filter, sort, and consolidate the data manually.

A federated system is a type of distributed database management system that makes it possible for you to send distributed requests to multiple data sources by issuing a single SQL statement. With a federated system, you simply query the nicknames of the data sources by using SELECT, INSERT, UPDATE, and DELETE statements.

In a federated system, you have transparent access to data that spans multiple heterogeneous sources. Federated systems complement the built-in database support that is provided by Web application servers and enterprise beans.

Federated views

By using the DB2 view objects, database administrators can create views of data that span multiple tables from different data sources. You begin the process of building a view in a federated system by creating nicknames for the remote data objects. Then you issue an SQL statement that creates a view that joins the nicknames. Even though the views that join multiple data sources are read-only views, you can map container-managed persistence entity beans to these read-only views. The container-managed persistence entity beans are read-only beans.

Materialized query tables

You can incorporate materialized query tables into your application to improve performance. With materialized query tables, you can precompute whole or parts of each query and then use the computed results to answer future queries. Materialized query tables help to avoid redundant scanning, aggregating, and joining of data

With materialized query tables, you can process queries when the remote data source is not available (such as when the network is not available). A materialized query table on a remote table is perceived as a cache for that table, which enhances system availability.

A materialized query table can increase the scalability of the overall system by reducing the work on the primary database. You can use several local databases to perform the work. Each database contains a copy of a subset of the frequently used primary data.

By using materialized query tables, you can avoid some connections to remote systems for some queries. The overall system throughput can potentially increase, and your total response time can decrease.

Enterprise beans in a federated system

Programmers can use enterprise beans and federated system objects to perform database operations or transactional work and access data that spans multiple heterogeneous sources. Federated systems support automated development and deployment of a single container-managed persistence entity bean whose attributes map to data from multiple resources.

Enterprise beans are Java components that run on a Web server. You can create container-managed persistence entity beans and map them to nicknames that you create with federated systems. Federated objects such as wrappers, nicknames, and views extend the usefulness of entity beans to integrate disparate data through Enterprise JavaBean architecture.

The container-managed persistence entity bean can access data that is located in relational databases. The read-only container-managed persistence entity bean can access data that is located in nonrelational databases.

Enterprise JavaBean architecture

Enterprise bean components are part of the Enterprise JavaBean architecture. An enterprise bean implements business logic. For example, the relational and nonrelational tables in Figure 56 on page 197 describe a customer order scheduling system that is implemented by entity beans. The enterprise bean components run in an enterprise JavaBean container. The container runs on an enterprise JavaBean server. The enterprise JavaBean container provides services such as transaction and resource management, persistence, and security to the enterprise bean components. The enterprise JavaBean container controls the details of database manipulation, such as managing access to a target data source.

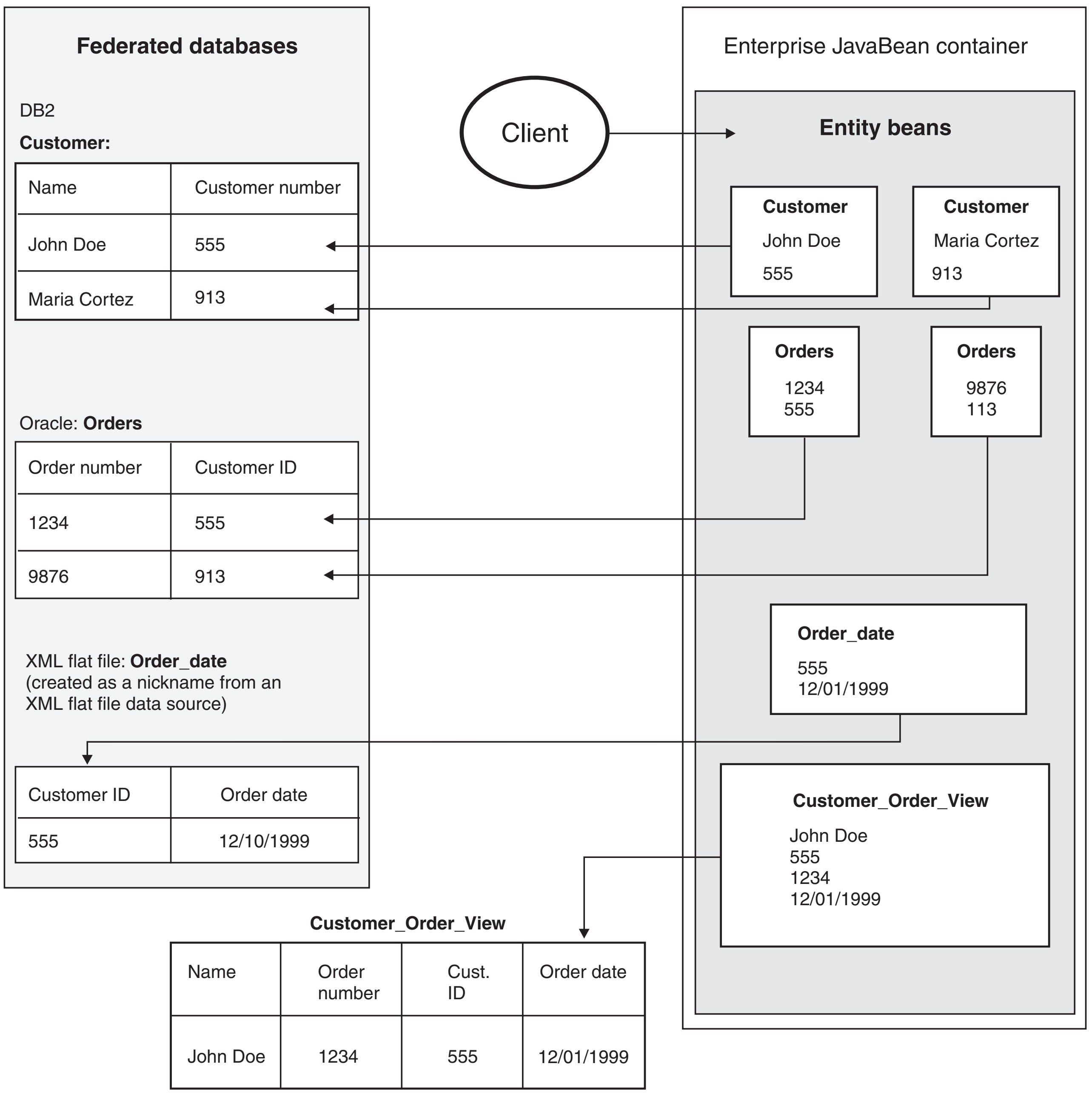


Figure 56. Container-managed entity beans mapped to federated objects

Entity beans and session beans

Two types of enterprise beans are used in a federated system:

Session beans
: Usually associated with a single client and is usually not persistent. The session bean acts as a single client that performs some actions on the server. The session bean does not represent or update existing database contents.

Entity beans
: Represents information that is stored persistently in a database. Entity beans are associated with database transactions. Entity beans can provide

data access to multiple users. An entity bean might represent an underlying database row or the result of a SELECT statement in a single row.

Figure 56 on page 197 shows four container-managed persistence entity beans that map to federated database objects:

`Customer` is a DB2 table.

`Orders` is an Oracle table that is accessed with the nickname `Orders`.

`Order_date` is an XML flat file that is accessed with the nickname `Order_date`.

`Cust_Order_View` is a view that is created by joining the `Customer` table, the `Orders` table, and the `Order_date` table.

The federated server translates the database access to data access requests that are appropriate to the data sources. When you deploy an enterprise bean, the bean resides in containers that provide services such as support for persistence. The entity bean automatically generates the code that implements persistence when you deploy the enterprise bean. By contrast, when you build session enterprise beans that access persistent data, you must write your own Java database connectivity statements to establish database connections and issue SQL statements.

A container-managed persistence entity bean defers all interaction with the database to the enterprise JavaBean container. Typically, the enterprise bean reads the data from the database and places the data into the fields in the container-managed persistence entity bean. You can reference or update (when the data is part of a relational database) the data in the entity bean. When a transaction ends, the Enterprise JavaBean container accesses the data in the entity bean and updates the underlying row in the relational table.

Creating and deploying a container-managed persistence bean

You can create container-managed persistence entity beans that map to federated systems nicknames.

Before you begin

- WebSphere Studio Version 5 or later or Rational Application Developer is required.
- You must register the objects that the bean accesses in your federated system.

About this task

A single container-managed persistence entity bean can span multiple data sources. The entity bean integrates disparate data through standard enterprise JavaBean technology. When you define a container-managed persistence entity bean, you use a deployment descriptor file to control the data source that contains the persistent data and any access restrictions.

Procedure

To create and deploy a container-managed persistence entity bean by using the WebSphere Studio:

1. From the Java 2 Enterprise Edition (J2EE) perspective in WebSphere Studio, create an EJB project for the entity bean.

2. Create the container-managed persistence entity bean:
   a. Name the container-managed persistence entity bean the same name as the nickname that you defined for the data source. For example, Figure 56 on page 197 shows a mapping between the Orders nickname and the Orders entity bean.
   b. Add attributes that correspond to each of the column names in the nickname. Specify the data type for each column that corresponds to the data type in the column of the nickname. Designate one of the attributes as the key field, which must map to the primary key column of the nickname.
3. Generate the DDL for your bean by using the enterprise JavaBean data modeling window:
   a. Select the entity bean that you created.
   b. Right-click the bean and click **Generate** → **EJB to RDB mapping**.
   c. Click **Top-down modeling**.
   d. Click **Next**.
   e. Ensure that you set the database name and the schema names properly. The database name must map to the federated database that is known to your DB2 client. The schema name must map to the authorized federated database user.
   f. Clear the **Generate DDL** check box.
   g. Click **Finish**.
4. Verify that the mapping between the entity bean and the database completed successfully:
   a. Select the entity bean module.
   b. Right-click the bean and click **Open With** → **Mapping Editor**.
   c. Correct any errors on the Tasks window.
5. Bind the entity bean to the data source that you created for the federated database:
   a. Select the entity bean.
   b. Right-click the bean and click **Open With** → **Deployment Descriptor Editor**.
   c. On the Overview page, scroll to WebSphere Bindings. For JNDI-CMP Factory Bindings, specify a valid JNDI name and container authorization type.
   d. Save your modifications and close the window.
6. Generate the code to deploy the entity bean:
   a. Select the entity bean.
   b. Right-click the bean and click **Generate Deploy Code**.

After you package and deploy the entity bean, you can change the properties of the bean by changing the deployment descriptors of the bean. You can assemble your bean with other beans to create applications. Or, you can export your bean from your current development environment and deploy it on another Web application server.

Examples of federated applications

Cottonwood Distributors, Inc. is an existing, well-established distribution company. The company acts as a broker for commodity parts. The company has been in business for many years as a DB2 customer. Cottonwood Distributors programmers can use the federated applications to get customer bid requests and to update quotes from suppliers.

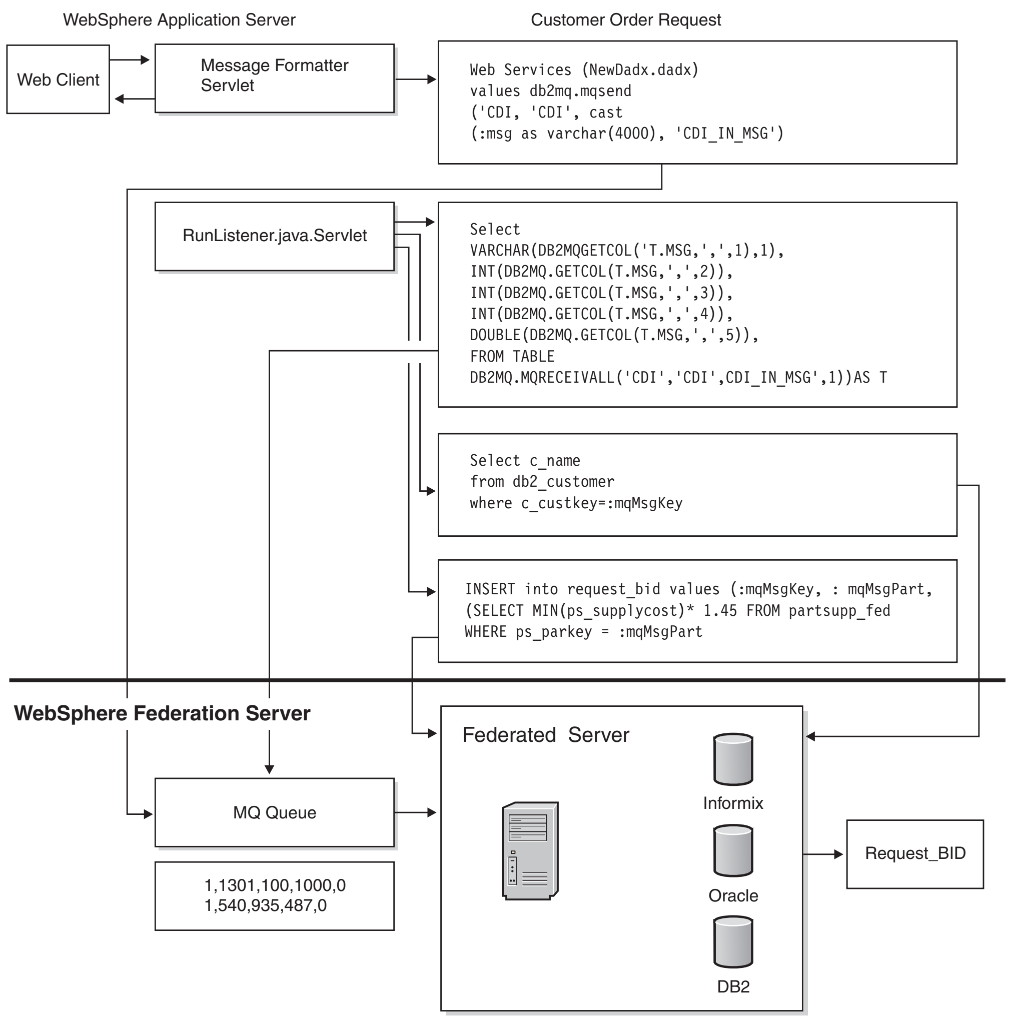
Customer bid requests

The federated application provides the customer bid requests and status of bids that is required of the new merged Cottonwood Distributors enterprise.

The customer bid requests are sent through the Web services. The actual request is a message that is routed to a WebSphere message queue. Cottonwood Distributors has an application on that message queue that listens for activity so that the programs can retrieve database requests from the queue as the requests enter the queue. This listener application invokes a set of actions to process the requests. The listener balances the load that is introduced by the number of bid requests that enter the queue and processes the customer bid requests with the following steps:

- The customer bid Web services request writes a message to the queue.
- The listener application invokes a DB2 function to extract the order information from the request.
- The application runs a read-only query to obtain a quote for the requested part.
- The application inserts a record into the local table to record the order.

The following graphic displays the flow for the customer orders:



```
Web Services (NewDadx.dadx)
values db2mq.mqsend
('CDI', 'CDI', cast
(:msg as varchar(4000), 'CDI_IN_MSG')
```

```
Select
VARCHAR(DB2MQGETCOL('T.MSG,',',1),1),
INT(DB2MQ.GETCOL(T.MSG,',',2)),
INT(DB2MQ.GETCOL(T.MSG,',',3)),
INT(DB2MQ.GETCOL(T.MSG,',',4)),
DOUBLE(DB2MQ.GETCOL(T.MSG,',',5)),
FROM TABLE
DB2MQ.MQRECEIVALL('CDI','CDI',CDI_IN_MSG',1))AS T
```

```
Select c_name
from db2_customer
where c_custkey=:mqMsgKey
```

```
INSERT into request_bid values (:mqMsgKey, : mqMsgPart,
(SELECT MIN(ps_supplycost)* 1.45 FROM partsupp_fed
WHERE ps_parkey = :mqMsgPart
```

```
1,1301,100,1000,0
1,540,935,487,0
```

Figure 57. Cottonwood customer order requests

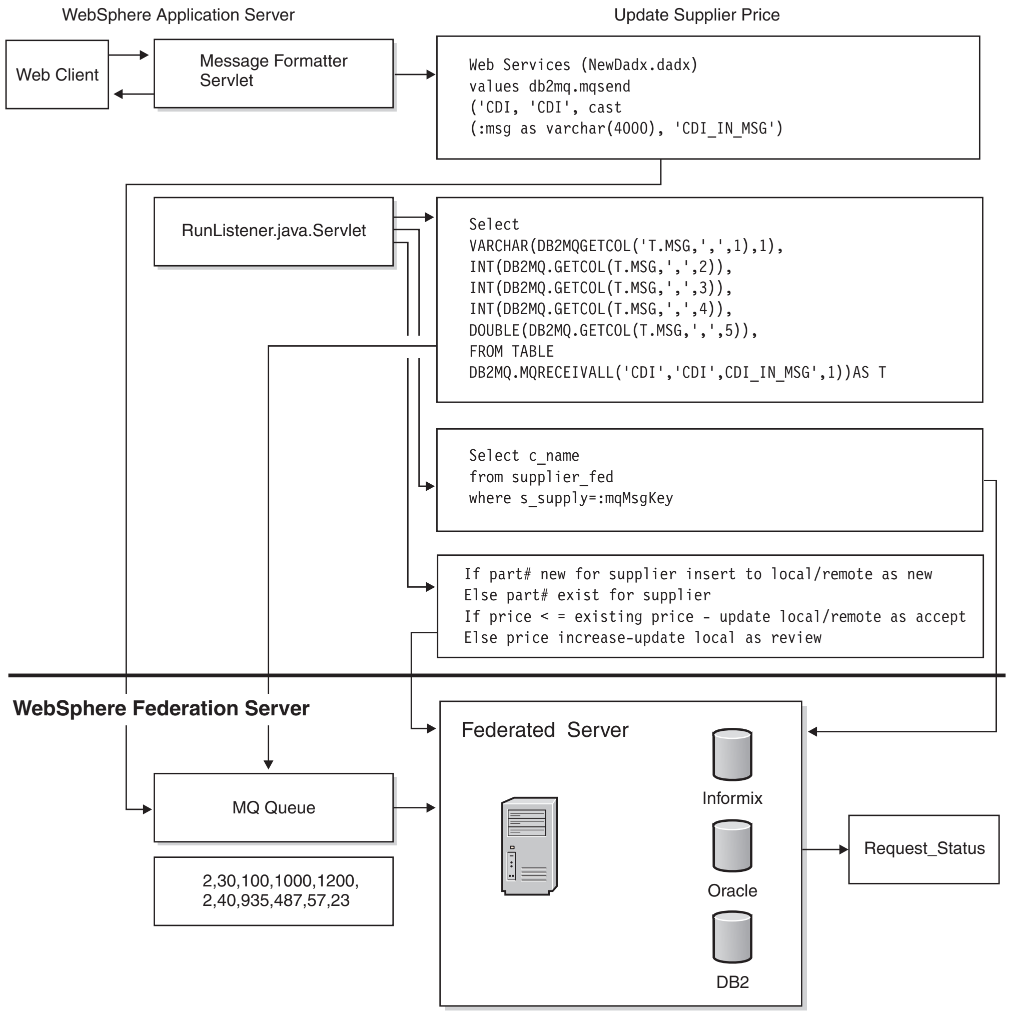
Supplier quote requests

Cottonwood Distributors processes the supplier quote updates in a series of steps that are similar to the customer bid requests. However, the supplier quotes query is not read-only because the program needs to update the quotes. The listener application allows Cottonwood Distributors to inspect the request before allowing the update to commit.

Here are the specific steps for the supplier quote requests:

- The supplier quote request writes a message to the queue.
- Based on the format of the message, the listener application invokes a DB2 function to extract the request to update a quote for a part.
- The applicaton reviews the previously lowest quote from that supplier for that part. If the new quote is lower or if the quote is for a part not previously supplied by that supplier, the application updates the database. If the new quote is higher, Cottonwood Distributors might not make the update. Instead, the Cottonwood application marks the quotes as something that the Cottonwood marketing team should review later or to negotiate with the supplier.

The following graphic displays the complete flow for the supplier price updates:



```
Web Services (NewDadx.dadx)
values db2mq.mqsend
('CDI, 'CDI', cast
(:msg as varchar(4000), 'CDI_IN_MSG')
```

```
Select
VARCHAR(DB2MQGETCOL('T.MSG,',',1),1),
INT(DB2MQ.GETCOL(T.MSG,',',2)),
INT(DB2MQ.GETCOL(T.MSG,',',3)),
INT(DB2MQ.GETCOL(T.MSG,',',4)),
DOUBLE(DB2MQ.GETCOL(T.MSG,',',5)),
FROM TABLE
DB2MQ.MQRECEIVALL('CDI','CDI',CDI_IN_MSG',1))AS T
```

```
Select c_name
from supplier_fed
where s_supply=:mqMsgKey
```

```
If part# new for supplier insert to local/remote as new
Else part# exist for supplier
If price < = existing price - update local/remote as accept
Else price increase-update local as review
```

```
2,30,100,1000,1200,
2,40,935,487,57,23
```

Figure 58. Supplier prices can be updated

Accessing information about the product

IBM has several methods for you to learn about products and services.

You can find the latest information on the Web: http://www.ibm.com/software/data/sw-bycategory/subcategory/SWB50.html

To access product documentation, go to publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order.
- To order publications by telephone in the United States, call 1-800-879-2755.

To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide.

Providing comments on the documentation

Please send any comments that you have about this information or other documentation.

Your feedback helps IBM to provide quality information. You can use any of the following methods to provide comments:

- Send your comments using the online readers' comment form at www.ibm.com/software/awdtools/rcf/.
- Send your comments by e-mail to comments@us.ibm.com. Include the name of the product, the version number of the product, and the name and part number of the information (if applicable). If you are commenting on specific text, please include the location of the text (for example, a title, a table number, or a page number).

Accessible documentation

Documentation is provided in XHTML format, which is viewable in most Web browsers.

XHTML allows you to view documentation according to the display preferences that you set in your browser. It also allows you to use screen readers and other assistive technologies.

Syntax diagrams are provided in dotted decimal format. This format is available only if you are accessing the online documentation using a screen reader.

Notices

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 document 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 document. The furnishing of this document does not grant 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 106-0032, 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 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.

Trademarks

Index

Printed in USA

SC19-1021-01

Spine information:

IBM Information Integration   Version 9.5

Application Development Guide for Federated Systems