

Bare JAX-WS

Paul Glezen, IBM

Abstract

This document is a member of the Bare Series of WAS topics distributed in both stand-alone and in collection form. The latest renderings and source are available on GitHub at <http://pglezen.github.io/was-config>.

Table of Contents

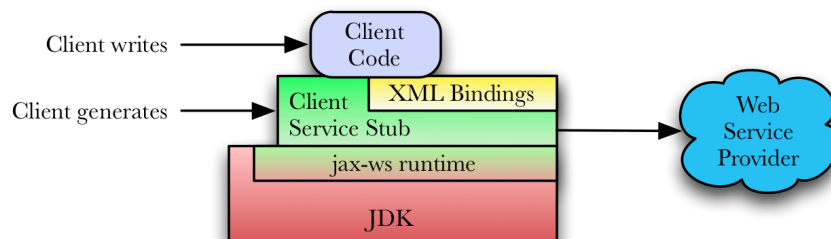
1. Bare JAX-WS Web Services	1
1.1. JAX-WS Clients	1
1.1.1. Thin Clients	2
1.1.2. Managed Clients	5
1.2. JAX-WS Servers	8
1.2.1. Using the Java Bindings Wizard	8
2. Source Listings	12
2.1. ccQuery.wsdl	12
2.2. AccountQueryEJB.java	14
3. References	15

1. Bare JAX-WS Web Services

1.1. JAX-WS Clients

Java API for XML - Web Services (JAX-WS) is a specification [3] that addresses Java web services development. For web service clients, this amounts to generating Java client service stubs and XML bindings for use in invoking the web service. Figure 1 illustrates how these components fit together. The green and yellow components are generated from the WSDL. The green is the service stub. Its objects make the remote call to the service provider. The yellow represents the XML-type-to-Java-type bindings (or simply XML bindings). The XML bindings are actually addressed by a separate specification called Java API for XML Bindings (JAXB) that is referenced by JAX-WS. This is part of what makes JAX-WS so much more powerful than its predecessor, Java API for XML - Remote Procedure Call (JAX-RPC).

Figure 1. JAX-WS Overview



Another recent improvement is the inclusion of the JAX-WS runtime in the Java 6 SE (standard edition). One no longer needs to reference special "thin client" libraries to make web service clients run. The generated bindings run directly against a Java 6 or later runtime. And not only are the runtime classes available in the JRE, the *wsimport*

utility, responsible for generating the bindings from the WSDL, is part of the JDK on any platform. No special IDEs or tools are needed.

The XML bindings are Java classes that map to the XML schema types defined in the WSDL. (One says that a Java type is bound to the XML schema type.) These types play the role of parameters for the service invocation. The invocation functions themselves are methods on the service stub objects. The bindings objects are passed as parameters to the service objects.

The generated service and binding objects tie into the JAX-WS runtime. This may be part of the JDK as in the diagram above. Or it may be implemented by a vendor such as Apache CXF or IBM WebSphere Application Server. In any case, it is responsible for

- marshaling the data structures into a serialized XML stream, and
- implementing the network protocol to transport the XML stream to the server.

Finally, the client code is the consumer of the service. It issues the request to the service stub and does whatever it requires with the result.

Web service clients may be *managed* or *unmanaged*. Managed clients are typically associated with an application server. The client is managed in the sense that aspects of its configuration are controllable through the administrative capabilities of the application server. References to the service stub objects are usually retrieved from JNDI. Unmanaged clients, also known as *thin clients*, do not rely on any underlying application server structure for configuration. Their service client proxy objects are directly instantiated. Their configuration is usually done by setting properties on the service stub instances. There is nothing wrong with running a thin client inside an application server. It simply won't benefit from enterprise manageability features.

1.1.1. Thin Clients

A thin client is one that does not expect the presence of any application server infrastructure. That's not to say a thin client can't run within an application server container. It simply doesn't depend on the container for resources or initialization.

Generating a thin client is easy and requires nothing more than a valid WSDL and JDK 6. The command for generating the JAX-WS bindings is **wsimport**. It should be in your command line path so long as your JDK is. To verify its version and presence in your path, query its version.

```
$ wsimport -version
JAX-WS RI 2.1.6 in JDK 6
$
```

We'll use the WSDL listed in [Section 2.1](#). It's a standalone WSDL file with a single operation that queries information about a credit card account. Let's start with the following invocation of `wsimport`.

```
thinclient$ wsimport -d bin -s src -p org.acme.cc.jaxws ccQuery.wsdl
parsing WSDL...
generating code...
compiling code...
thinclient$
```

The options have the following meanings.

- `-d` directory into which the compiled class files are placed
- `-s` directory into which the source code is generated
- `-p` package into which the source code is generated

If you run this command without first creating the `bin` and `src` directories, the command will give an error. Otherwise you get the following generated bindings classes.

```
thinclient/src/org/acme/cc/jaxws$ ls -l
total 64
-rw-r--r-- 1 pglezen  staff  1073 Jun 16 13:42 CCPortType.java
-rw-r--r-- 1 pglezen  staff  2341 Jun 16 13:42 CCSERVICE.java
-rw-r--r-- 1 pglezen  staff  1363 Jun 16 13:42 ObjectFactory.java
-rw-r--r-- 1 pglezen  staff  1813 Jun 16 13:42 QueryFault.java
-rw-r--r-- 1 pglezen  staff  1053 Jun 16 13:42 QueryFaultMsg.java
-rw-r--r-- 1 pglezen  staff  2061 Jun 16 13:42 QueryRequest.java
-rw-r--r-- 1 pglezen  staff  3727 Jun 16 13:42 QueryResponse.java
-rw-r--r-- 1 pglezen  staff   108 Jun 16 13:42 package-info.java
thinclient/src/org/acme/cc/jaxws$
```

The problem with this generation of bindings concerns the `CCService` class. It needs to find a copy of the WSDL and without any additional arguments to `wsimport`, it uses the fully-qualified path name to the WSDL file from which the bindings were generated.

```
URL baseUrl;
baseUrl = org.acme.cc.jaxws.CCService.class.getResource(".");
url = new URL(baseUrl, "file:/Users/pglezen/thinclient/ccQuery.wsdl");
```

Clearly we don't want code referencing an absolute path on a developer's workstation. We provide information to `wsimport` via the `-wsdllocation`. From the code snippet above, one can see that the base of the URL begins with the package directory of the class itself. The WSDL will be found if we add it to the directory holding `CCService.java`.

```
thinclient$ wsimport -d bin -s src -p org.acme.cc.jaxws -wsdllocation ccQuery.wsdl
ccQuery.wsdl
```

This results in the following snippet in `CCService.java`.

```
URL baseUrl;
baseUrl = org.acme.cc.jaxws.CCService.class.getResource(".");
url = new URL(baseUrl, "ccQuery.wsdl");
```

But then we have to make sure to copy the WSDL file to the source directory where `CCService.java` resides. An alternative is to count the directory levels between `CCService.java` (four in this case) and specify this to the `wsimport`. Then we can simply copy the WSDL to the `bin` directory.

```
thinclient$ wsimport -d bin -s src -p org.acme.cc.jaxws
-wsdllocation ../../../../ccQuery.wsdl ccQuery.wsdl
```

This result in the following snippet in `CCService.java`.

```
try {
    URL baseUrl;
    baseUrl = org.acme.cc.jaxws.CCService.class.getResource(".");
    url = new URL(baseUrl, "../../../../ccQuery.wsdl");
} catch (MalformedURLException e) {
    logger.warning("Failed to create URL for the wsdl
        Location: '../../../../ccQuery.wsdl', retrying as a local file");
    logger.warning(e.getMessage());
}
```

It makes for a funny-looking warning message if the WSDL is not found. The lesser evil is probably a matter of choice.

The final step is a main method to drive everything. An example is shown in [Example 1](#). If `Main.java` is in the current directory, it may be compiled as shown below.

```
pglezen:~/thinclient$ ls
Main.java bin/      ccQuery.wsdl  src/
```

```
pglezen:~/thinclient$ javac -d bin -classpath bin Main.java
pglezen:~/thinclient$
```

The `-d` option tells **javac** the root directory in which to place the class files. By putting it relative to `bin` directory, it will be placed with the bindings. Since the `Main` class references the bindings, and the bindings have already been compiled into the `bin` directory, it is all that's needed for the `-classpath` option.

Example 1. Main.java

```
package org.acme.cc.client;

import java.util.Map;

import javax.xml.ws.BindingProvider;

import org.acme.cc.jaxws.CCService;
import org.acme.cc.jaxws.CCPortType;
import org.acme.cc.jaxws.QueryRequest;
import org.acme.cc.jaxws.QueryResponse;
import org.acme.cc.jaxws.QueryFaultMsg;

public class Main {

    public static void main(String[] args) {
        String endpointUrl = "http://localhost:9080/cc/CCService";
        if (args.length == 1) {
            endpointUrl = args[0];
        }
        System.out.println("Using endpoint URL " + endpointUrl);
        CCService service = new CCService(); ❶
        CCPortType port = service.getCCPort(); ❷
        BindingProvider bp = (BindingProvider)port; ❸
        Map<String, Object> reqCtx = bp.getRequestContext();
        reqCtx.put("javax.xml.ws.service.endpoint.address", endpointUrl);

        QueryRequest request = new QueryRequest();
        request.setCcNo("2982-3929-5122-4829");
        request.setLastName("Brown");

        QueryResponse response = null;

        try {
            response = port.query(request); ❹
            System.out.println("Remote method returned.");
        } catch (QueryFaultMsg fault) {
            System.out.println("Caught service exception.");
            System.out.println("\tmsg = " + fault);
        }
        if (response != null) {
            System.out.println("Got response.");
            System.out.println("Account Num = " + response.getAcctNo());
            System.out.println("First name = " + response.getFirstName());
            System.out.println("Balance = " + response.getBalance());
        }
    }
}
```

- ❶ The `CCService` class corresponds to the `<wsdl:service>` definition that starts on line 79 of [Section 2.1](#). This class extends `javax.xml.ws.Service` as required by the JAX-WS specification.

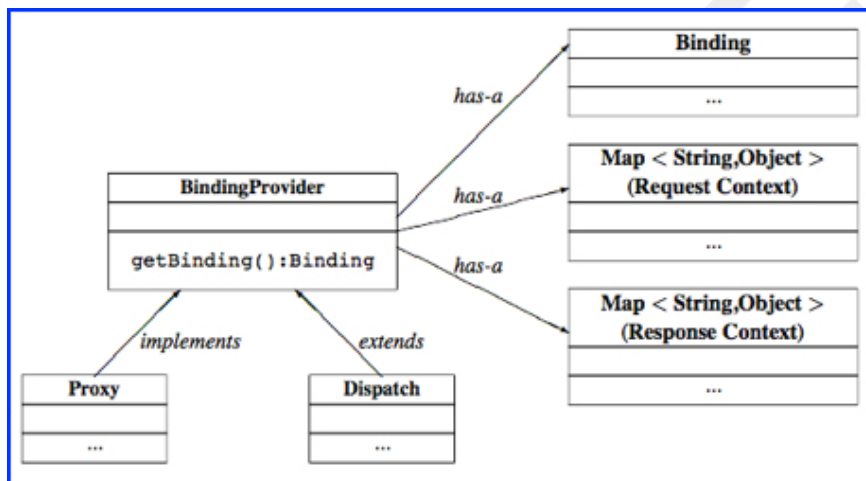
- ② The `CCPortType` interface corresponds to the `<wsdl:portType>` definition that starts on line 52 of [Section 2.1](#). The implementation is retrieved using the `getCCPort()` method on the service class. Such a method exists on the service class for each `<wsdl:port>` defined as in line 80 of [Section 2.1](#). Often there will be only one such definition. Examples of when there might be more are when there are multiple port-types or multiple SOAP bindings (1.1 and 1.2) for a single port-type.

Section 4.2.3 of the JAX-WS specification [3] warns us that these proxy types are not guaranteed to be thread safe. Some vendor implementations opt to provide a thread-safe implementation. But the specification does not require it; so appropriate care must be taken when executing in multithreaded environments.

- ③ The cast from a `CCPortType` to a `BindingProvider` may seem dangerous since `CCPortType` does not extend `BindingProvider`. But the JAX-WS specification requires that the implementation of `CCPortType` returned by the `getCCPort()` method also implement the `BindingProvider` interface as shown in [Figure 2](#).
- ④ This line is the actual remote invocation.

The `javax.xml.ws.BindingProvider` interface is key to the ability to dynamically set the remote endpoint. This and other capabilities are described in Section 4.2.1 on the JAX-WS 2.1 specification [3]. [Figure 2](#) comes from the JAX-WS 2.1 specification. It illustrates the relationship between the `BindingProvider` and the request context.

Figure 2. JAX-WS Binding Provider



1.1.2. Managed Clients

Managed clients are typically invoked by JEE components that rely on a server administrator for the configuration of the web service client. A JEE infrastructure administrator determines properties such as target URLs and timeouts. The instantiation of the client service object is typically injected into a field annotated with `@WebServiceRef`.

Example 2. JAX-WS Client Annotations

```

@Resource(name="MockCC", description="Stub the remote CC call")
Boolean mockCC = false;

@WebServiceRef(name="jaxwsCC") CCService ccService;
CCPortType account = null;
  
```

In the code snippet above, two variables are declared with the help of annotations. The first one is a `@Resource` annotation used to inject a `Boolean` value into `mockCC`. This determines whether our application returns a mock object implementation or actually attempts a remote invocation. This technique is very useful in web service client implementations since one cannot always rely on the service provider being available.

A CCService instance is injected at initialization time. The CCPortType instance is populated in initialization code. For an EJB, this would likely be a method annotated with `@PostConstruct`.

Example 3. EJB PostConstruct Method

```
@PostConstruct
void initDelegate() {
    log.entering(CLASSNAME, "initDelegate");
    if (mockCC) {
        log.warning("Mock is set to true. Using mock implementation of delegate.");
        account = new AccountQueryMock();
        log.fine("AccountQueryMock instance created.");
    } else {
        account = ccService.getCCService();
        log.fine("AccountQueryDelegate instance created.");
    }
    log.exiting(CLASSNAME, "initDelegate");
}
```

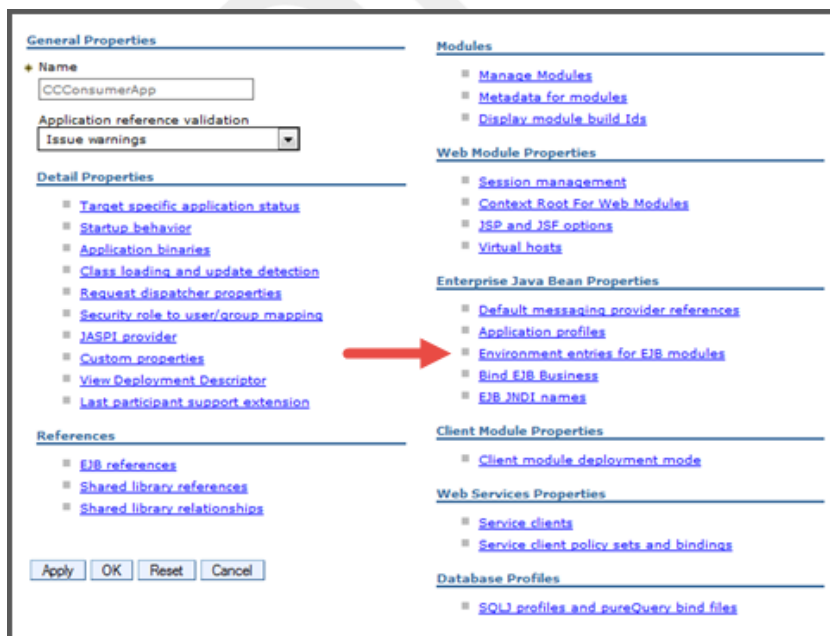
The `mockCC` value is used to determine whether to create a mock implementation of the delegate or a remote proxy implementation. (The full listing for this managed sample is in [Section 2.2.](#)) Make sure to log the fact that a mock object is being used instead of a true remote object. Moreover, the mock implementation should also log the fact it runs for every method request. It should be logged at the info level so that its involvement is clear for every request (in `SystemOut.log`). Otherwise one could easily forget that mocking is enabled and precipitate all kinds of backend fire drills.

The mock setting can be adjusted in the WAS admin console in the following way.

Procedure 1. Modify Mock Behavior

1. Login to the WAS admin console and navigate to the application panel. A sample application panel is shown in [Figure 3.](#)

Figure 3. CC Consumer Application Panel



2. Click the link labeled Environment entries for EJB modules as indicated in [Figure 3](#)
3. You can provide the MockCC value in the table as shown in [Figure 4](#). Notice that the values of the Name and Description columns correspond to the name and description attributes of the `@Resource` annotation for the `mockCC` field in [???](#).

Figure 4. CC Consumer EJB Mock Environment Entry

Module	URI	Bean	Name	Type	Description	Value
CCConsumerEJB.jar	CCConsumerEJB.jar,META-INF/ejb-jar.xml	AccountQueryEJB	MockCC	Boolean	Stub the remote CC call	<input type="text"/>

4. Click OK and save the changes.

The save operation will cause the application to restart so that the new value is immediately effective. This does *not* cause the JVM to restart. The restart should only take a few seconds for a properly written application.

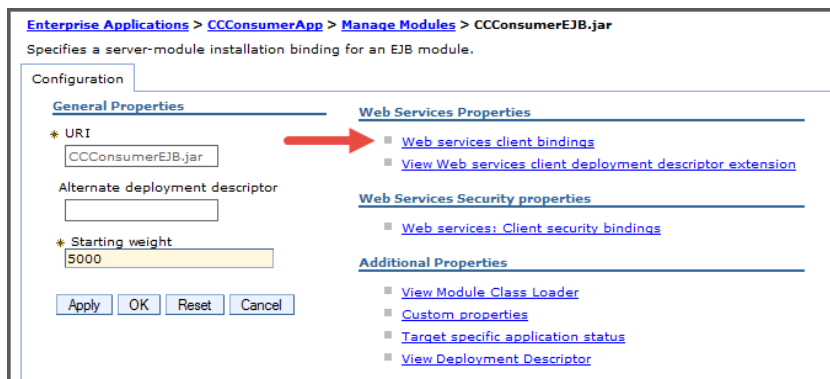
Another common JAX-WS administrative task is to *change the target endpoint*. The default endpoint is specified in the WSDL from which the bindings were generated. Since the endpoint is rarely known and WSDL design time and can change across environments, it is often simply set to `localhost`. Developers will often override this using their own proprietary schema via a property file somewhere on the file system. Once the endpoint host name is known, it can be set on the web service binding provider just as in the unmanaged case in [???](#). A drawback to this technique is that the property file must be consistent across each machine in the cluster. Moreover, a WAS admin must understand each application-specific JAX-WS customization technique.

The target endpoint can be managed through the WAS admin console (this is a *managed* client after all!). If the application doesn't override the endpoint host and port, the endpoint can be set following these steps.

Procedure 2. Modify JAX-WS Endpoint Destination

1. In the WAS admin console, navigate to the application panel.
2. Select the Manage Modules link.
3. Select the module containing the JAX-WS client.
4. Click the Web service client bindings link shown in [Figure 5](#).

Figure 5. Module with Client Bindings



- From the list of bindings, choose the one corresponding to the `@WebServiceRef` annotation in [Example 2](#). The name attribute of the annotation should match the Web Service column in the table of [Figure 6](#). From this row, click the Edit link in the Port Information column.

Figure 6. Client Binding List

Web Service	EJB	WSDL Filename	Preferred Port Mappings	Port Information
CCService	AccountQueryEJB	Use default (null) ▼	Edit...	Edit...
jaxwsCC	AccountQueryEJB	Use default (null) ▼	Edit...	Edit...

- Enter the full endpoint address in the column labeled Overridden Endpoint URL as shown in [Figure 7](#).

Figure 7. JAX-WS Endpoint Override

Port Information for Web service jaxwsCC, EJB AccountQueryEJB			
Port	Request Timeout (seconds)	Overridden Endpoint URL	Overridden Binding Namespace
{urn:issw:bare:wssec:cc:query}CCService		http://localhost:9086/provider/CCService	

- Click the OK and save.

The Save action will restart the application. The new new destination will become immediately effective. Remember this only works if the client code does not override the `BindingProvider` settings as done in [Example 1](#).

1.2. JAX-WS Servers

JAX-WS XML bindings for the server are no different than they are for the client. When using `wsimport` from the command line, there is no difference in the appearance of the output. The difference is *what you do with the output*. You can start by deleting the port type and service classes (the classes by annotations ❶ and ❷ of [Example 1](#)). These would only be used by a client and we're implementing the provider. But this begs the question: what do we use for the implementation class?

The answer, when using `wsimport` from the command line, is that there is nothing to help you with the implementation class. You just create one from scratch that matches the operations in the WSDL and use Java annotations to piece everything together. The implementation class doesn't even have to implement the service interface! This may seem a bit scary at first. But it's not hard once you know which annotations to use; and we'll get to that in a bit.

The good news for RAD (Rational Application Developer) users is that the Generate Java Bean Skeleton wizard actually does generate the skeleton with the annotations. It is this wizard that we'll discuss in detail for the provider implementation.

1.2.1. Using the Java Bindings Wizard

The Generate Java Bean Skeleton wizard is how we create and update the Java bindings and service implementation skeleton. Many of these instructions apply just as well to Eclipse JEE edition as they do to RAD.

1.2.1.1. JAX-WS Runtime

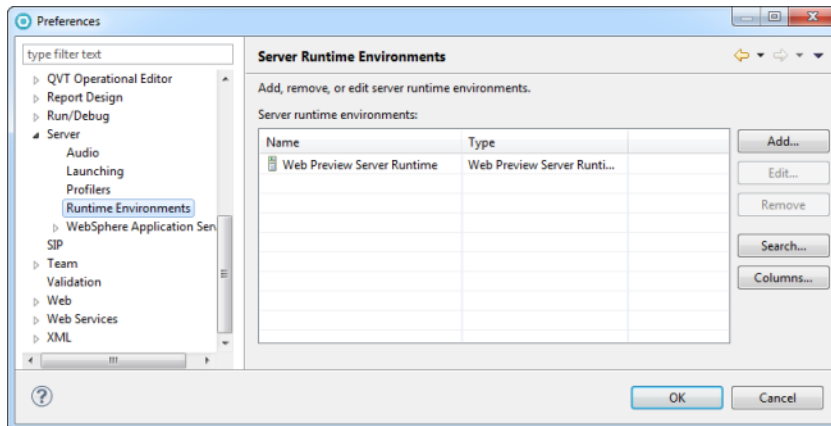
The JAX-WS bindings wizard requires that you specify which JAX-WS runtime library implementation to target. In our case, we'll target the WAS version of this runtime. But for the WAS runtime to be an available option, we need to register the location of the WAS runtime libraries on our developer workstation. The WAS runtime libraries are available from a local WAS installation on which you intend to run your local tests.

Follow these steps to register your WAS installation with RAD installation.

Procedure 3. Register WAS runtime with RAD

1. Open the preferences window to Window → Preferences and navigate to Server → Runtime Environment. This should reveal the currently available server runtime environment as shown in [Figure 8](#).

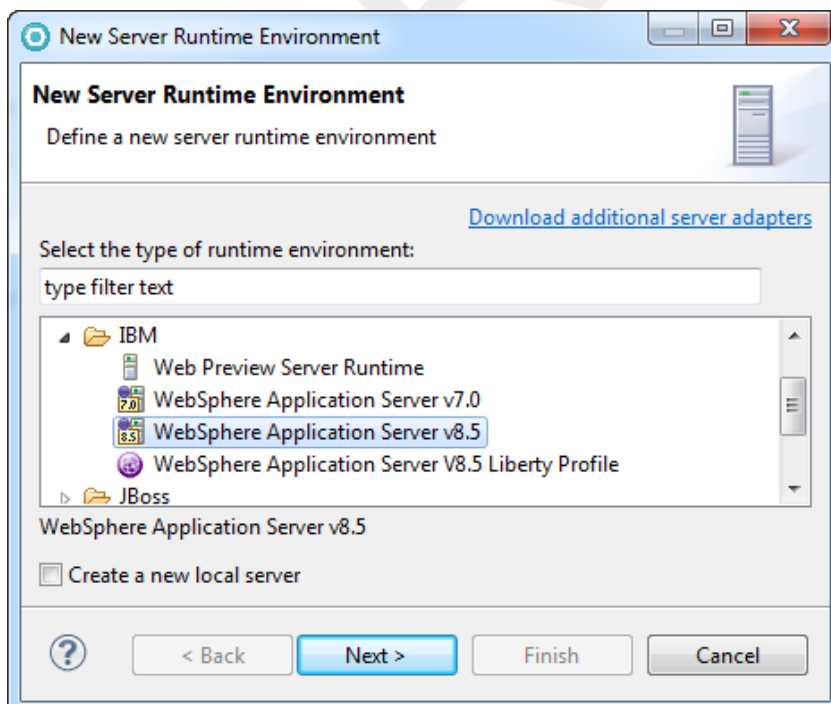
Figure 8. Initial Server Runtimes



If this is the first runtime you've cataloged, you will only see the Web Preview Server Runtime.

2. Click the Add in [Figure 8](#).
3. Select WebSphere Application Server v8.5 from the list of available server adapters as shown in [Figure 9](#).

Figure 9. Choose the WAS 8.5 Adapter



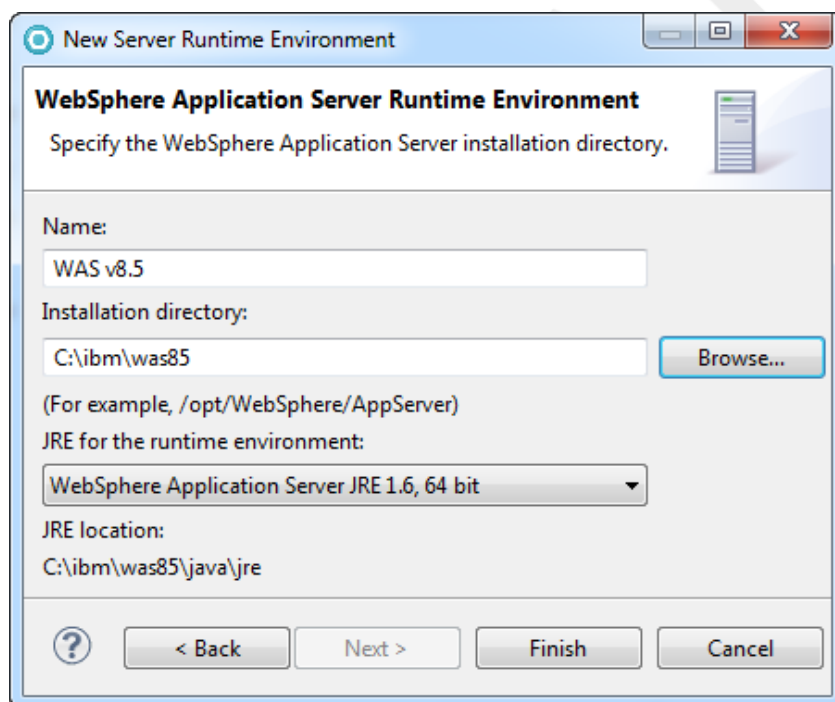
If this selection is not available in [Figure 9](#), it is because you did not install the WAS 8.5 test server component during your RAD install. You must

- a. download the WAS 8.5 test environment repository
 - b. make it available to your IBM Installation Manager
 - c. run the Installation Manager Modify command
 - d. add the WAS 8.5 test environment in the wizard.
4. Provide a Name for the runtime environment. You can shorten the default "WebSphere Application Server v8.5" to just "WAS v8.5". But be sure to note [Impact of Runtime Label on Team Development](#).

Impact of Runtime Label on Team Development

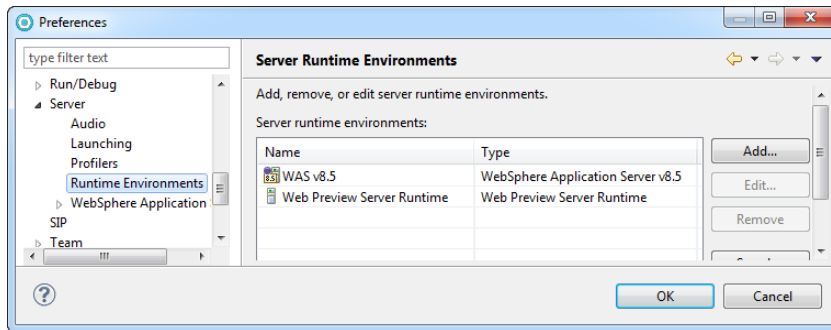
The value of the Name field in [Figure 10](#) will be embedded in your project metadata where ever you declare a dependency on it. Other developers with whom you share projects may declare this differently. When you share projects via source control, these dependencies won't resolve due to the name conflict. It's helpful to make sure everyone on the team agrees to the same environment name.

Figure 10. Provide WAS 8.5 location

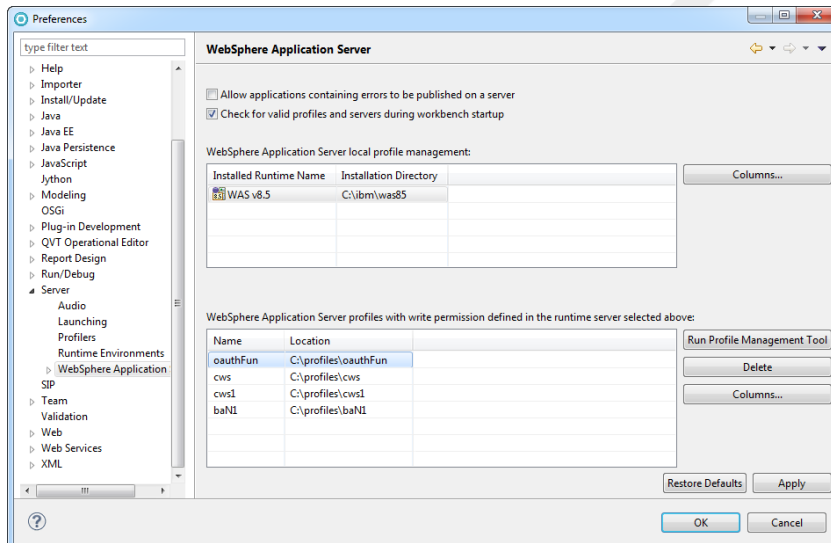


For the Installation directory field, enter the location of the WAS 8.5 installation on your workstation. The JRE information will be completed automatically when you enter a valid WAS 8.5 installation directory.

5. Click Finish. The result should be that the WAS 8.5 installation appears in the Server Runtime Environment panel as shown in [Figure 11](#).

Figure 11. Server runtimes with WAS 8.5

If you select the WebSphere Application Server entry in the preference navigation panel (just under Runtime Environments), you'll see the WAS 8.5 entry is present in the top list of [Figure 12](#). Select the WAS 8.5 entry and the bottom list will display a list of profiles.

Figure 12. WAS 8.5 Profiles

From this panel you may choose to create a new WAS 8.5 profile for your development activities.

Tip

It's good to use separate profiles for applications that will run in separate JVMs in order to model the isolation you expect in the production environment. On the other hand, applications which are expected to run together should share the same profile.

- Click OK to save your changes.

1.2.1.2. JAX-WS Emitter Preferences

The prospect of running the emitter wizard multiple times raises a concern regarding consistency. How do you remember to run the wizard with the same options every time? It would be nice if the emitter wizard allowed you to save your options in a file for future invocations (like when exporting a JAR archive in Eclipse). But there is no such

feature. The closest we can come is to configure defaults for the emitter wizard that are as close as possible to what you want so you can lessen the likelihood of making an error.

In the RAD preferences window there are several navigation choices under Web Services. Not all of them apply to JAX-WS generation. The Web Services → Resource Management. panel allows you to specify that

1.2.1.3. JAX-WS Emitter Wizard

To start the emitter wizard, right-click on the WSDL source file and select Web Services → Generate Java Bean Skeleton. The first panel is shown in [???](#). The WSDL file name should already be populated if you started the emitter wizard by right-clicking the WSDL file.

The skeleton generated by the RAD wizard is shown in the listing below.

```
package org.acme.cc.jaxws;

@javax.jws.WebService (endpointInterface="org.acme.cc.jaxws.CCPortType",
    targetNamespace="urn:issw:bare:wssec:cc:query",
    serviceName="CCService",
    portName="CCService",
    wsdlLocation="WEB-INF/wsdl/ccQuery.wsdl")
public class CCSoapBindingImpl{

    public QueryResponse query(QueryRequest parameters) throws QueryFaultMsg {
        // TODO Auto-generated method stub
        return null;
    }
}
```

2. Source Listings

2.1. ccQuery.wsdl

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <wsdl:definitions targetNamespace="urn:issw:bare:wssec:cc:query"
    xmlns:tns="urn:issw:bare:wssec:cc:query"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5     xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
      <schema targetNamespace="urn:issw:bare:wssec:cc:query"
        xmlns="http://www.w3.org/2001/XMLSchema">
10     <element name="QueryRequest">
      <complexType>
        <sequence>
          <element name="ccNo" type="xsd:string"/>
          <element name="lastName" type="xsd:string"/>
15     </sequence>
      </complexType>
    </element>
    <element name="QueryResponse">
      <complexType>
20     <sequence>
          <element name="ccNo" type="string"/>
          <element name="acctNo" type="string"/>
          <element name="lastName" type="string"/>
          <element name="firstName" type="string"/>

```

```
25         <element name="balance" type="int" />
          </sequence>
        </complexType>
      </element>
      <element name="QueryFault">
30        <complexType>
          <sequence>
            <element name="ccNo" type="string" />
            <element name="txnId" type="int" />
          </sequence>
35        </complexType>
      </element>
    </schema>
  </wsdl:types>

40 <wsdl:message name="QueryRequestMsg">
  <wsdl:part element="tns:QueryRequest" name="parameters" />
</wsdl:message>

<wsdl:message name="QueryResponseMsg">
45 <wsdl:part element="tns:QueryResponse" name="parameters" />
</wsdl:message>

<wsdl:message name="QueryFaultMsg">
  <wsdl:part element="tns:QueryFault" name="parameters" />
50 </wsdl:message>

<wsdl:portType name="CCPortType">
  <wsdl:operation name="query">
    <wsdl:input message="tns:QueryRequestMsg" name="queryRequest" />
55    <wsdl:output message="tns:QueryResponseMsg" name="queryResponse" />
    <wsdl:fault message="tns:QueryFaultMsg" name="queryFault" />
  </wsdl:operation>
</wsdl:portType>

60 <wsdl:binding name="CCSoapBinding" type="tns:CCPortType">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
http" />
  <wsdl:operation name="query">
    <wsdlsoap:operation soapAction="ccQuery" style="document" />

65    <wsdl:input name="queryRequest">
      <wsdlsoap:body use="literal" />
    </wsdl:input>

    <wsdl:output name="queryResponse">
70    <wsdlsoap:body use="literal" />
    </wsdl:output>

    <wsdl:fault name="queryFault">
      <wsdlsoap:fault name="queryFault" use="literal" />
75    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="CCService">
80 <wsdl:port binding="tns:CCSoapBinding" name="CCPort">
  <wsdlsoap:address location="http://localhost/services/statement" />
  </wsdl:port>
</wsdl:service>
```

```
</wsdl:definitions>
```

```
85
```

2.2. AccountQueryEJB.java

```
1 package org.acme.cc.ejb;

import java.util.logging.Logger;

5 import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.xml.ws.WebServiceRef;
10

import org.acme.cc.delegate.AccountQuery;
import org.acme.cc.delegate.AccountQueryDelegate;
import org.acme.cc.jaxws.CCPortType;
import org.acme.cc.jaxws.CCService;
15 import org.acme.cc.jaxws.QueryFaultMsg;
import org.acme.cc.jaxws.QueryRequest;
import org.acme.cc.jaxws.QueryResponse;

@Stateless
20 @LocalBean
public class AccountQueryEJB {

    public static final String CLASSNAME = AccountQueryEJB.class.getName();
    private static final Logger log = Logger.getLogger(CLASSNAME);
25

    @Resource(name="MockCC", description="Stub the remote CC call")
    Boolean mockCC = false;

    @WebServiceRef(name="jaxwsCC") CCService ccService;
30 CCPortType account = null;

    public AccountQueryEJB() {
        log.entering(CLASSNAME, "ctor");
        log.exiting(CLASSNAME, "ctor");
35    }

    @PostConstruct
    void initDelegate() {
        log.entering(CLASSNAME, "initDelegate");
40        if (mockCC) {
            log.warning("Mock is set to true. Using mock implementation of
delegate.");
            account = new AccountQueryMock();
            log.fine("AccountQueryMock instance created.");
        } else {
45            account = ccService.getCCService();
            log.fine("AccountQueryDelegate instance created.");
        }
        log.exiting(CLASSNAME, "initDelegate");
50    }

    public QueryResponse query(QueryRequest request) throws QueryFaultMsg {
        log.entering(CLASSNAME, "query");
        QueryResponse response = account.query(request);
```

```
    log.exiting(CLASSNAME, "query");  
55     return response;  
    }  
}
```

3. References

- [1] WAS 8.0 Info Center, IBM. Online: <http://pic.dhe.ibm.com/infocenter/wasinfo/v8r0/>
- [2] WS-SecurityPolicy 1.2 Specification, December, 2006. OASIS. Online: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512>
- [3] JAX-WS 1.2 Specification, May, 2007. Sun Microsystems, Inc. Online: <http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html>
- [4] Developing Web Service Applications, IBM. Red Paper Online: <http://www.redbooks.ibm.com/redpapers/pdfs/redp4884.pdf>

DRAFT