Martin Keen
Rafael Coutinho
Sylvi Lippmann
Salvatore Sollami
Sundaragopal Venkatraman
Steve Baber
Henry Cui
Craig Fleming

Developing Web Services Applications

This IBM® Redpaper™ publication introduces the concept of a service-oriented architecture (SOA). The intended audience is web developers interested in SOA. It explains how to realize this type of an architecture using the following Java Enterprise Edition (Java EE 6) web services specifications:

- *Java Specification Request (JSR) 224*: *Java API for XML-Based Web Services (JAX-WS) 2.2*
- *JSR 311: Java API for RESTful Web Services 1.1 (JAX-RS)*

It explores the features that are provided by IBM Rational Application Developer for web services development and security. It also demonstrates how Rational Application Developer can help with testing web services and developing web services client applications.

The paper is organized into the following sections:

- Introduction to web services
- New function in Java EE 6 for web services
- JAX-WS programming model
- Web services development approaches
- Web services tools in Rational Application Developer
- Preparing for the JAX-WS samples
- Creating bottom-up web services from a JavaBean
- Creating a synchronous web service JSP client
- Creating a web service JavaServer Faces client
- Creating a web service thin client
- Creating asynchronous web service clients
- Creating web services from an EJB
- Creating a top-down web service from a WSDL
- Creating web services with Ant tasks
- Sending binary data using MTOM
- JAX-RS programming model
- Web services security
- WS-Policy
- WS-MetadataExchange (WS-MEX)
- Security Assertion Markup Language (SAML) support
- More information

The sample code for this paper is in the `4884code\webservice` folder.

This paper was originally published as a chapter in the IBM Redbooks® publication, *Rational Application Developer for WebSphere Software V8 Programming Guide*, SG24-7835. The full publication includes working examples that show how to develop applications and achieve the benefits of visual and rapid application development. It is available at this website:

http://www.redbooks.ibm.com/abstracts/sg247835.html?Open

Introduction to web services

This section introduces architecture and concepts of the SOA and web services.

SOA

In an SOA, applications are made up of loosely coupled software services, which interact to provide all the functionality needed by the application. Each service is generally designed to be self-contained and stateless to simplify the communication that takes place between them.

There are three major roles involved in an SOA:

- Service provider
- Service broker
- Service requester

Figure 1 shows the interactions between these roles.

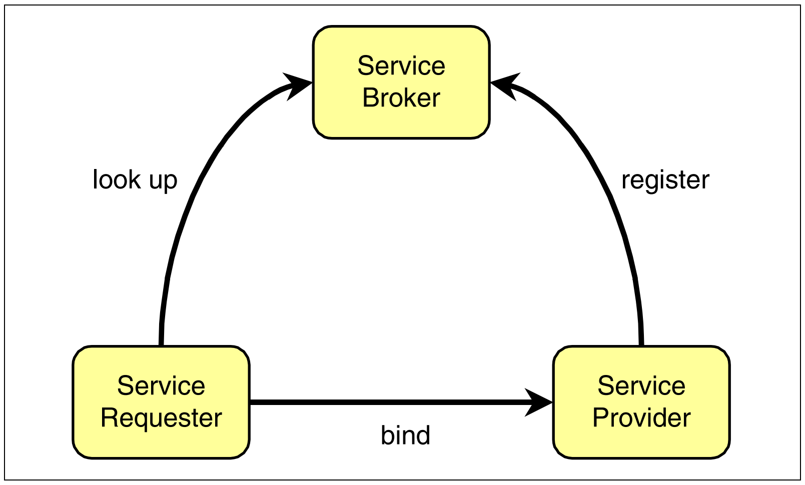


Figure 1 Service-oriented architecture

Service provider

The *service provider* creates a service and can publish its interface and access information to a *service broker*.

A service provider must decide which services to expose and how to expose them. Often, a trade-off exists between security and interoperability; the service provider must make technology decisions based on this trade-off. If the service provider uses a service broker, decisions must be made about how to categorize the service, and the service must be registered with the service broker using agreed-upon protocols.

Service broker

The *service broker,* also known as the *service registry,* is responsible for making the service interface and implementation access information that is available to any potential service requester.

The service broker provides mechanisms for registering and finding services. A particular broker might be public (for example, available on the Internet) or private, only available to a limited audience (for example, on an intranet). The type and format of the information stored by a broker and the access mechanisms used is implementation-dependent.

Service requester

The *service requester*, also known as a *service client*, discovers services and then uses them as part of its operation.

A service requester uses services provided by service providers. Using an agreed-upon protocol, the requester can find the required information about services using a broker (or this information can be obtained in another way). After the service requester has the necessary details of the service, it can bind or connect to the service and invoke operations on it. The binding is usually static, but the possibility of dynamically discovering the service details from a service broker and configuring the client accordingly makes dynamic binding possible.

Web services as an SOA implementation

Web services provides a technology foundation for implementing an SOA. A major focus of this technology is interoperability. The functional building blocks must be accessible over standard Internet protocols. Internet protocols are independent of platforms and programming languages, which ensures that high levels of interoperability are possible.

Web services are self-contained software services that can be accessed using simple protocols over a network. They can also be described using standard mechanisms, and these descriptions can be published and located using standard registries. Web services can perform a wide variety of tasks, ranging from simple request-reply tasks to full business process interactions.

By using tools, such as Rational Application Developer, existing resources can be exposed as web services easily.

The following core technologies are used for web services:

- Extensible Markup Language (XML)
- SOAP
- Web Services Description Language (WSDL)

Extensible Markup Language (XML)

XML is the markup language that underlies web services. XML is a generic language that can be used to describe any content in a structured way, separated from its presentation to a specific device. All elements of web services use XML extensively, including XML namespaces and XML schemas.

The specification for XML is available at the following address:

http://www.w3.org/XML/

SOAP

SOAP is a network, transport, and programming language-neutral protocol that allows a client to call a remote service. The message format is XML. SOAP is used for all communication between the service requester and the service provider. The format of the individual SOAP messages depends on the specific details of the service being used.

The specification for SOAP is available at the following address:

http://www.w3.org/TR/soap/

Web Services Description Language (WSDL)

WSDL is an XML-based interface and implementation description language. The service provider uses a WSDL document to specify the following items:

- The operations that a web service provides
- The parameters and data types of these operations
- The service access information

WSDL is one way to make service interface and implementation information available in a service registry. A server can use a WSDL document to deploy a web service. A service requester can use a WSDL document to work out how to access a web service (or a tool can be used for this purpose).

The specification for WSDL is available at the following address:

http://www.w3.org/TR/wsdl/

New function in Java EE 6 for web services

Java EE 6 includes several API specifications that provide web services support. Several of these specifications were already included in Java EE 5 and have been upgraded in Java EE 6. Several of these specifications are entirely new in Java EE 6. The most notable example is *JSR 311: JAX-RS: Java API for RESTful Web Services 1.1*.

The specifications for web services support in Java EE are available at the following web address:

http://www.oracle.com/technetwork/java/javaee/tech/webservices-139501.html

For information about standards related to web services supported by IBM Rational® Application Developer, see the following address:

http://publib.boulder.ibm.com/infocenter/radhelp/v8/index.jsp?topic=/com.ibm.webservice.doc/topics/core/cwsfpstandards.html

This information center describes which versions of the standards are supported by IBM WebSphere® Application Server V8.0, V7.0, and V6.1 with or without the Feature Pack for Web Services.

JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.2

The Java API for XML-Based Web Services (JAX-WS) is a programming model that simplifies application development through the support of a standard, annotation-based model to develop web services applications and clients.

The JAX-WS programming standard aligns itself with the document-centric messaging model and replaces the remote procedure call programming model defined by the Java API for XML-based RPC (JAX-RPC) specification. Although Rational Application Developer still supports the JAX-RPC programming model and applications, JAX-RPC has limitations and does not support many current document-centric services. JAX-RPC will not be described further in this paper.

Table 1 shows the WebSphere Application Server versions that support JAX-WS 2.0, 2.1, and 2.2.

Table 1 WebSphere Application Server support for JAX-WS versions

Java EE version	JAX-WS version	WebSphere Application Server version
Java EE 5	JAX-WS 2.0	6.1 with Feature Pack for Web Services 7.0 8.0
Java EE 5	JAX-WS 2.1	7.0 8.0
Java EE 6	JAX-WS 2.2	8.0

JAX-WS 2.1 introduces support for the WS-Addressing in a standardized API. Using this function, you can create, transmit, and use endpoint references to target a web service endpoint. You can use this API to specify the action uniform resource identifiers (URIs) that are associated with the WSDL operations of your Web service.

JAX-WS 2.1 introduces the concept of features as a way to programmatically control specific functions and behaviors. Three standard features are available: the AddressingFeature for WS-Addressing, the MTOMFeature when optimizing the transmission of binary attachments, and the RespectBindingFeature for wsdl:binding extensions. JAX-WS 2.1 requires Java Architecture for XML Binding (JAXB) Version 2.1 for data binding.

For more information about the features of JAX-WS 2.1, refer to the WebSphere Application Server 7.0 Information Center:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/cwbs_jaxws.html`

WebSphere Application Server Version 8.0 supports the *JSR 109: JAX-WS Version 2.2 and Web Services for Java EE Version 1.3 specifications*.

The JAX-WS 2.2 specification supersedes and includes functions within the JAX-WS 2.1 specification. JAX-WS 2.2 adds client-side support for using WebServiceFeature-related annotations, such as `@MTOM`, `@Addressing`, and the `@RespectBinding` annotations. JAX-WS 2.1 had previously added support for these annotations on the server.

For more information about the features of JAX-WS 2.2, refer to this website:

`http://publib.boulder.ibm.com/infocenter/radhelp/v8/index.jsp?topic=/com.ibm.webservice.doc/topics/core/cjaxws.html`

In Rational Application Developer, you can choose which version of JAX-WS code to produce when generating web services *top-down* (from an existing WSDL file) and when generating web services clients. You can find the corresponding options by selecting **Windows** → **Preferences** → **Web Services** → **WebSphere** → **JAX-WS Code Generation**:

- **Top Down** → **Version of JAX-WS code to be generated**
- **Client** → **Version of JAX-WS code to be generated**

These default options can be further overridden in the Web Services code generation wizard.

JSR 222: Java Architecture for XML Binding (JAXB) 2.2

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for the simplified development of web services. JAXB uses the flexibility of platform-neutral XML data in Java applications to bind XML schema to Java applications without requiring extensive knowledge of XML programming.

JAXB is the default data binding technology that the JAX-WS tooling uses and is the default implementation within this product. You can develop JAXB objects for use within JAX-WS applications.

JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents. JAXB data binding replaces the data binding described by the JAX-RPC specification.

WebSphere Application Server V7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding. JAXB 2.1 provides enhancements, such as improved compilation support and support for the `@XML` annotation, and full schema 1.0 support.

WebSphere Application Server V8.0 supports the JAXB 2.2 specification. JAXB 2.2 provides minor enhancements to its annotations for improved schema generation and better integration with JAX-WS. JAX-WS 2.2 requires Java Architecture for XML Binding (JAXB) Version 2.2 for data binding.

JSR 109: Implementing Enterprise Web Services

Implementing Enterprise Web Services: JSR 109 defines the programming model and runtime architecture to deploy and look up web services in the Java EE environment, more specifically, in the web, Enterprise JavaBeans (EJB), and client application containers. One of the major goals of JSR 109 is to ensure that vendors' implementations interoperate.

WebSphere Application Server V8 introduces support for *Web Services for Java EE (JSR 109) Version 1.3 specification*. The Web Services for Java EE 1.3 specification introduces support for WebServiceFeature-related annotations, as well as support for using deployment descriptor elements to configure these features on both the client and server.

Related web services standards

Next we describe the related web services specifications.

JSR 67: SOAP with Attachments API for Java (SAAJ)

The SOAP with Attachments API for Java (SAAJ) interface is used for SOAP messaging that provides a standard way to send XML documents over the Internet from a Java programming model. SAAJ is used to manipulate the SOAP message to the appropriate context as it traverses through the runtime environment.

JSR 173: Streaming API for XML (StAX)

Streaming API for XML (StAX) is a streaming Java-based, event-driven, pull-parsing API for reading and writing XML documents. With StAX, you can create bidirectional XML parsers that are fast, relatively easy to program, and have a light memory footprint.

JSR 181: Web Services Metadata for the Java Platform

Web Services Metadata for the Java Platform defines an annotated Java format that uses *JSR 175: Metadata Facility for the Java Programming Language* to enable the easy definition of Java web services in a Java EE container.

Web Services Interoperability Organization

In an effort to improve the interoperability of web services, the Web Services Interoperability Organization (known as WS-I) was formed. WS-I produces a specification known as the *WS-I Basic Profile*. This specification describes the technology choices that maximize interoperability between web services and clients running on separate platforms, using separate runtime systems, and written in multiple languages.

The WS-I Basic Profile is available at the following address:

`http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile`

Web Services Security

The WS-Security specification describes extensions to SOAP that allow for the quality of protection of SOAP messages, including message authentication, message integrity, and message confidentiality. The specified mechanisms can be used to accommodate a wide variety of security models and encryption technologies. It also provides a general-purpose mechanism for associating security tokens with message content. For additional information, see the following web address:

`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

JAX-WS programming model

JAX-WS is the strategic programming model for developing web services and is a required part of the Java EE 5 and Java EE 6 platforms. JAX-WS simplifies application development through the support of a standard, annotation-based model to develop web service applications and clients. The JAX-WS programming standard strategically aligns itself with the current industry trend toward a more document-centric messaging model.

Implementing the JAX-WS programming standard provides the enhancements described in the following sections for developing web services and clients.

Better platform independence for Java applications

Using JAX-WS APIs and developing web services and clients are simplified with better platform independence for Java applications. JAX-WS takes advantage of dynamic proxies whereas JAX-RPC uses generated stubs. The dynamic proxy client invokes a web service that is based on a service endpoint interface (SEI) that is generated or provided. The dynamic proxy client is similar to the stub client in the JAX-RPC programming model. Although the JAX-WS dynamic proxy client and the JAX-RPC stub client are both based on the SEI that is generated from a WSDL file, note the following major differences:

- The dynamic proxy client is dynamically generated at run time using the Java 5 dynamic proxy functionality. The JAX-RPC-based stub client is a non-portable Java file that is generated by tooling.
- Unlike the JAX-RPC stub clients, the dynamic proxy client does not require you to regenerate a stub prior to running the client on an application server for a separate vendor, because the generated interface does not require the specific vendor information.

Annotations

JAX-WS introduces support for annotating Java classes with metadata to indicate that the Java class is a web service. JAX-WS supports the use of annotations based on the *JSR 175: Metadata Facility for the Java Programming Language specification*, *the JSR 181: Web Services Metadata for the Java Platform specification,* and annotations that are defined by the *JAX-WS 2.0/2.1/2.2 specification*. Using annotations in the Java source and Java class simplifies the development of web services by defining part of the additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL files into the source artifacts.

For example, you can embed a simple `@WebService` annotation in the Java source to expose the bean as a web service (Example 1).

Example 1 JAX-WS annotation

```
@WebService
public class BankBean {
   public String getCustomerFullName(String ssn) { ... }
}
```

The `@WebService` annotation tells the server runtime environment to expose all public methods on that bean as a web service. Additional levels of granularity can be controlled by adding additional annotations on individual methods or parameters. The use of annotations makes it much easier to expose Java artifacts as web services. In addition, as artifacts are created from using part of the top-down mapping tools starting from a WSDL file, annotations are included within the source and Java classes as a way of capturing the metadata along with the source files.

Invoking web services asynchronously

With JAX-WS, web services can be called both synchronously and asynchronously. JAX-WS adds support for both a polling mechanism and callback mechanism when calling web services asynchronously. By using a polling model, a client can issue a request and get a response object back, which is polled to determine whether the server has responded. When the server responds, the actual response is retrieved. With the polling model, the client can continue to process other work without waiting for a response to return.

With the callback model, the client provides a callback handler to accept and process the inbound response object. Both the polling and callback models enable the client to focus on continuing to process work while providing for a more dynamic and efficient model to invoke web services.

For example, a web service interface has methods for both synchronous and asynchronous requests (Example 2). Asynchronous requests are identified in bold.

Example 2 Asynchronous methods in the web service interface

```
@WebService
public interface CreditRatingService {
   // sync operation
   Score getCreditScore(Customer customer);
   // async operation with polling
   Response<Score> getCreditScoreAsync(Customer customer);
   // async operation with callback
   Future<?> getCreditScoreAsync(Customer customer,
                                    AsyncHandler<Score> handler);
}
```

The asynchronous invocation that uses the callback mechanism requires an additional input by the client programmer. The *callback handler* is an object that contains the application code that is executed when an asynchronous response is received. Example 3 shows an asynchronous callback handler.

Example 3 Asynchronous callback handler

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerFred,
   new AsyncHandler<Score>() {
      public void handleResponse(Response<Score> response) {
         Score score = response.get();
         // do work here...
      }
   }
);
```

Example 4 shows an asynchronous polling client.

Example 4 Asynchronous polling

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerFred);

while (!response.isDone()) {
   // do something while we wait
}

// no cast needed, thanks to generics
Score score = response.get();
```

Dynamic and static clients

The dynamic client programming API for JAX-WS is called the *dispatch client* (`javax.xml.ws.Dispatch`). The dispatch client is an XML messaging-oriented client. The data is sent in either `PAYLOAD` or `MESSAGE` mode:

- PAYLOAD: When using the `PAYLOAD` mode, the dispatch client is only responsible for providing the contents of the `<soap:Body>` element and JAX-WS adds the `<soap:Envelope>` and `<soap:Header>` elements.
- MESSAGE: When using the `MESSAGE` mode, the dispatch client is responsible for providing the entire SOAP envelope including the `<soap:Envelope>`, `<soap:Header>`, and `<soap:Body>` elements and JAX-WS does not add anything additional to the message. The dispatch client supports asynchronous invocations using a callback or polling mechanism.

The static client programming model for JAX-WS is called the *proxy client*. The proxy client invokes a web service based on an SEI that is generated or provided.

Message Transmission Optimization Mechanism support

With JAX-WS, you can send binary attachments, such as images or files, along with web services requests. JAX-WS adds support for optimized transmission of binary data as specified by Message Transmission Optimization Mechanism (MTOM).

Multiple payload structures

JAX-WS exposes the XML Source, SAAJ 1.3, and JAXB 2.2 binding technologies to the user.

With XML Source, a user can pass a `javax.xml.transform.Source` to the run time, which represents the data in a source object to be passed to the run time. SAAJ 1.3 now has the ability to pass an entire SOAP document across the interface, rather than only the payload. This action is done by the client passing the SAAJ `SOAPMessage` object across the interface. JAX-WS uses the JAXB 2.2 support as the data binding technology of choice between Java and XML.

SOAP 1.2 support

Support for SOAP 1.2 was added to JAX-WS 2.0. JAX-WS supports both SOAP 1.1 and SOAP 1.2. SOAP 1.2 provides a more specific definition of the SOAP processing model, which removes many of the ambiguities that sometimes led to interoperability problems in the absence of the WS-I profiles. SOAP 1.2 reduces the chances of interoperability issues with SOAP 1.1 implementations between separate vendors. It is not interoperable with earlier versions.

Web services development approaches

You can follow two general approaches to web service development:

- In the *top-down approach*, a web service is based on the web service interface and XML types, defined in WSDL and XML Schema Definition (XSD) files. You first design the implementation of the web service by creating a WSDL file using the WSDL editor. You can then use the Web Service wizard to create the web service and skeleton Java classes to which you can add the required code. You then modify the skeleton implementation to interface with the business logic.

  The top-down approach provides more control over the web service interface and the XML types used. Use this approach for developing new web services.

- In the *bottom-up approach*, a web service is created based on the existing business logic in JavaBeans or EJB. A WSDL file is generated to describe the resulting web service interface.

  The bottom-up pattern is often used for exposing existing function as a web service. It might be faster, and no XSD or WSDL design skills are needed. However, if complex objects (for example, Java collection types) are used, the resulting WSDL might be difficult to understand and less interoperable.

Web services tools in Rational Application Developer

Rational Application Developer provides tools to create web services from existing Java and other resources or from WSDL files. Rational Application Developer also provides tools for web services client development and for testing web services.

Creating a web service from existing resources

Rational Application Developer provides wizards for exposing a variety of resources as web services. You can use the following resources to build a web service:

- JavaBean: The Web Service wizard assists you in creating a new web service from a simple Java class, configures it for deployment, and deploys the web service to a server. The server can be the WebSphere Application Server V6.1, V7.0, or V8.0 that is included with Rational Application Developer or another application server.
- EJB: The Web Service wizard assists you in creating a new web service from a stateless session EJB, configuring it for deployment, and deploying the web service to a server.

Creating a skeleton web service

Rational Application Developer provides the functionality to create web services from a description in a WSDL or Web Services Inspection Language (WSIL) file:

- JavaBean from WSDL: The web services tools assist you in creating a skeleton JavaBean from an existing WSDL document. The skeleton bean contains a set of methods that correspond to the operations described in the WSDL document. When the bean is created, each method has a trivial implementation that you replace by editing the bean.
- EJB from WSDL: The web services tools support the generation of a skeleton EJB from an existing WSDL file. Apart from the type of component produced, the process is similar to that for JavaBeans.

Client development

To assist in the development of web service clients, Rational Application Developer provides the following features:

- Java client proxy from WSDL: The Web Service client wizard assists you in generating a proxy JavaBean. This proxy can be used within a client application to greatly simplify the client programming required to access a web service.
- Sample web application from WSDL: Rational Application Developer can generate a sample web application, which includes the proxy classes described before, and sample JavaServer Pages (JSP) that use the proxy classes.
- Web Service Discovery Dialog: On this window, you can discover a web service that exists online or in your workspace, create a proxy for the web service, and then place the methods of the proxy into a Faces JSP file.

Testing tools for web services

To allow developers to test web services, Rational Application Developer provides a range of features:

- WebSphere Application Server V8.0, V7.0, and V6.1 test environment: These servers are included with Rational Application Developer as test servers and can be used to host web services. This feature provides a range of web services run times, including an implementation of the J2EE specification standards.

- Generic service client: The generic service client can invoke calls to any service that uses an HTTP, a Java Message Service (JMS), or WebSphere MQ transport and can view the message returned by the service.
- Sample JSP application: The web application mentioned before can be used to test web services and the generated proxy it uses.
- Web Services Explorer: This simple test environment can be used to test any web service, based only on the WSDL file for the service. The service can be running on a local test server or anywhere else in the network. The Web Services Explorer is a JSP web application that is hosted on the Apache Tomcat servlet engine in Eclipse. The Web Services Explorer uses the WSDL to render a SOAP request. It does not involve data marshalling and unmarshalling. The return parameter is stripped out, and the values are displayed in a predefined format.
- Universal Test Client: The Universal Test Client (UTC) is a powerful and flexible test application that is normally used for testing EJB. Its flexibility makes it possible to test ordinary Java classes, so it can be used to test the generated proxy classes created to simplify client development.
- TCP/IP Monitor: The TCP/IP Monitor works similarly to a proxy server, passing TCP/IP requests to another server and directing the returned responses back to the originating client. The TCP/IP messages that are exchanged are displayed in a special view within Rational Application Developer.

Preparing for the JAX-WS samples

To prepare for this sample, we import sample code, which is a simple web application that includes Java classes and an EJB.

Importing the sample

In this section, prepare the environment for the JAX-WS web services application samples:

1. In the Java EE perspective, select **File** → **Import**.
2. Select **General** → **Existing Projects into Workspace**.
3. In the Import Projects window, select **Select archive file**.
4. Click **Browse**. Navigate to the `4884code\webservices` folder and select the **RAD8WebServiceStart.zip** file. Click **Open**.
5. Click **Select All** and click **Finish**.

After the build, no warning or error messages are displayed in the workspace.

Sample projects

The sample application that we use for creating the web service consists of the following projects:

- `RAD8WebServiceUtility` project: This project is a simple banking model with `BankMemory`, `Customer`, and `Account` beans.
- `RAD8WebServiceWeb` project: This project contains the `SimpleBankBean`, a JavaBean with a few methods that retrieve data from the `MemoryBank`, a search HTML page, and a resulting JSP. We use annotations to generate web services for this project.
- `RAD8WebServiceWeb2` project: This project contains the same code as the `RAD75WebServiceWeb` project. We use the Web Service wizard to generate web services for this project.
- `RAD8WebServiceEJB` project: This project contains the `SimpleBankFacade` session EJB with a few methods that retrieve data from the `MemoryBank`.
- `RAD8WebServiceEAR` project: This project is the enterprise application that contains the other four projects.

Testing the application

To start and test the application, follow these steps:

1. In the Servers view, start WebSphere Application Server V8.0.
2. Right-click the server and select **Add and remove projects**.
3. In the Add and Remove Projects window, select **RAD8WebServiceEAR**, click **Add**, and then click **Finish**.
4. Expand **RAD8WebServiceWeb** → **WebContent**, right-click **search.html**, and select **Run As** → **Run on Server**.
5. Select **Choose an existing server** and select the **v8.0** server to run the application. Then click **Finish**.
6. When the search page opens in a web browser, in the Social Security number field, enter an appropriate value, for example, `111-11-1111`, and click **Search**. If everything works correctly, you can see the customer's full name, first account, and its balance, which have been read from the memory data.
7. Test the stateless session EJB, `SimpleBankFacade`, by using the Universal Test Client (UTC). The following methods are valid:
   - `getCustomerFullName(ssn)`: Retrieves the full name (use `111-11-1111`)
   - `getNumAccounts(ssn)`: Retrieves the number of accounts
   - `getAccountId(ssn, int)`: Retrieves the account ID by index (`0`,`1`,`2`,...)
   - `getAccountBalance(accountId)`: Retrieves the balance

We now have resources in preparation for the web services sample, including a JavaBean in the `RAD8WebServiceWeb` project and a session EJB in the `RAD8WebServiceEJB` project. We use these resources as a base for developing and testing the web services examples.

Creating bottom-up web services from a JavaBean

In this section, we create a web service from an existing Java class using the bottom-up approach. The imported application contains a Java class called `SimpleBankBean`, which has various methods to get customer and account information from the bank. We can either use the Web Service wizard to generate the web service or use the annotations directly. The Web Service wizard does not inject annotations to the delegate class derived from the JavaBean. Therefore, these two approaches are essentially the same.

Creating a web service using annotations

The JAX-WS programming standard relies on the use of annotations to specify metadata that is associated with web service implementations. The standard also relies on annotations to simplify the development of web services. The JAX-WS standard supports the use of annotations that are based on several JSRs:

- *A Metadata Facility for the Java Programming Language (JSR 175)*
- *Web Services Metadata for the Java Platform (JSR 181)*
- *Java API for XML-Based Web Services (JAX-WS) 2.2 (JSR 224)*
- *Common Annotations for the Java Platform (JSR 250)*
- *Java Architecture for XML Binding (JAXB) (JSR 222)*

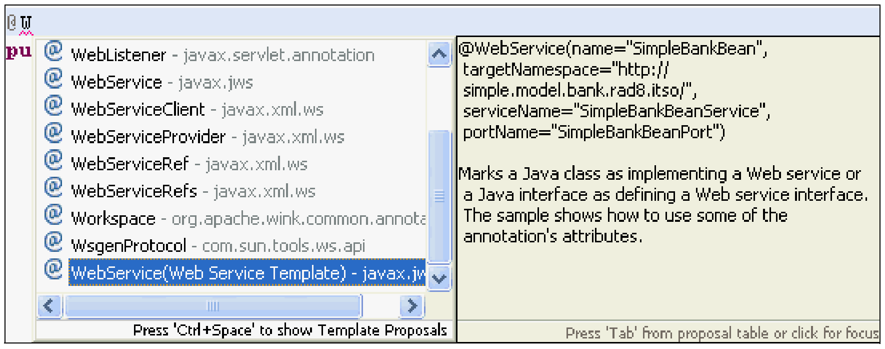
Using annotations from the JSR 181 standard, we can annotate a service implementation class or a service interface. Then we can generate a web service with a wizard or by publishing the application to a server. Using annotations within both Java source code and Java classes simplifies web service development. Using annotations in this way defines additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL into source artifacts.

In this section, we create a bottom-up web service from a JavaBean by using annotations. The web services are generated by publishing the application to a server. No wizard is required in this example.

Annotating a JavaBean

We can annotate types, methods, fields, and parameters in the JavaBean to specify a web service. To annotate the JavaBean, follow these steps:

1. In the `RAD75WebServiceWeb` project, open the **SimpleBankBean** (in `itso.rad8.bank.model.simple`).
2. Before the class declaration, type `@W` and press Ctrl+Spacebar for content assist. Scroll down to the bottom and select **WebService(Web Service Template) - javax.jws** (Figure 2).



```
@W
pu
WebListener - javax.servlet.annotation
WebService - javax.jws
WebServiceClient - javax.xml.ws
WebServiceProvider - javax.xml.ws
WebServiceRef - javax.xml.ws
WebServiceRefs - javax.xml.ws
Workspace - org.apache.wink.common.annota
WsgenProtocol - com.sun.tools.ws.api
WebService(Web Service Template) - javax.jw
Press 'Ctrl+Space' to show Template Proposals
```

```
@WebService(name="SimpleBankBean",
 targetNamespace="http://
 simple.model.bank.rad8.itso/",
 serviceName="SimpleBankBeanService",
 portName="SimpleBankBeanPort")

Marks a Java class as implementing a Web service or
 a Java interface as defining a Web service interface.
 The sample shows how to use some of the
 annotation's attributes.

Press 'Tab' from proposal table or click for focus
```

Figure 2   Content assist for WebService annotation

The annotation template is added to the Java class (Example 5).

Example 5 Web service annotation template

```
@WebService(name="SimpleBankBean",
     targetNamespace="http://simple.model.bank.rad8.itso/",
     serviceName="SimpleBankBeanService", portName="SimpleBankBeanPort")
```

The `@WebService` annotation marks a Java class as implementing a web service:

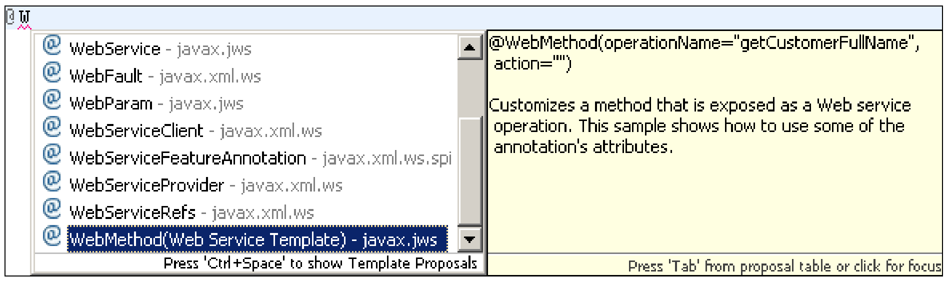
- The *name* attribute is used as the name of the `wsdl:portType` when mapped to WSDL 1.1.
- The *targetNamespace* attribute is the XML namespace used for the WSDL and XML elements generated from this web service.
- The *serviceName* attribute specifies the service name of the web service: `wsdl:service`.
- The *portName* attribute is the name of the endpoint port.

3. Change the web service name, service name, and port name, as listed in Example 6.

Example 6 Annotating a JavaBean web service

```
@WebService(name="Bank",
   targetNamespace="http://simple.model.bank.rad8.itso/",
   serviceName="BankService", portName="BankPort")
```

4. Before the `getCustomerFullName` method, type `@W` and press Ctrl+Spacebar for content assist. Scroll down to the bottom and select **WebMethod(Web Service Template) - javax.jws** (Figure 3).


@W

WebService - javax.jws
WebFault - javax.xml.ws
WebParam - javax.jws
WebServiceClient - javax.xml.ws
WebServiceFeatureAnnotation - javax.xml.ws.spi
WebServiceProvider - javax.xml.ws
WebServiceRefs - javax.xml.ws
WebMethod(Web Service Template) - javax.jws

Press 'Ctrl+Space' to show Template Proposals

@WebMethod(operationName="getCustomerFullName", action="")

Customizes a method that is exposed as a Web service operation. This sample shows how to use some of the annotation's attributes.

Press 'Tab' from proposal table or click for focus

Figure 3 Annotate method

The `@WebMethod` annotation is added to the method (Example 7).

Example 7 WebMethod template

```
@WebMethod(operationName="getCustomerFullName", action="")
```

The `@WebMethod` annotation identifies the individual methods of the Java class that are exposed externally as web service operations. In this example, we expose the `getCustomerFullName` method as a web service operation. The `operationName` is the name of the `wsdl:operation` matching this method. The `action` determines the value of the soap action for this operation.

5. Change the `operationName` and `action` (Example 8).

Example 8 @WebMethod annotation

```
@WebMethod(operationName="RetrieveCustomerName",
   action="urn:getCustomerFullName")
```

6. Annotate the method input and output (Example 9).

Example 9 Annotate the method input and output

```
@WebMethod(operationName="RetrieveCustomerName",
   action="urn:getCustomerFullName")
@WebResult(name="CustomerFullName")
public String getCustomerFullName(@WebParam(name="ssn")String ssn)
         throws CustomerDoesNotExistException
```

The `@WebParam` and `@WebResult` annotations customize the mapping of the method parameters and results to message parts and XML elements.

7. Select **Source** → **Organize Imports** (or press Ctrl+Shift+O) to resolve the imports.

Validating web service annotations

When developing web services, you can benefit from two levels of validation. The first level involves validating syntax and Java-based default values. This level of validation is performed by the Eclipse Java development tools (JDT). The second level of validation involves the implicit default checking and verification of WSDL contracts. This second level is performed by a JAX-WS annotation processor.

When you enable the annotation processor, warning and error messages for annotations are displayed similarly to Java errors. You can work with these messages in various workbench locations, such as the Problems view.

For instance, after annotating one method as `@WebMethod`, you see a QuickFix icon with the warning that is reported in Example 10.

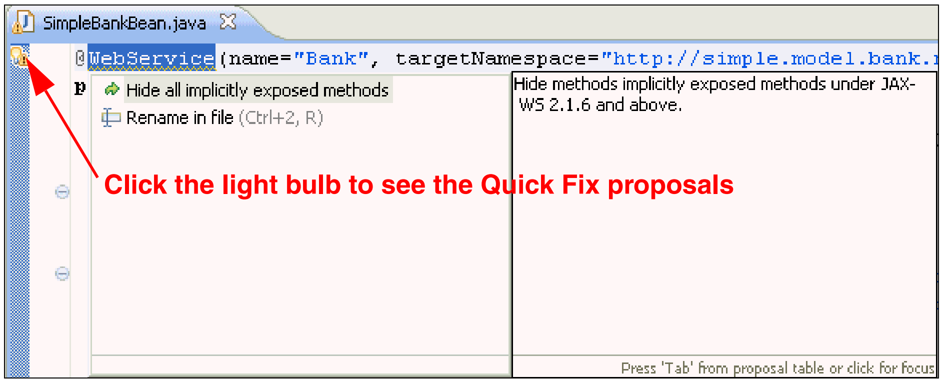
Example 10 Warning after adding @WebMethod in front of one method

```
JAX-WS 2.1.6, 3.3: The following methods will be implicitly exposed as web
methods: [BigDecimal getAccountBalance(String accountId), String
getAccountId(String customerId, int account), int getNumAccounts(String
customerId)]
```

If you click the light bulb icon corresponding to this QuickFix, you see two proposed solutions, as shown in Figure 4:

- Hide all implicitly exposed methods
- Rename in file

Select the first proposal: All mentioned methods are annotated with `@WebMethod(exclude=true)`.


SimpleBankBean.java
Hide all implicitly exposed methods
Rename in file (Ctrl+2, R)
Click the light bulb to see the Quick Fix proposals
Press 'Tab' from proposal table or click for focus

```
@WebService(name="Bank", targetNamespace="http://simple.model.bank.
```

Hide methods implicitly exposed methods under JAX-WS 2.1.6 and above.

Figure 4 QuickFix available after annotating one method with @WebMethod

Annotation processing: The annotation processing is enabled by default. To disable annotation processing, right-click the web service project in the Enterprise Explorer view and select **Properties** → **Java Compiler** → **Annotation Processing**. Clear the **Enable annotation processing** check box.

By using the annotation processor to detect problems at build time, you can prevent these problems from occurring at run time. For example, if you make the changes in Example 11, you receive validation errors, such as the errors that are shown in Example 12.

Example 11 Validating web service annotations

```
@WebService(name="!Bank", targetNamespace="simple.model.bank.rad8.itso/",
serviceName="BankService", portName="BankPort")
public class SimpleBankBean implements Serializable {
   private static final long serialVersionUID = -6375368405461558 53L;
   public SimpleBankBean() {
   }
@WebMethod(operationName="!RetrieveCustomerName",
action="urn:getCustomerFullName")
@WebResult(name="CustomerFullName")
@Oneway
   public String getCustomerFullName(@WebParam(name="ssn")String ssn)
         throws CustomerDoesNotExistException {
```

Example 12 JAX-WS annotation processor validation results

```
JSR-181, 4.3.1: Oneway methods cannot return a value
JSR-181, 4.3.1: Oneway methods cannot throw checked exceptions
name must be a valid nmToken
operationName must be a valid nmToken
targetNamespace must be a valid URI
```

Creating a web service from an annotated JavaBean by publishing to the server

After annotating a JavaBean, you can generate a web service application by publishing the application project of the bean directly to a server. When the web service is generated, no WSDL file is created in your project.

Perform these steps to create a web service from an annotated JavaBean:

1. In the Servers view, start WebSphere Application Server V8.0 (if it is not running).
2. Publish the web service project on the server. Depending on the server configuration, this step happens either automatically or manually (by right-clicking the server and selecting **Publish**).

Testing the JAX-WS web service: The Generic Service Client

To test the web service by using the Generic Service Client, follow these steps:

1. Make sure that the project is already published to the server.
2. Switch to the **Services** view that is under the Enterprise Explorer.
3. Expand the **JAX-WS** folder, right-click **RAD8WebServiceWeb: {http://simple.model.bank.rad8bank.itso/}BankService**, and select **Test with Generic Service Client** (Figure 5).

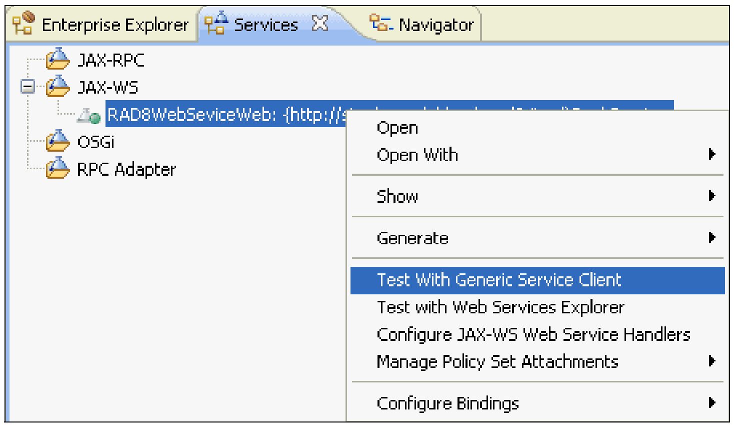

Enterprise Explorer
Services
Navigator
JAX-RPC
JAX-WS
RAD8WebSeviceWeb: {http://s
OSGi
RPC Adapter
Open
Open With
Show
Generate
Test With Generic Service Client
Test with Web Services Explorer
Configure JAX-WS Web Service Handlers
Manage Policy Set Attachments
Configure Bindings

Figure 5 Test with Generic Service Client

The Generic Service Client opens, as shown in Figure 6.

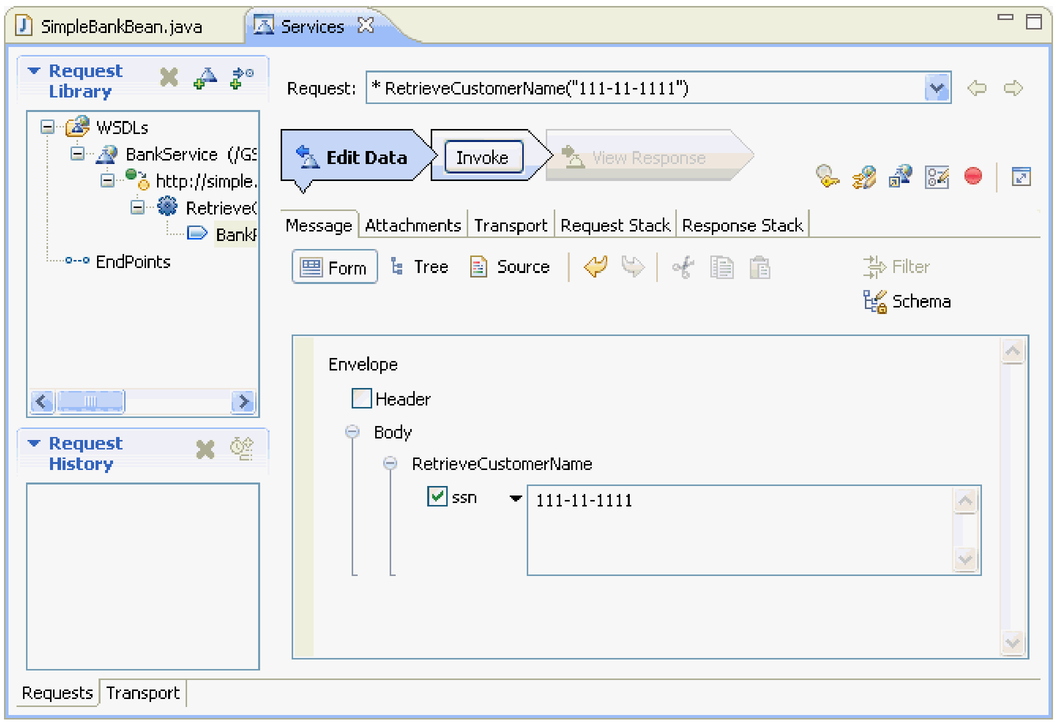


Figure 6 Generic Service Client

4. The **RetrieveCustomerName** operation is already selected.
5. Click the field **ssn**. In the ssn field, type `111-11-1111` and then click **Invoke**. The result (`Mr. Henry Cui`) is displayed in the Form pane. See Figure 7.

Tip: The Generic Service Client creates a WSDL dynamically and places it inside a hidden project called GSC Store inside the Rational Application Developer workspace. For this WSDL to have the correct URL (host name and port) to invoke the service on your WebSphere Application Server Test Environment, you must publish the project to WebSphere Application Server before invoking the GSC.

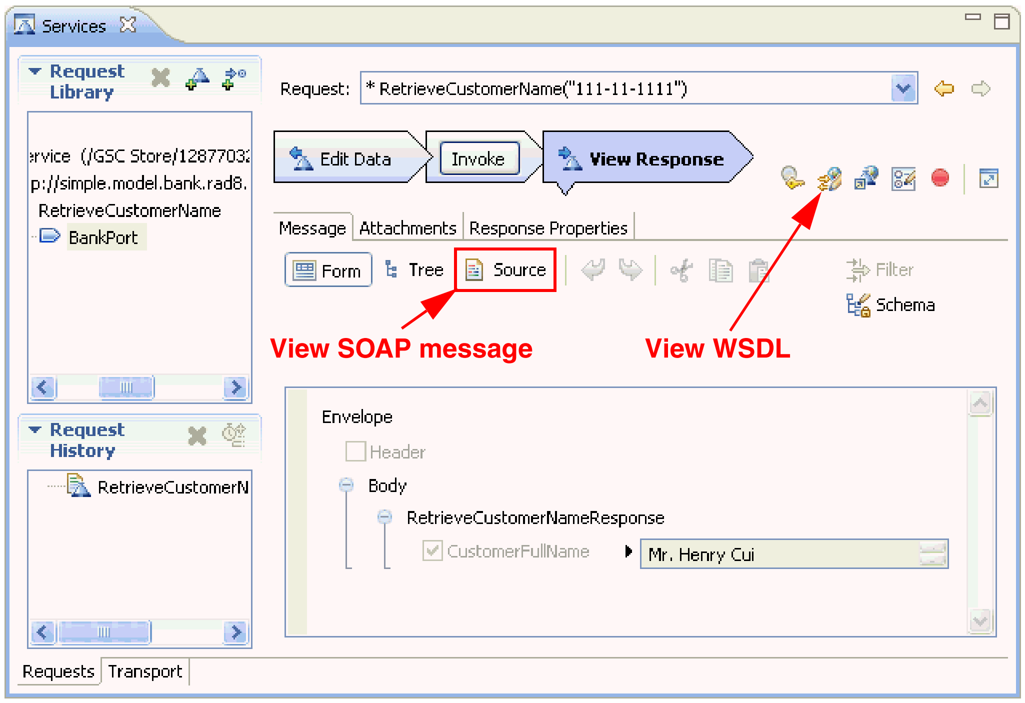


Figure 7   Results of invocation of the web service with GSC

6. Click the **Source** pane to view the SOAP messages as raw XML, as shown in Example 13.

Example 13   SOAP message

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><ns2:Re
trieveCustomerNameResponse
xmlns:ns2="http://simple.model.bank.rad8.itso/"><CustomerFullName>Mr. Henry
Cui</CustomerFullName></ns2:RetrieveCustomerNameResponse></soapenv:Body></soape
nv:Envelope>
```

Viewing the dynamically generated WSDL

In JAX-WS web services, the deployment descriptors are optional, because they use annotations. The WSDL file can be dynamically generated by the run time based on information that it gathers from the annotations added to the Java classes.

The URL for the dynamically generated WSDL is in the following format:

`http://<hostname>:<port>/<Web project context root>/<service name>?wsdl`

To view the dynamically generated WSDL, enter the following URL in the browser (`908x` is the port number, most probably `9080` or `9081`):

`http://localhost:908x/RAD8WebServiceWeb/BankService?wsdl`

Tip: You can also see the WSDL from the Generic Service Client, as shown in Figure 7.

The dynamically generated WSDL file is displayed. We also notice that the URL for the WSDL is changed:

```
http://localhost:908x/RAD75WebServiceWeb/BankService/BankService.wsdl
```

Examine the generated WSDL. We can see that the generated WSDL matches the web services annotations that we added. Example 14 shows an extract of the generated WSDL snippet.

Example 14 Dynamically generated WSDL snippet

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="BankService"
   targetNamespace="http://simple.model.bank.rad8.itso/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"xmlns:tns=
      "http://simple.model.bank.rad8.itso/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
   <types> ......
   <message> ......
   ......
   <portType name="Bank">
      <operation name="RetrieveCustomerName">
         <input message="tns:RetrieveCustomerName" />
         <output message="tns:RetrieveCustomerNameResponse" />
            <fault name="CustomerDoesNotExistException"
               message="tns:CustomerDoesNotExistException" />
      </operation>
   </portType>
   <binding name="BankPortBinding" type="tns:bank">
      <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
         <operation name="RetrieveCustomerName">
            <soap:operation soapAction="urn:getCustomerFullName" />
            <input>
               <soap:body use="literal" />
            </input>
            <output>
               <soap:body use="literal" />
            </output>
            <fault name="CustomerDoesNotExistException">
              <soap:fault name="CustomerDoesNotExistException" use="literal"/>
            </fault>
         </operation>
   </binding>
   <service name="BankService">
      <port name="BankPort" binding="tns:BankPortBinding">
         <soap:address
            location="http://localhost:9080/RAD8WebServiceWeb/BankService" />
      </port>
    </service>
</definitions>
```

To see the dynamically generated XML schema, enter the following URL:

```
http://localhost:908x/RAD8WebServiceWeb/BankService/BankService_schema1.xsd
```

For a simple test to verify that the web service is running in the server, enter the following URL:

```
http://localhost:908x/RAD8WebServiceWeb/BankService
```

The following result is displayed in the browser:

```
{http://simple.model.bank.rad8.itso/}BankService
Hello! This is an Axis2 Web Service!
```

Creating web services using the Web Service wizard

The Web Service wizard assists you in creating a new web service, configuring it for deployment, and deploying the web service to a server. To create a web service from a JavaBean, follow these steps:

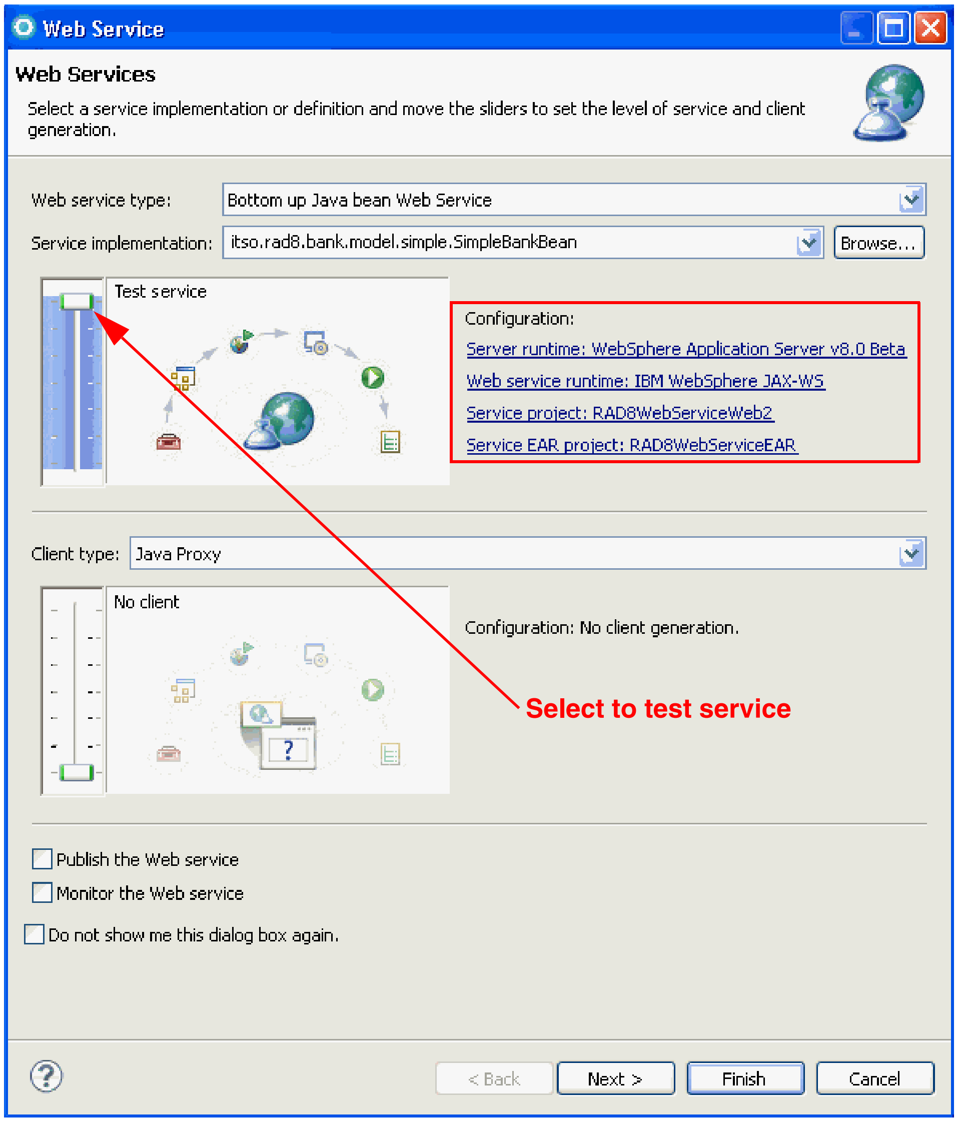
1. In the Java EE perspective, expand **RAD8WebServiceWeb2** → **Java Resources: src** → **itso.rad8.bank.model.simple**. Right-click **SimpleBankBean.java** and select **Web Services** → **Create Web service**. The Web Service wizard starts.
2. In the Web Services window, select the following options:
   a. For Web service type, ensure that **Bottom up Java bean Web Service** is selected (default).
   b. Under Service implementation, move the slider to the **Test** position (top) to access testing options for the service on subsequent windows.

The slider: The slider offers a more granular division of web services development. By using the slider, you can select from the following stages of web services development:

Develop	Develops the WSDL definition and implementation of the web service. It includes tasks, such as creating the modules that will contain the generated code, WSDL files, deployment descriptors, and Java files when appropriate.
Assemble	Ensures that the project that hosts the web service or client is associated with an EAR when required by the target application server.
Deploy	Creates the deployment code for the service.
Install	Installs and configures the web module and EAR files on the target server. If any changes to the endpoints of the WSDL file are required, they are made in this stage.
Start	Starts the web service after the service is installed on the server.
Test	Provides various options for testing the service, such as using the Web Services Explorer or sample JSP.

c. Ensure that the following server-side configurations are selected, as shown in Figure 8:

- Server runtime: **WebSphere Application Server v8.0 Beta**
- Web service run time: **IBM WebSphere JAX-WS**
- Service project: **RAD8WebServiceWeb2**
- Service EAR project: **RAD8WebServiceEAR**


Web Service
Web Services
Select a service implementation or definition and move the sliders to set the level of service and client generation.
Test service
No client
Publish the Web service
Monitor the Web service
Do not show me this dialog box again.
Browse...
< Back
Next >
Finish
Cancel
Select to test service

Web service type: Bottom up Java bean Web Service

Service implementation: itso.rad8.bank.model.simple.SimpleBankBean

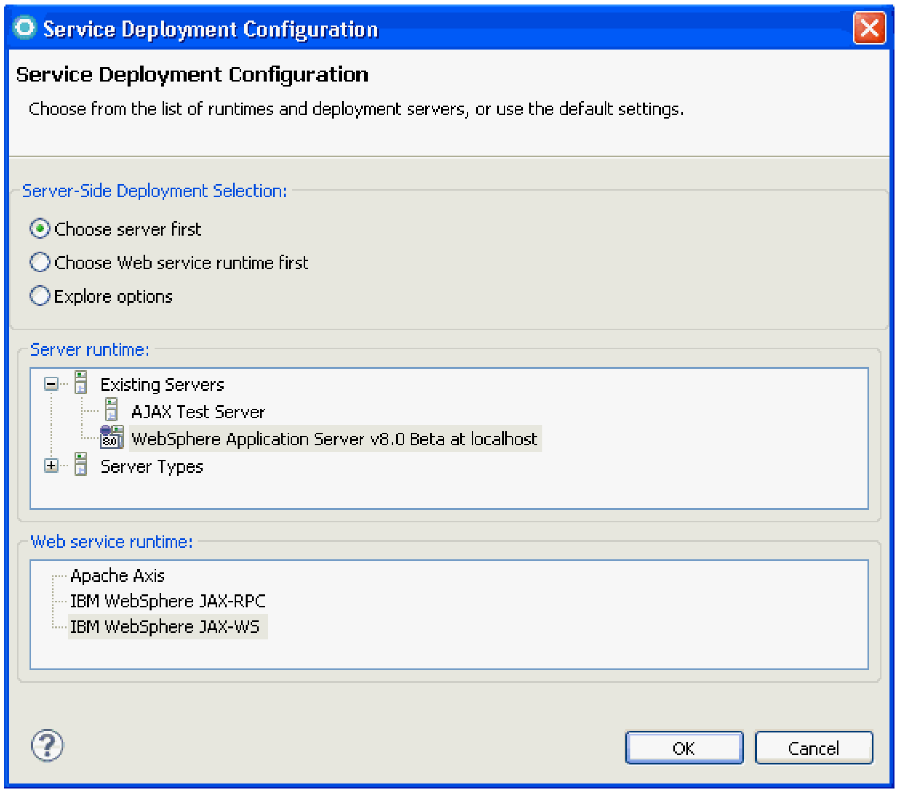
Configuration:
Server runtime: WebSphere Application Server v8.0 Beta
Web service runtime: IBM WebSphere JAX-WS
Service project: RAD8WebServiceWeb2
Service EAR project: RAD8WebServiceEAR

Client type: Java Proxy

Configuration: No client generation.

Figure 8 Web Services dialog window

d. Under Configuration, if you click the **Server: WebSphere Application Server v8.0 Beta** link, the Service Deployment Configuration window (Figure 9 on page 24) opens. In this window, you can select the server and run time. We use the default settings of this window. Click **Cancel** to close the window and return to the Web Services window.


Service Deployment Configuration

Service Deployment Configuration

Choose from the list of runtimes and deployment servers, or use the default settings.

Server-Side Deployment Selection:

[✓ Choose server first]

Choose Web service runtime first

Explore options

Server runtime:

- Existing Servers
  - AJAX Test Server
  - WebSphere Application Server v8.0 Beta at localhost
- Server Types

Web service runtime:

- Apache Axis
- IBM WebSphere JAX-RPC
- IBM WebSphere JAX-WS

Figure 9 Web Services wizard: Service Deployment Configuration

e. Clear the **Publish the Web service** check box (because we do not publish to a Universal Description, Discovery, and Integration (UDDI) registry).

f. Clear the **Monitor the Web service** check box (because we select to monitor the web service later).

g. Click **Next**.

3. In the WebSphere JAX-WS Bottom Up Web Service Configuration window (Figure 10 on page 25):

a. For Delegate class name, accept the default (**SimpleBankBeanDelegate**).

The delegate class is a wrapper that contains all the methods from the JavaBean and the JAX-WS annotation that the run time recognizes as a web service.

b. For Java to WSDL mapping style, accept the default.

The style defines the encoding style for messages that are sent to and from the web service. The recommended WSDL style is **Document Wrapped**.

c. Select **Generate WSDL file into the project**.

Because the annotations in the delegate class are used to indicate to the run time that the bean is a web service, a static WSDL file is no longer generated to your project automatically. The run time can dynamically generate a WSDL file from the information in the bean. Select this option to generate a static WSDL file for the web service. There are several reasons to select this option:

- Performance improvements. For a large bean with lots of methods and complex data types, this option prevents the penalty of the initial generation by the run time when the service is accessed.
- This option is required for SOAP 1.2.

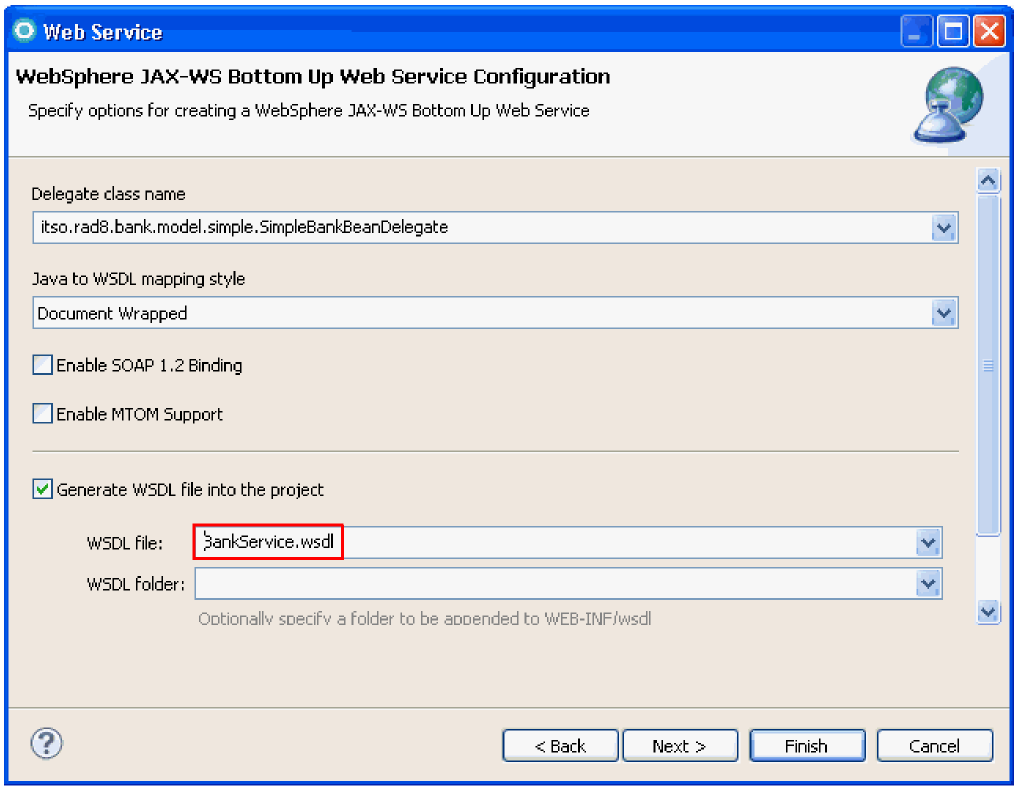
- To enforce a contract with the bean using `@WebService.wsdlLocation`. The JAX-WS annotations processor will validate the WSDL against the bean.

Change the name of the generated WSDL to **BankService.wsdl**.

d. Select **Generate Web service deployment descriptor**.

For JAX-WS web services, deployment information is generated dynamically by the run time; static deployment descriptors are optional. Selecting this check box generates the deployment descriptors.

e. Click **Next**.


Web Service

WebSphere JAX-WS Bottom Up Web Service Configuration

Specify options for creating a WebSphere JAX-WS Bottom Up Web Service

Delegate class name: itso.rad8.bank.model.simple.SimpleBankBeanDelegate

Java to WSDL mapping style: Document Wrapped

☐ Enable SOAP 1.2 Binding

☐ Enable MTOM Support

☑ Generate WSDL file into the project

WSDL file: BankService.wsdl

WSDL folder:

Optionally specify a folder to be appended to WEB-INF/wsdl

Figure 10 WebSphere JAX-RS Bottom Up Web Service Configuration

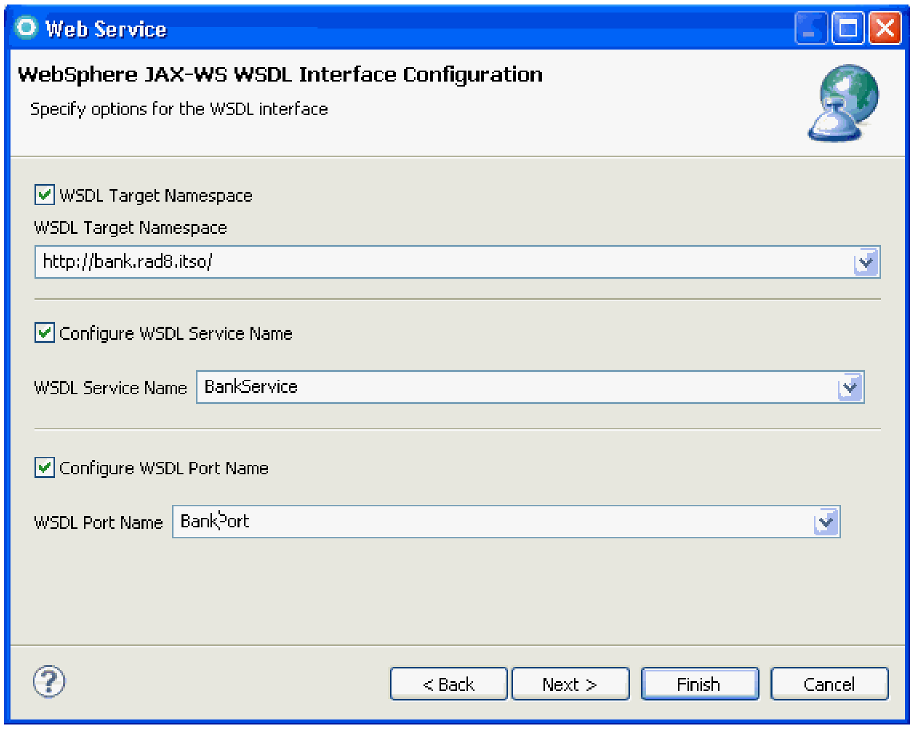
4. In the WebSphere JAX-WS WSDL Configuration window (Figure 11 on page 26), perform these tasks:

a. Select **WSDL Target Namespace**, and for the WSDL Target Namespace, enter `http://bank.rad8.itso/`.

b. Select **Configure WSDL Service Name**, and for the WSDL Service Name, enter `BankService`.

c. Select **Configure WSDL Port Name**, and for the WSDL Port Name, enter `BankPort`.

d. Click **Next**.


Web Service

WebSphere JAX-WS WSDL Interface Configuration

Specify options for the WSDL interface

☑ WSDL Target Namespace

WSDL Target Namespace: http://bank.rad8.itso/

☑ Configure WSDL Service Name

WSDL Service Name: BankService

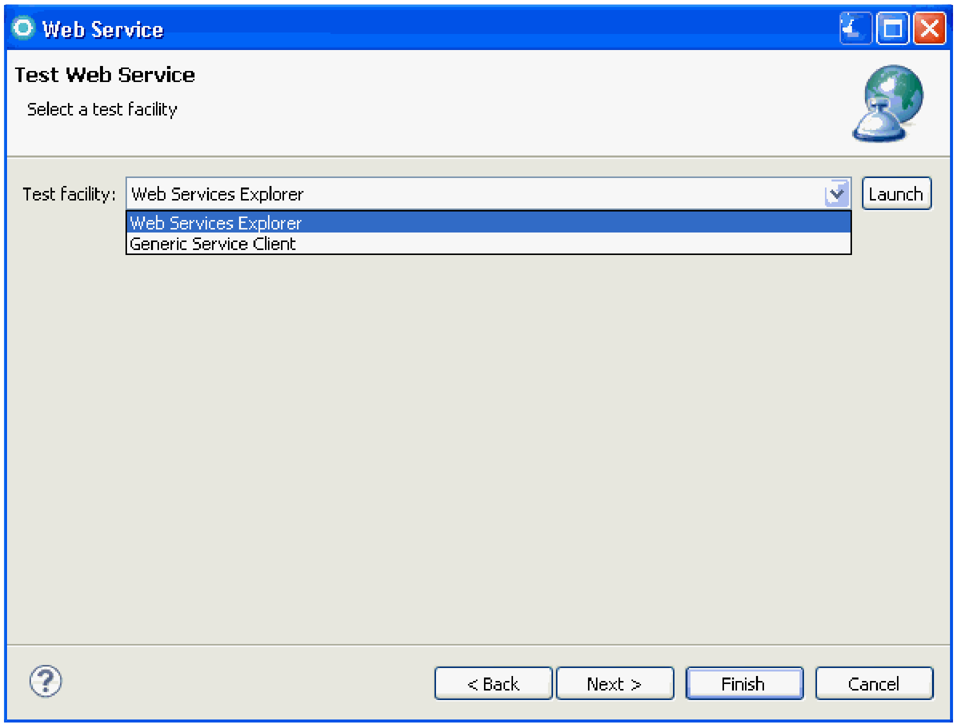
☑ Configure WSDL Port Name

WSDL Port Name: BankPort

Figure 11   WebSphere JAX-WS WSDL Interface Configuration

The web service is generated and deployed to the server.

5. In the Test Web Service window (Figure 12), which opens because we moved the slider for the service to the Test position, select the **Generic Service Client** and click **Launch**.


Web Service

Test Web Service

Select a test facility

Test facility: Web Services Explorer

Web Services Explorer
Generic Service Client

Figure 12   Select the Generic Service Client as a test facility

6. The Generic Service Client opens in an external web browser (see Figure 13). Complete these tasks:
   a. Select the **getNumAccounts** operation and click **Add**.
   b. Enter a value for the customer ID, such as `111-11-1111`, and click **Go**.

   The result `2` is displayed in the status pane. Optional: Try other operations.
7. Close the Web Services Explorer. Click **Finish** to exit the Web Service wizard.

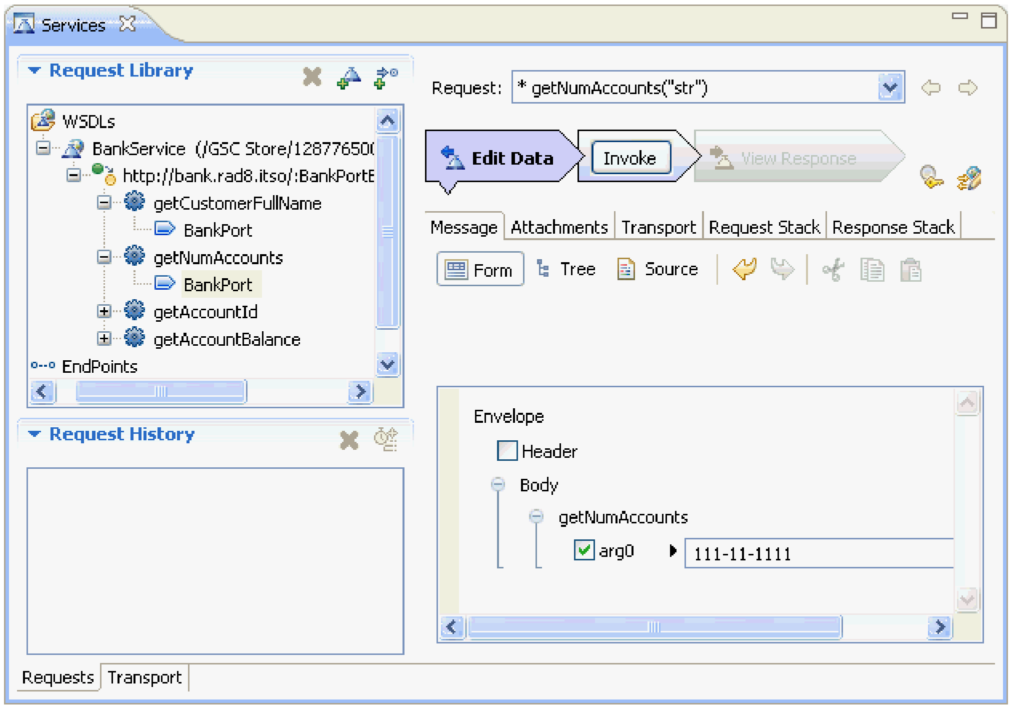


Figure 13 Test getNumAccounts with Generic Service Client

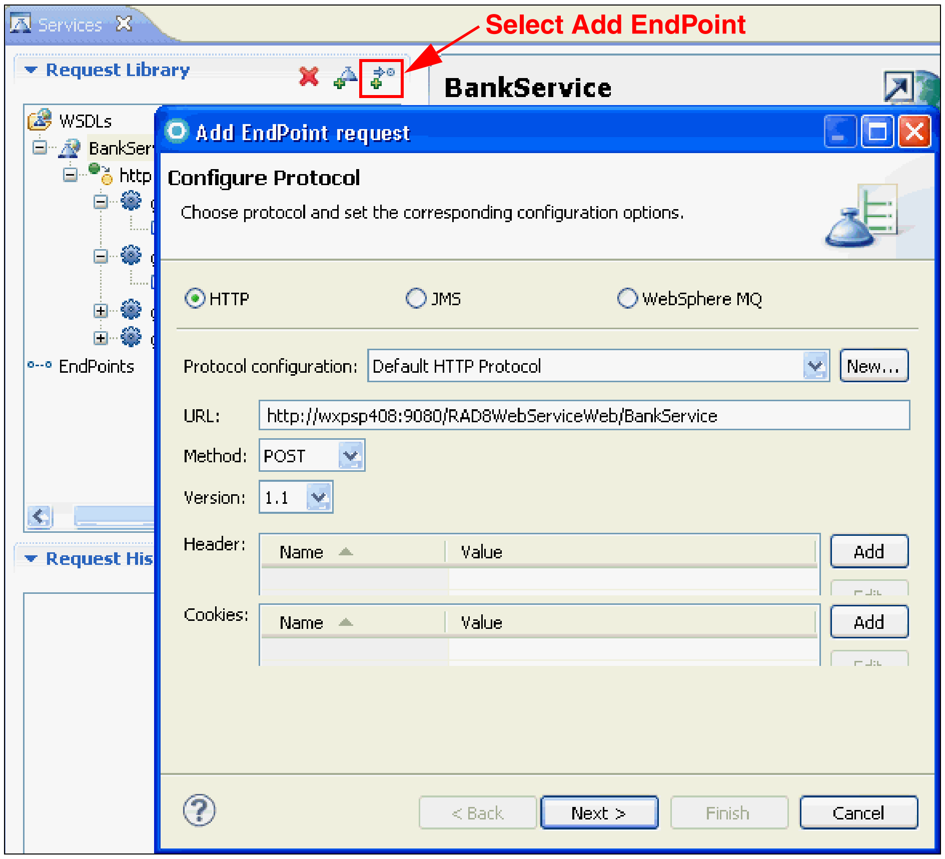
The web services are available at two endpoints: the HTTP endpoint and the HTTPS endpoint. If your server is not secured, the endpoint is:

```
http://localhost:908x/RAD8WebServiceWeb2/BankService
```

If your server is secured, the web service listed in the Generic Server Client has the following endpoint:

```
https://localhost:944x/RAD8WebServiceWeb2/BeanService
```

You can see the current endpoint by selecting the **Add EndPoint Request** icon, as shown in Figure 14.



Protocol configuration: Default HTTP Protocol

URL: http://wxpsp408:9080/RAD8WebServiceWeb/BankService

Method: POST

Version: 1.1

Figure 14 Add EndPoint request

To test the HTTPS protected web service with the Generic Service Client, you can configure a new protocol configuration. The signer certificate from the WebSphere Application Server must be imported into the Eclipse truststore. For more information about creating a new Secure Sockets Layer (SSL) configuration, see this website:

http://publib.boulder.ibm.com/infocenter/radhelp/v8/index.jsp?topic=/com.ibm.rational.ttt.common.doc/topics/tgsccreatessl.html

You have successfully created web services from a JavaBean.

Resources generated by the Web Service wizard

After code generation, examine the generated code.

You can see that the wizard generates the following artifacts:

- A delegate class named `SimpleBankBeanDelegate`. The delegate class is a wrapper that contains all the methods from the JavaBean and the JAX-WS annotation that the run time recognizes as a web service. The annotation `@javax.jws.WebService` in the delegate class tells the server runtime environment to expose all public methods on that bean as a web service. The `targetNamespace`, the `serviceName`, and the `portName` are what we specified in the Web Service wizard.

```
@javax.jws.WebService (targetNamespace="http://bank.rad8.itso/",
   serviceName="BankService", portName="BankPort",
    wsdlLocation="WEB-INF/wsdl/BankService.wsdl")
```

- A `webservices.xml` file in the `WebContent/WEB-INF` folder. This file is the optional web services deployment descriptor. A deployment descriptor can be used to override or enhance the information provided in the service. For example, if the `<wsdl-service>` element is provided in the deployment descriptor, the namespace used in this element overrides the `targetNamespace` member attribute in the annotation.
- A WSDL file (`BankService.wsdl`) and an XSD file (`BankService_schema1.xsd`) in the `WEB-INF/wsdl` folder. If you plan to create the client at a later time or publish the WSDL for other users, you can use this WSDL file.

You can locate the projects developed up to this point in the `4884codesolution\webservices` folder in the `RAD8WebServiceImplemented.zip` file.

Creating a synchronous web service JSP client

The Web Service Client wizard assists you in generating a JavaBean proxy and a sample application. The sample web application demonstrates how to invoke the web services proxy. You can invoke the web services using the JAX-WS synchronous model, or the asynchronous model. In the section, we generate a synchronous web service client.

Generating and testing the web service client

To generate a client and test the client proxy, follow these steps:

1. Switch to the **Services** view, right-click **RAD8WebServiceWeb**, and select **Generate** → **Client**, as shown in Figure 15.

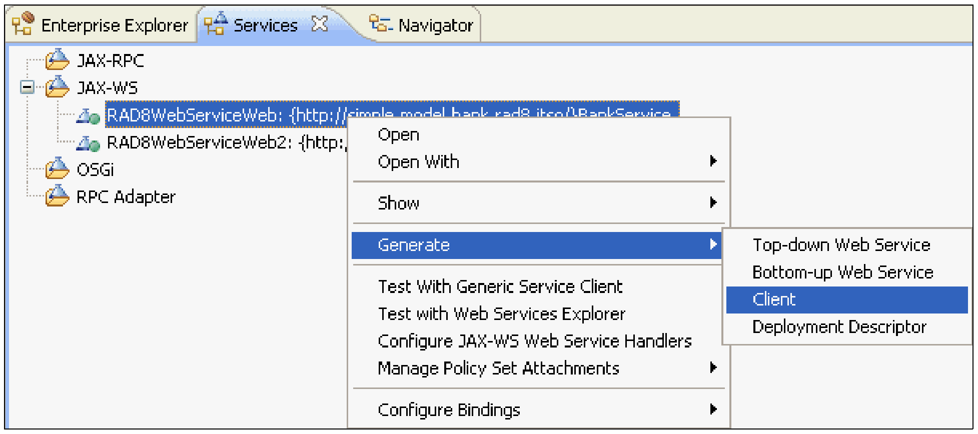
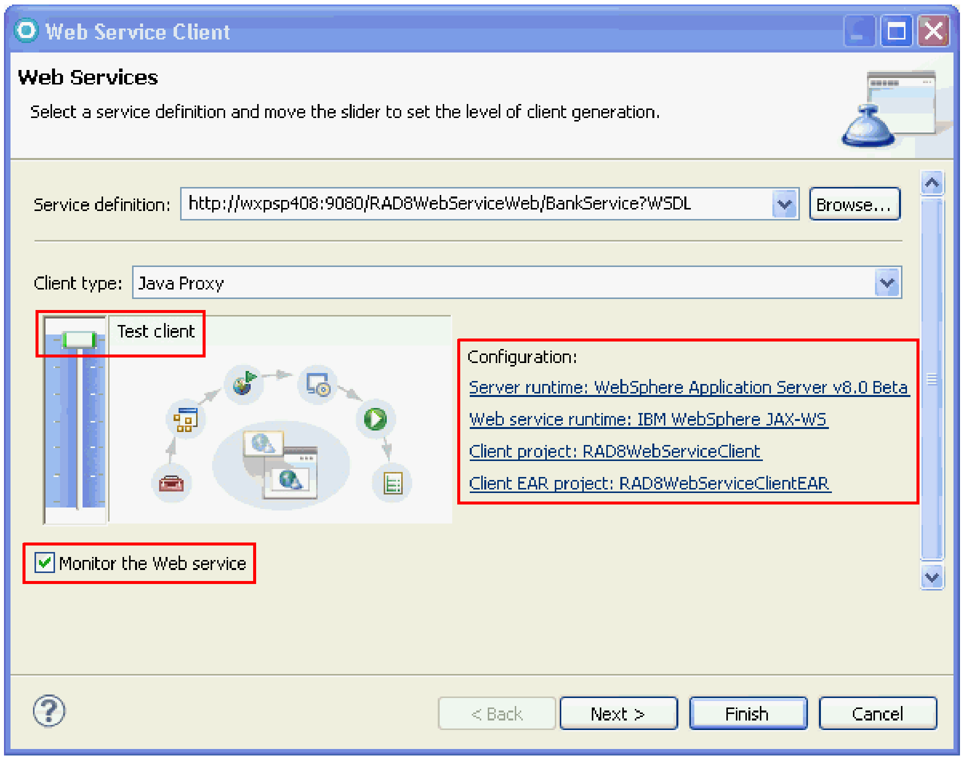

Enterprise Explorer
Services
Navigator
JAX-RPC
JAX-WS
RAD8WebServiceWeb: {http://simple.model.bank.rad8.itso/}BankService
RAD8WebServiceWeb2: {http:
OSGi
RPC Adapter
Open
Open With
Show
Generate
Test With Generic Service Client
Test with Web Services Explorer
Configure JAX-WS Web Service Handlers
Manage Policy Set Attachments
Configure Bindings
Top-down Web Service
Bottom-up Web Service
Client
Deployment Descriptor

Figure 15 Invoking Generate Client

2. In the Web Services Client window (Figure 16 on page 30), perform these steps:
   a. Move the slider up to the **Test** position. This position provides options for testing the service using a JSP-based sample application.
   b. Select **Monitor the Web service**.
   c. Place the web service and web service client in separate web and EAR projects. Click **Client project**.

3. In the Specify Client Project Settings window, complete the following actions and then click **Next**:
   a. Change the client project name to `RAD8WebServiceClient`.
   b. For Project type, accept **Dynamic Web project**.
   c. For Client EAR project name, accept **RAD8WebServiceClientEAR**.
   d. Click **OK**. The wizard creates the Web and EAR projects.


Web Service Client

Web Services

Select a service definition and move the slider to set the level of client generation.

Service definition: http://wxpsp408:9080/RAD8WebServiceWeb/BankService?WSDL

Client type: Java Proxy

Test client

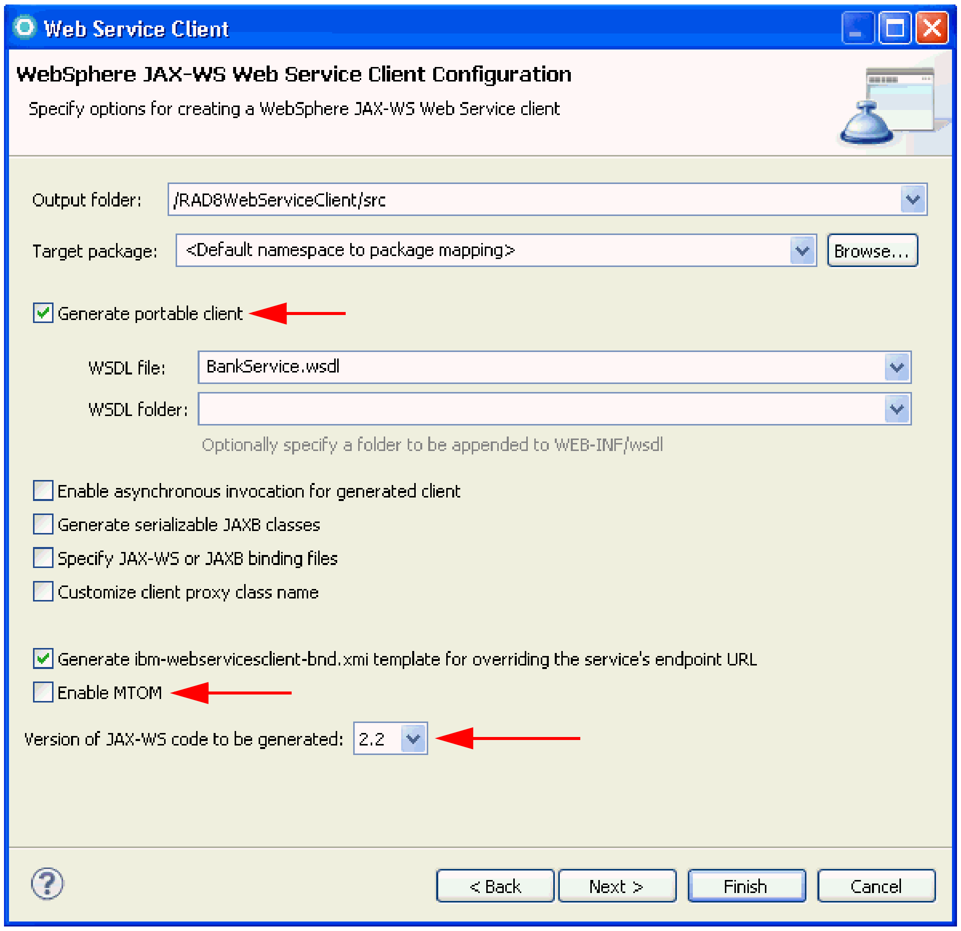
Configuration:
Server runtime: WebSphere Application Server v8.0 Beta
Web service runtime: IBM WebSphere JAX-WS
Client project: RAD8WebServiceClient
Client EAR project: RAD8WebServiceClientEAR

[✓] Monitor the Web service

Figure 16   Generating the web service client

4. Perform these steps in the WebSphere JAX-WS Web Service Client Configuration window (Figure 17 on page 31):
   a. Accept the default name and location of the Deployment Descriptor.
   b. Accept **Generate Portable Client**.
   c. Clear **Enable MTOM**.
   d. The version of JAX-WS to be generated is **2.2**, by default.
   e. Click **Next**.

   The client code is generated into the new client project.



WebSphere JAX-WS Web Service Client Configuration

Specify options for creating a WebSphere JAX-WS Web Service client

Output folder: /RAD8WebServiceClient/src

Target package: <Default namespace to package mapping>

[✓] Generate portable client

WSDL file: BankService.wsdl

WSDL folder:

Optionally specify a folder to be appended to WEB-INF/wsdl

[ ] Enable asynchronous invocation for generated client

[ ] Generate serializable JAXB classes

[ ] Specify JAX-WS or JAXB binding files

[ ] Customize client proxy class name

[✓] Generate ibm-webservicesclient-bnd.xmi template for overriding the service's endpoint URL

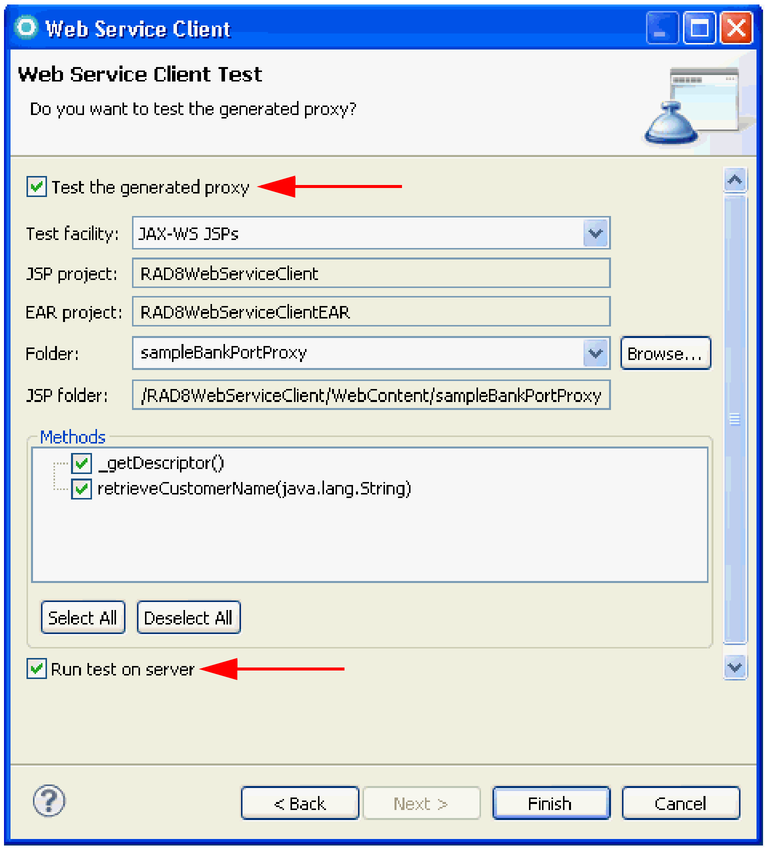
[ ] Enable MTOM

Version of JAX-WS code to be generated: 2.2

Figure 17 JAX-WS Web Service Client Configuration

5. In the Web Service Client Test window (Figure 18 on page 32), use these settings:
   a. Select **Test the generated proxy**.
   b. For Test facility, select **JAX-WS JSPs** (default).
   c. For Folder, select **sampleBankPortProxy** (default). You can specify a separate folder for the generated application if you want.
   d. Under Methods, leave all methods selected.
   e. Select the **Run test on server** check box.
   f. Click **Finish**.

   The sample application is published to the server, and the sample JSP is displayed in a Web browser.


Web Service Client

Web Service Client Test

Do you want to test the generated proxy?

✓ Test the generated proxy

Test facility: JAX-WS JSPs

JSP project: RAD8WebServiceClient

EAR project: RAD8WebServiceClientEAR

Folder: sampleBankPortProxy

JSP folder: /RAD8WebServiceClient/WebContent/sampleBankPortProxy

Methods

✓ _getDescriptor()

✓ retrieveCustomerName(java.lang.String)

✓ Run test on server

Figure 18 Web Service Client Test

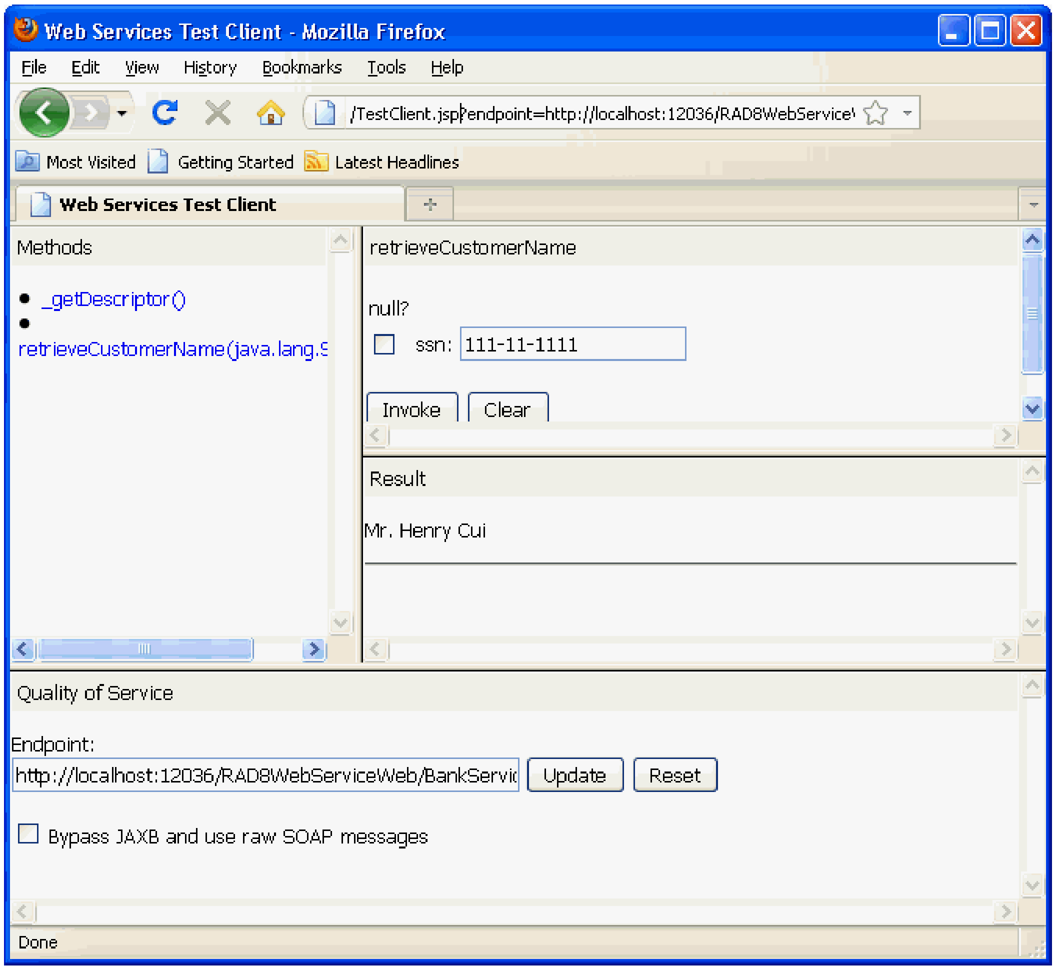
6. In the Web Services Test Client window (Figure 19 on page 33), perform these steps:
   a. Select the **retrieveCustomerName** method.
   b. Enter a valid value in the customer ID field, such as `111-11-1111`.
   c. Click **Invoke**.

   The results are displayed in the Result pane.

   Notice the endpoint in the Quality of Service pane:

   ```
   http://hostname:12036/RAD8WebServiceWeb/BankService
   ```

   You might see another port number. It depends on the port number that the wizard generated for the TCP/IP Monitor.


Web Services Test Client - Mozilla Firefox

/TestClient.jsp?endpoint=http://localhost:12036/RAD8WebService

Methods

- _getDescriptor()
- retrieveCustomerName(java.lang.S

retrieveCustomerName

null?

ssn: 111-11-1111

Result

Mr. Henry Cui

Quality of Service

Endpoint: http://localhost:12036/RAD8WebServiceWeb/BankServic

Bypass JAXB and use raw SOAP messages

Done

Figure 19   Testing the generated JSP

The TCP/IP Monitor is also started. With the TCP/IP Monitor, you can intercept and examine the SOAP traffic that comes in and out of a web service call.

7. If you select **Window** → **Preferences** and then select **Run/Debug** → **TCP/IP Monitor**, you can see that a new monitor has been added. It is configured to listen to the same local port number (12036).

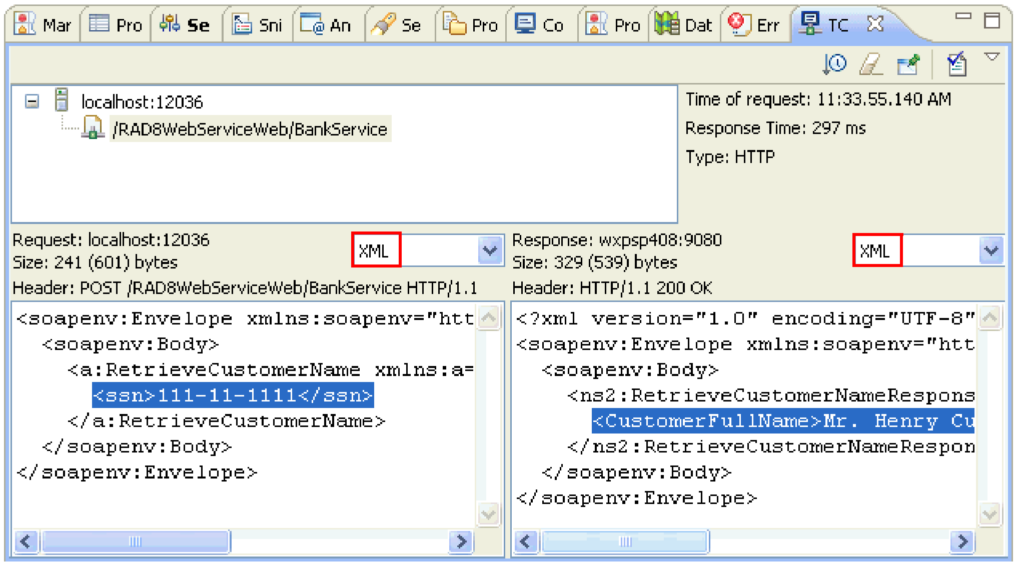
   The TCP/IP Monitor is started and ready to listen to the SOAP request and direct it to the web service provider (possibly on a separate host and at port 908*x*).

Monitor the Web service: When you select the “Monitor the Web service” option in the Web Service window, the Web Service Client wizard dynamically creates the TCP/IP Monitor. It uses an algorithm to locate an available listening port for the monitor. The sample JSP client window uses the URL to dynamically set the web service endpoint to match the monitor port. Using the wizard to create the TCP/IP Monitor is convenient, because you do not have to spend time determining how to redirect the SOAP request to the TCP/IP Monitor, especially when monitoring remote web services.

All requests and responses are routed through the TCP/IP Monitor and are displayed in the TCP/IP Monitor view. The TCP/IP Monitor view might be displayed in the same pane as the Servers view.

The TCP/IP Monitor view shows all the intercepted requests in the top pane, and when a request is selected, the messages passed in each direction are shown in the bottom panes (the request in the left pane, and the response in the right pane). The TCP/IP Monitor can be a useful tool in debugging web services and clients.

8. Select the **XML** view to see the SOAP request and response in XML format, as shown in Figure 20.



localhost:12036
/RAD8WebServiceWeb/BankService

Time of request: 11:33.55.140 AM
Response Time: 297 ms
Type: HTTP

Request: localhost:12036
Size: 241 (601) bytes
Header: POST /RAD8WebServiceWeb/BankService HTTP/1.1
XML

Response: wxpsp408:9080
Size: 329 (539) bytes
Header: HTTP/1.1 200 OK
XML

```
<soapenv:Envelope xmlns:soapenv="htt
  <soapenv:Body>
    <a:RetrieveCustomerName xmlns:a=
      <ssn>111-11-1111</ssn>
    </a:RetrieveCustomerName>
  </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8"
<soapenv:Envelope xmlns:soapenv="htt
  <soapenv:Body>
    <ns2:RetrieveCustomerNameRespons
      <CustomerFullName>Mr. Henry Cu
    </ns2:RetrieveCustomerNameRespon
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 20 TCP/IP Monitor

9. Optional: To ensure that the web service SOAP traffic is WS-I compliant, you can generate a log file by clicking the icon in the upper-right corner. In the window that opens, select a name for the log file and specify where you want to store it (for example, in the client project).

   The log file is validated for WS-I compliance. You see a confirmation message, "`The WS-I Message Log file is valid`." You can open the log file in an XML editor to examine its contents.

To stop the TCP/IP Monitor, perform these steps:

1. Select **Window** → **Preferences**.
2. Select **Run/Debug** → **TCP/IP Monitor**.
3. Select the TCP/IP Monitor from the list and select **Stop**.

Manually starting the TCP/IP Monitor: To start the TCP/IP Monitor manually, remember that the Local Monitoring port is a randomly chosen free port on localhost, while the host and port refer to the actual parameters of the server where your service is running. To test the service through the monitor, you have to manually change the host and port in the endpoint of the service you are testing, so that your request is sent to the monitor instead of the actual server.

Resources generated by the Web Service Client wizard

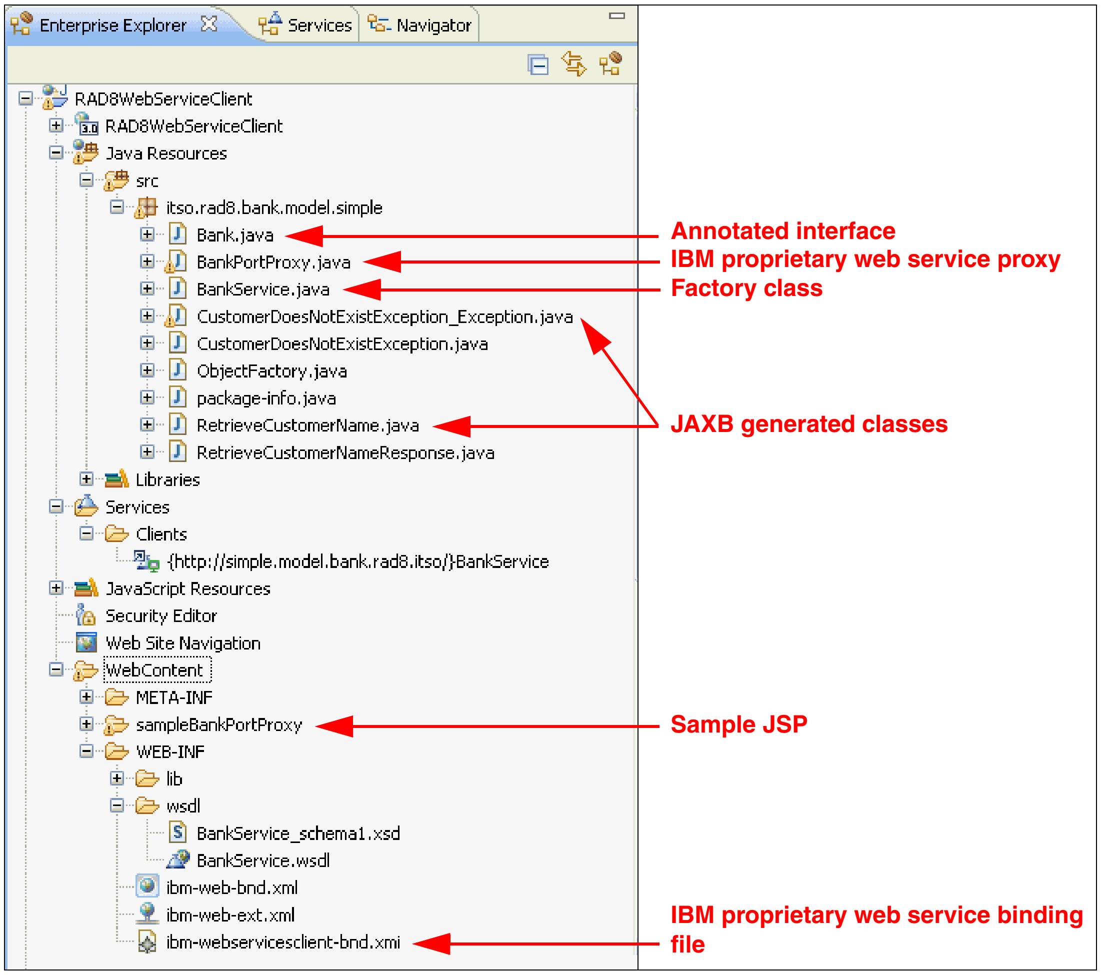
Figure 21 on page 36 shows the generated web service client artifacts.

We provide a description of each of the web service client artifacts here:

- `Bank.java` is the annotated service interface based on the WSDL to Java mapping.
- `BankService.java` is generated from the WSDL service. It is a factory class that returns an instance that implements the service's interface, which is also known as a *JAX-WS Service class*. In JAX-RPC, this implementation class is called a *stub*. In JAX-WS, no stub class exists; the stub is a class that is dynamically generated from WSDL.
- `BankPortProxy.java` is an IBM-proprietary proxy class. JAX-WS does not define this class. It is a convenience class that implements the web service's interface and hides programming details, such as the service factory and binding provider calls.
- The rest of the Java classes are the JAXB artifacts that are based on the schema types used by the WSDL.
- The `sampleBankPortProxy` folder contains the generated sample JSP, which demonstrates how to invoke the web services proxy.
- The `ibm-webservicesclient-bnd.xmi` file is the IBM proprietary Web Services Client binding file.
- The `web.xml` is no longer required and was not generated.

You can get the results in this file:

```
4884codesolution\webservices\RAD8WebServiceJSPClient.zip
```



```
RAD8WebServiceClient
  RAD8WebServiceClient
  Java Resources
    src
      itso.rad8.bank.model.simple
        Bank.java
        BankPortProxy.java
        BankService.java
        CustomerDoesNotExistException_Exception.java
        CustomerDoesNotExistException.java
        ObjectFactory.java
        package-info.java
        RetrieveCustomerName.java
        RetrieveCustomerNameResponse.java
    Libraries
  Services
    Clients
      {http://simple.model.bank.rad8.itso/}BankService
  JavaScript Resources
  Security Editor
  Web Site Navigation
  WebContent
    META-INF
    sampleBankPortProxy
    WEB-INF
      lib
      wsdl
        BankService_schema1.xsd
        BankService.wsdl
      ibm-web-bnd.xml
      ibm-web-ext.xml
      ibm-webservicesclient-bnd.xmi
```

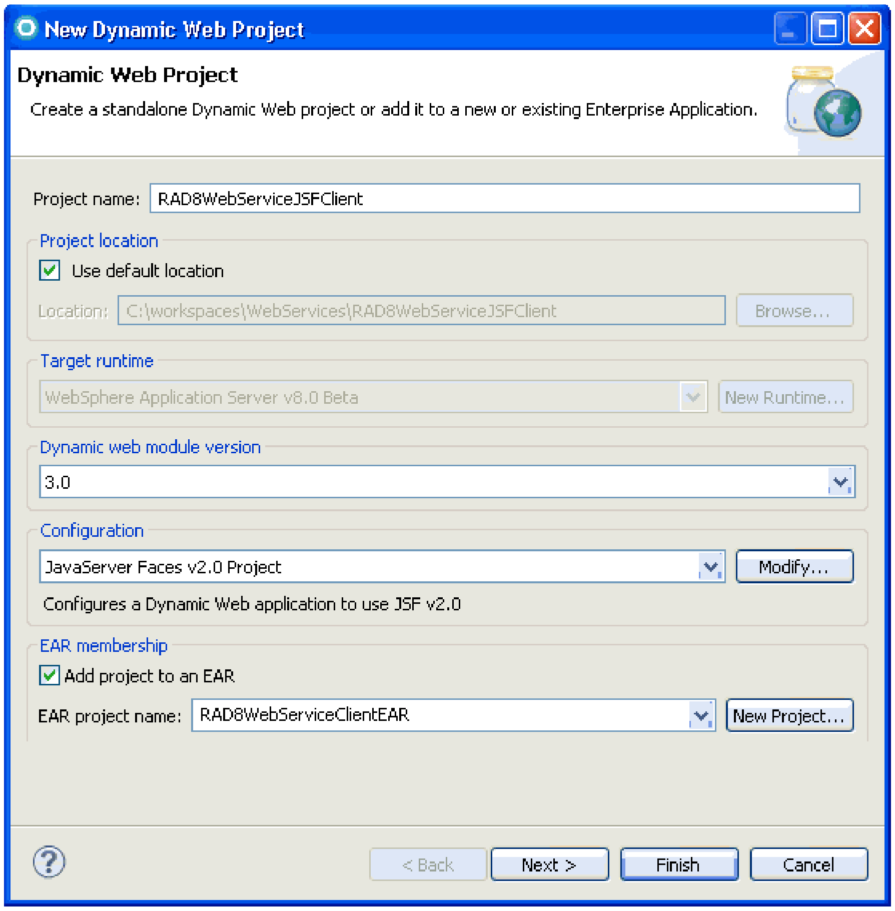
Annotated interface
IBM proprietary web service proxy
Factory class
JAXB generated classes
Sample JSP
IBM proprietary web service binding file

Figure 21 Generated web service client artifacts

Creating a web service JavaServer Faces client

With the Web Service Discovery window, you can discover a web service that exists online or in the workspace, create a proxy to the web service, and then place the methods of the proxy on a Faces JSP file:

1. Click **Add and Remove Projects** to remove the `RAD8WebServiceClientEAR` from the server. (We add a project to the EAR and automatic publishing interferes.) Alternatively, expand the server, right-click the project, and select **Remove**.
2. Create a dynamic web project by selecting **File** → **New** → **Dynamic Web Project**.
3. In the Dynamic Web Project window (Figure 22), complete the following steps:
   a. For Project name, enter `RAD8WebServiceJSFClient`.
   b. In the Configuration section, select **JavaServer Faces v2.0 Project** to add the required JSF facets to the project facets list.
   c. For EAR Project Name, select **RAD8WebServiceClientEAR**.
   d. Click **Finish**.



Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: RAD8WebServiceJSFClient

Project location

☑ Use default location

Location: C:\workspaces\WebServices\RAD8WebServiceJSFClient

Target runtime

WebSphere Application Server v8.0 Beta

Dynamic web module version

3.0

Configuration

JavaServer Faces v2.0 Project

Configures a Dynamic Web application to use JSF v2.0

EAR membership

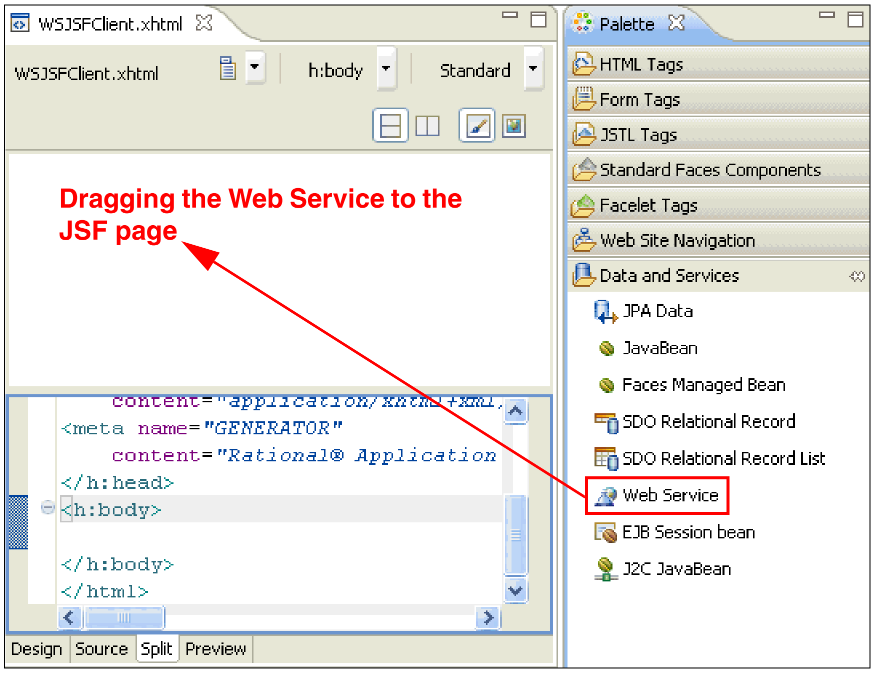
☑ Add project to an EAR

EAR project name: RAD8WebServiceClientEAR

Figure 22 Create a new JSF 2.0 project

4. If you are prompted to open the Web perspective, click **Yes**.
5. In the `RAD8WebServiceJSFClient` project, right-click **WebContent** and select **New** → **Web Page**.
6. For File name, enter `WSJSFClient`. For Basic template, select **Facelet** and click **Finish**. The `WSJSFClient.jsp` opens in an editor.
7. Select the **Design** or **Split** tab.

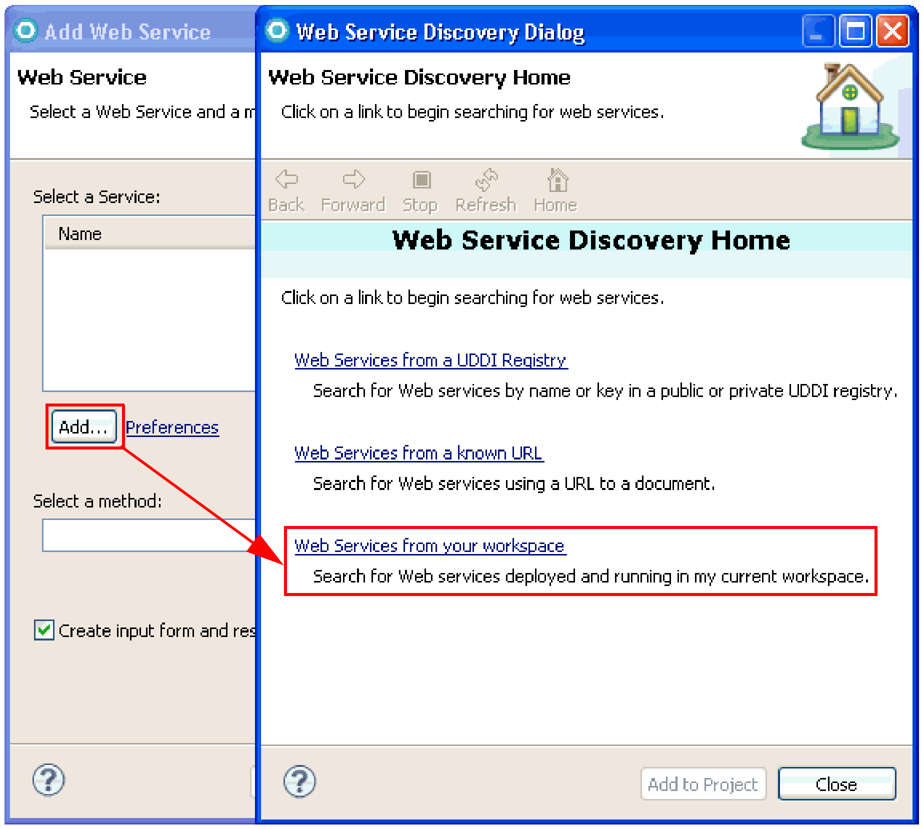
8. In the Palette, select the **Data and Services** category. Select **Web Service** and click the JSF page, as shown in Figure 23.



```
    content="application/xhtml+xml;
<meta name="GENERATOR"
    content="Rational® Application
</h:head>
<h:body>

</h:body>
</html>
```

Figure 23   Dragging the Web Service to the JSF page

9. In the Add Web Service window, as shown in Figure 24, click **Add**. In the Web Services Discovery window, select **Web services from your workspace**.



Web Service Discovery Home

Click on a link to begin searching for web services.

Web Service Discovery Home

Click on a link to begin searching for web services.

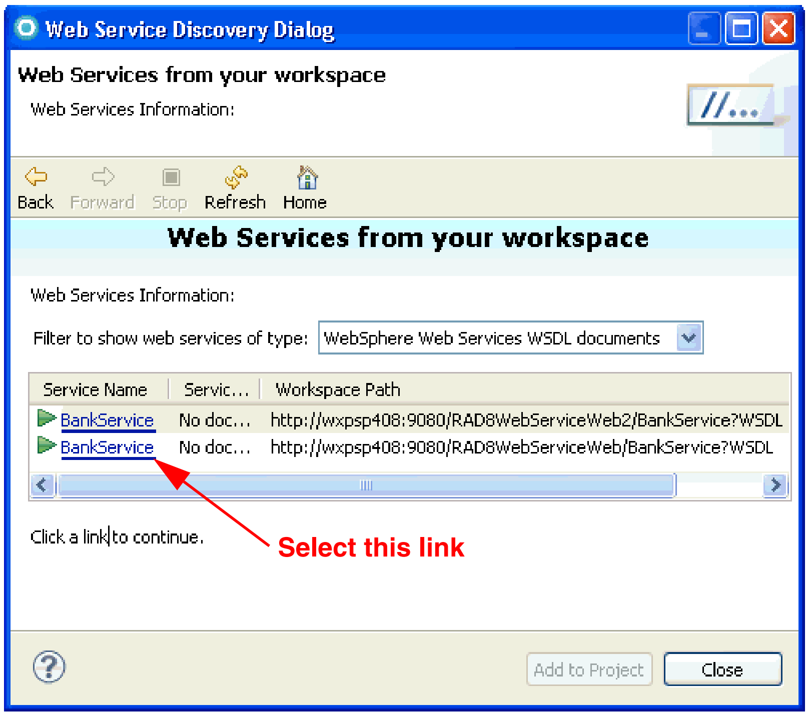
Web Services from a UDDI Registry
Search for Web services by name or key in a public or private UDDI registry.

Web Services from a known URL
Search for Web services using a URL to a document.

Web Services from your workspace
Search for Web services deployed and running in my current workspace.

Figure 24 Adding Web Services from your workspace

10. In the Web Services from your workspace window, which is shown in Figure 25, click **BankService** with the URL of the **RAD8WebServiceWeb** project (not the `RAD8WebServiceWeb2` project).



Web Services from your workspace

Web Services Information:

Web Services from your workspace

Web Services Information:

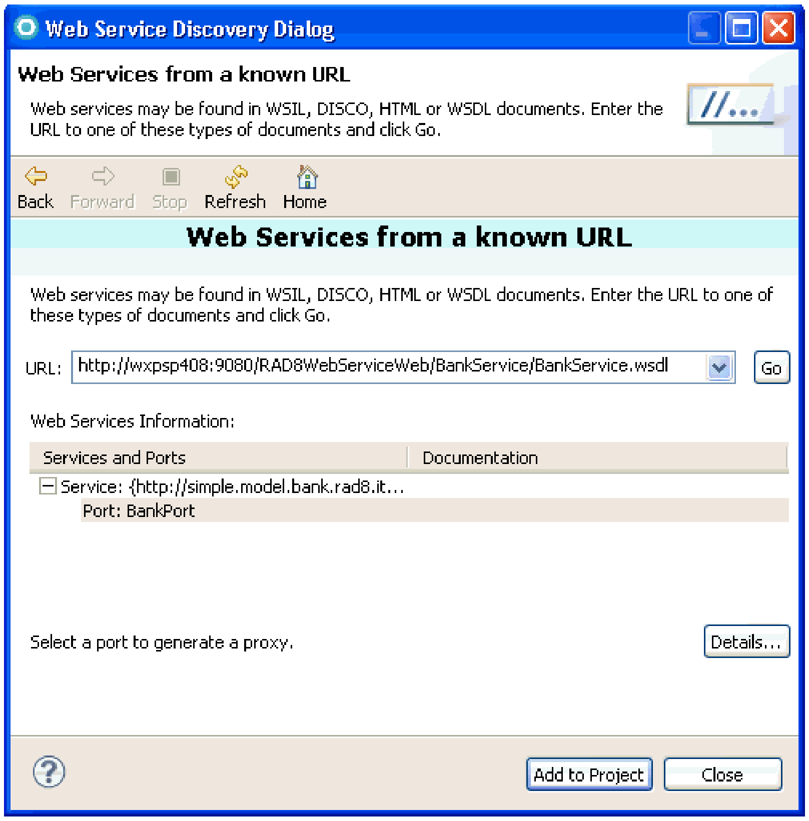
Filter to show web services of type: WebSphere Web Services WSDL documents

Service Name	Servic...	Workspace Path
BankService	No doc...	http://wxpsp408:9080/RAD8WebServiceWeb2/BankService?WSDL
BankService	No doc...	http://wxpsp408:9080/RAD8WebServiceWeb/BankService?WSDL

Click a link to continue.

Figure 25 Web Service Discovery Dialog: Web Services from your workspace

11.Select **Port: BankPort** and click **Add to Project** (Figure 26).



Web Services from a known URL

Web services may be found in WSIL, DISCO, HTML or WSDL documents. Enter the URL to one of these types of documents and click Go.

Web Services from a known URL

Web services may be found in WSIL, DISCO, HTML or WSDL documents. Enter the URL to one of these types of documents and click Go.

URL: http://wxpsp408:9080/RAD8WebServiceWeb/BankService/BankService.wsdl

Web Services Information:

Services and Ports	Documentation
Service: {http://simple.model.bank.rad8.it...	
Port: BankPort	

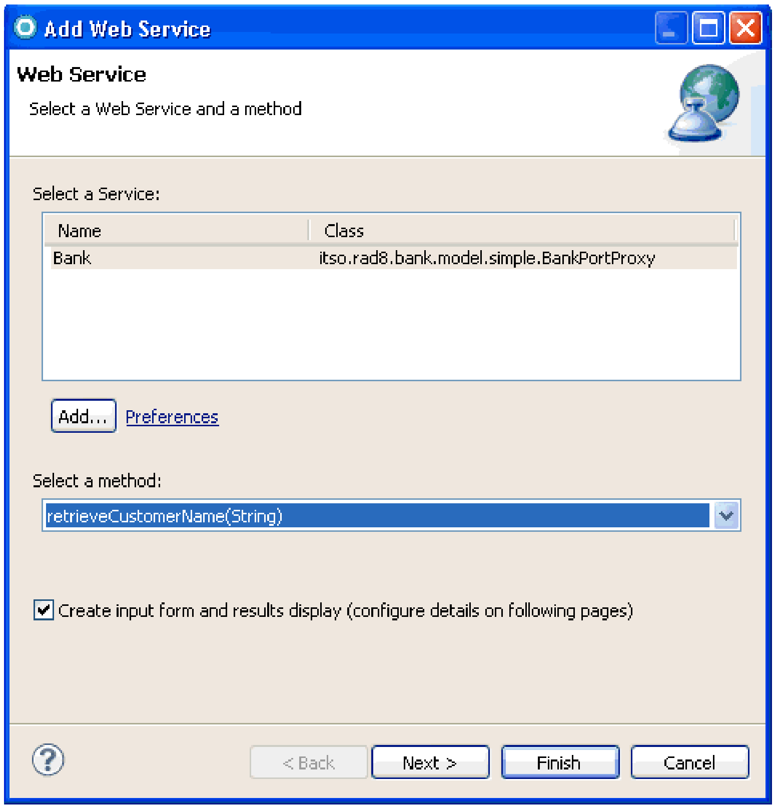
Select a port to generate a proxy.

Figure 26 Web Services Discovery Dialog: Clicking the Add to Project button

The web service that you selected is now listed in the list of web services.

12. In the Web Service window (Figure 27), perform these steps:

a. For Service Name, select **Bank**.
b. For the method, select **retrieveCustomerName(String)**.
c. Select **Create input form and results display**.
d. Click **Next**.



Select a Service:

Name	Class
Bank	itso.rad8.bank.model.simple.BankPortProxy

Select a method: retrieveCustomerName(String)

[✓] Create input form and results display (configure details on following pages)

Figure 27   Selecting a web service and method

13. In the Input Form window (Figure 28), perform these steps:

   a. Change the label to `Enter Social Security Number:`.
   b. Click **Options** and change the label from Submit to `Get Full Name`.
   c. Click **OK** and click **Next**.

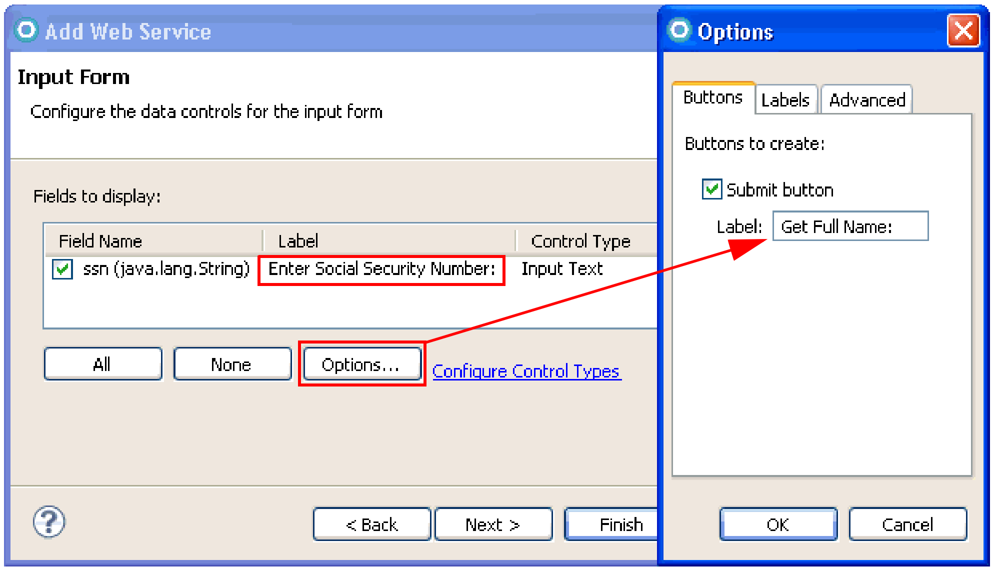


Figure 28 Web service input form

14. In the Results form window (Figure 29), change the Label to `Customer's full name is:`.

15. Click **Finish** to generate the input and output parts into the JSF page (Figure 29). Save the file.

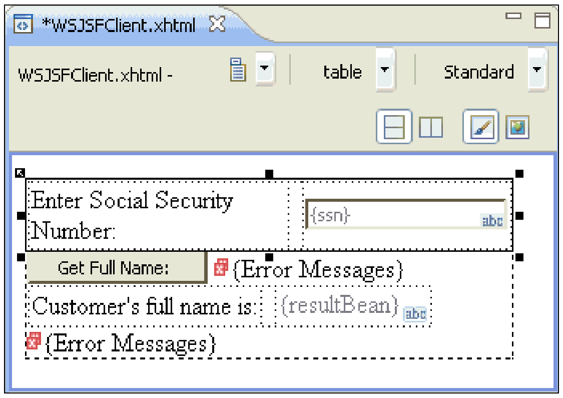
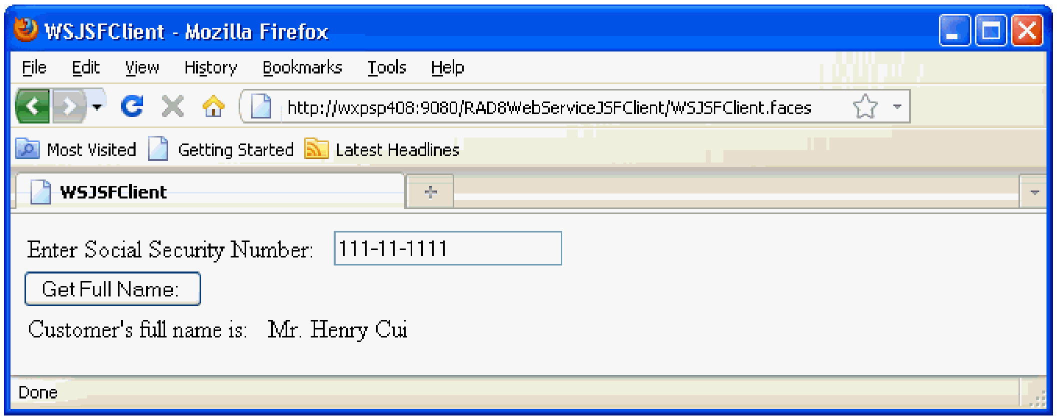


Figure 29 JSF page with the web service invocation

16. Right-click **WSJSFClient.jsp** and select **Run As** → **Run on Server**. The client application is deployed to the server for testing. Perform this test:

   a. In the Enter Social Security Number field, type `111-11-1111`.
   b. Click **Get Full Name**.

   The result is displayed (Figure 30).


WSJSFClient - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://wxpsp408:9080/RAD8WebServiceJSFClient/WSJSFClient.faces
Most Visited Getting Started Latest Headlines
WSJSFClient
Get Full Name:
Done

Enter Social Security Number: 111-11-1111

Customer's full name is: Mr. Henry Cui

Figure 30 JSF client run

The projects that have been developed up to this point are available in this folder:

`4884codesolution\webservices\RAD8WebServiceJSFClient.zip`

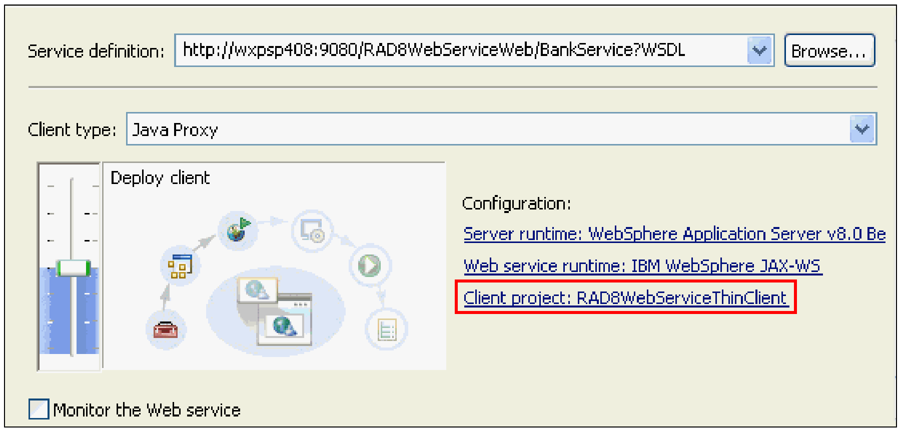
Creating a web service thin client

WebSphere Application Server provides an unmanaged client implementation that is based on the JAX-WS 2.2 specification. The *thin client* for JAX-WS with WebSphere Application Server is an unmanaged and stand-alone Java client environment. The thin client enables running JAX-WS client applications to invoke web services that are hosted by WebSphere Application Server. A web service thin client relies only on a Java developer kit that is compatible with IBM WebSphere Application Server V8 and a thin client JAR file that is available in the `<WAS_HOME>\runtimes\com.ibm.jaxws.thinclient_8.0.0.jar` file where typically `<WAS_HOME>=C:\Program Files\IBM\WebSphere\AppServer`.

Creating the thin client project and generating the client code

To create the web service thin client, follow these steps:

1. Create a Java project by selecting **File** → **New** → **Project** → **Java Project**.
2. For the Project name, enter `RAD8WebServiceThinClient` and click **Finish**.
3. In the Java EE perspective: Services view, expand **JAX-WS**, right-click **RAD8WebServiceWeb: {http://.../}BankService**, and select **Generate** → **Client**.
4. Complete the following actions:
   a. Keep the slider at the **Deploy client** level. Click the hyperlink **Client project**.
   b. In the Specify Client Project Settings window (Figure 31), for the Client project, select **RAD8WebServiceThinClient** and click **OK**.


Service definition: http://wxpsp408:9080/RAD8WebServiceWeb/BankService?WSDL

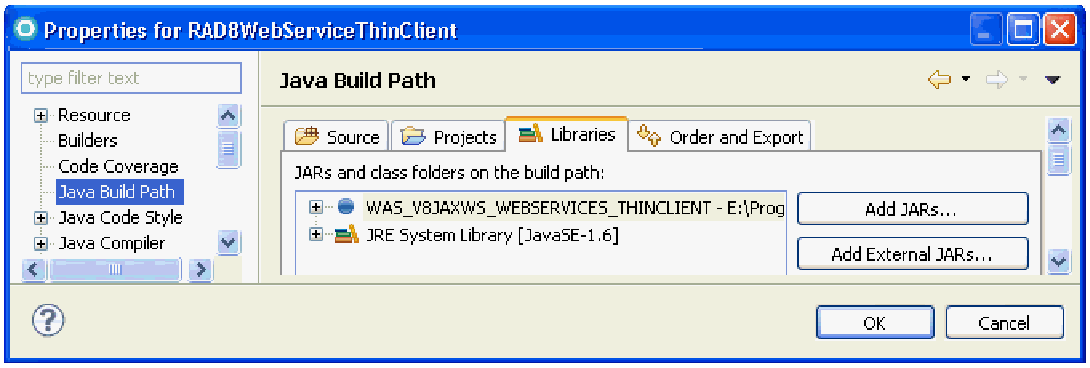
Client type: Java Proxy

Configuration:
Server runtime: WebSphere Application Server v8.0 Be
Web service runtime: IBM WebSphere JAX-WS
Client project: RAD8WebServiceThinClient

Figure 31 Generating a thin client

   c. Click **Finish** to generate the helper classes and WSDL file into the client project.

5. After the code generation, switch to the **Enterprise Explorer** view. Right-click **RAD8WebServiceThinClient** and select **Properties**. Select **Java Build Path**. Click the **Libraries** tab (Figure 32).


Properties for RAD8WebServiceThinClient

Java Build Path

JARs and class folders on the build path:
WAS_V8JAXWS_WEBSERVICES_THINCLIENT - E:\Prog
JRE System Library [JavaSE-1.6]

Figure 32 Web service thin client build path

Notice that the thin client only requires the Java Runtime Environment (JRE) and a thin client JAR file. The wizard adds a class path variable `WAS_V8JAXWS_WEBSERVICES_THINCLIENT`, which points to the `com.ibm.jaxws.thinclient_8.0.0.jar` file.

Creating the client class to invoke the web service

To invoke the web service, create a Java class:

1. Right-click **RAD8WebServiceThinClient** and select **New** → **Class**.
2. For the Package name, type `itso.rad8.bank.test`, and for the Class name, type `WSThinClientTest`. Select **public static void main(String[] args)** and click **Finish**.
3. Copy and paste the code from `WSThinClientTest.java` in the `4884code\webservices\thinclient` directory (Example 15).

Example 15 WSThinClientTest

```
package itso.rad8.bank.test;

import itso.rad8.bank.model.simple.BankPortProxy;
import itso.rad8.bank.model.simple
                      .CustomerDoesNotExistException_Exception;
import java.util.Scanner;

public class WSThinClientTest {

   public static void main(String[] args) {
      try {
         Scanner scanner = new Scanner(System.in);
         BankPortProxy proxy = new BankPortProxy();
         System.out.println
            ("Please enter customer's social security number: ");
         String ssn = scanner.next();
         System.out.println("Customer's name is " +
                          proxy.retrieveCustomerName(ssn));
      } catch (CustomerDoesNotExistException_Exception e) {
         System.out.println("The customer does not exist!");
      }
   }
}
```

Notice how easy it is to invoke the web service. You instantiate the proxy class (`BankPortProxy`) and call the method (`retrieveCustomerName`) in the proxy.

4. Right-click **WSThinClientTest.java** and select **Run As** → **Java Application**.
5. When prompted in the console, for the customer's Social Security number, type `111-11-1111`, and the customer's name is displayed:

```
Retrieving document at
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Retrieving schema at 'BankService_schema1.xsd', relative to
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Please enter customer's social security number:
111-11-1111
Customer's name is Mr. Henry Cui
```

Creating asynchronous web service clients

An asynchronous invocation of a web service sends a request to the service endpoint and then immediately returns control to the client program without waiting for the response to return from the service. JAX-WS asynchronous web service clients consume web services using either the polling approach or the callback approach:

- Using a *polling model*, a client can issue a request and receive a response object that is polled to determine if the server has responded. When the server responds, the actual response is retrieved.
- Using the *callback model*, the client provides a callback handler to accept and process the inbound response object. The `handleResponse` method of the handler is called when the result is available.

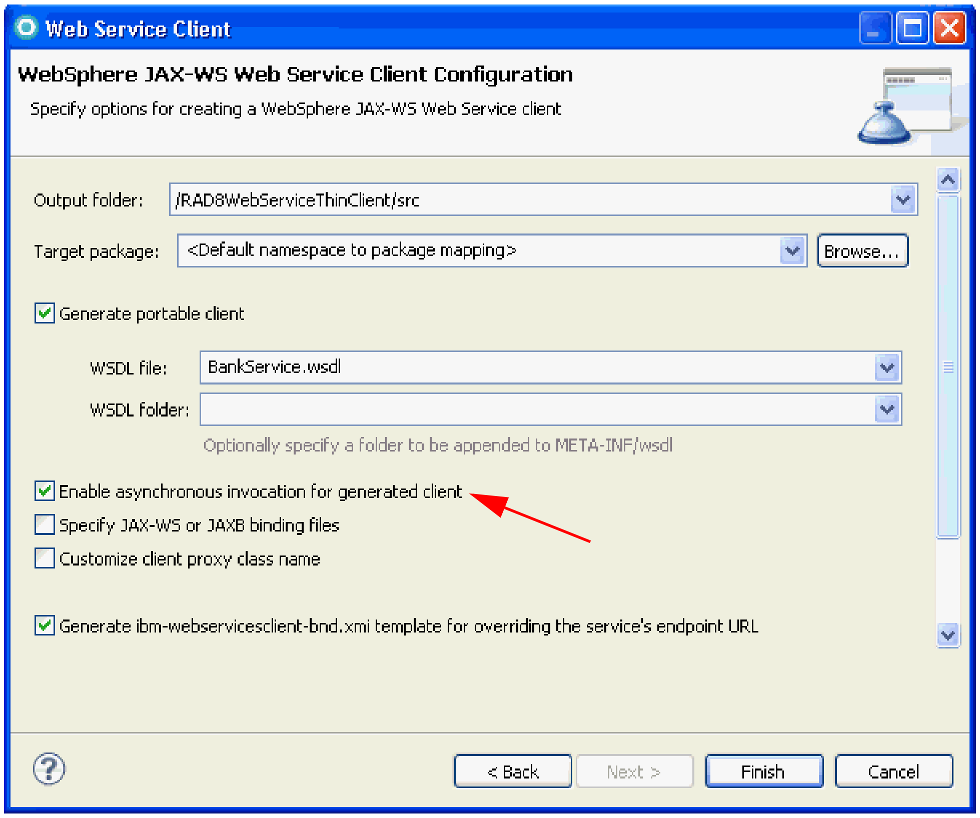
Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke web services.

Polling client

Using the polling model, a client can issue a request and receive a response object that can subsequently be polled to determine if the server has responded. When the server responds, the actual response can then be retrieved. The response object returns the response content when the `get` method is called. The client receives an object of type `javax.xml.ws.Response` from the `invokeAsync` method. That `Response` object is used to monitor the status of the request to the server, determine when the operation has completed, and to retrieve the response results.

To create an asynchronous web service client using the polling model, follow these steps:

1. In the Java EE perspective: Services view, expand **JAX-WS**, right-click **RAD8WebServiceWeb: {http://.../}BankService**, and select **Generate** → **Client**.
2. Keep the slider at the **Deploy client** level. Click the hyperlink **Client project**. In the Specify Client Project Settings window, select **RAD8WebServiceThinClient** and click **OK**. Click **Next**.
3. In the WebSphere JAX-WS Web Service Client Configuration window (Figure 33), select **Enable asynchronous invocation for generated client** and click **Finish**.



WebSphere JAX-WS Web Service Client Configuration

Specify options for creating a WebSphere JAX-WS Web Service client

Output folder: /RAD8WebServiceThinClient/src

Target package: <Default namespace to package mapping>

[✓] Generate portable client

WSDL file: BankService.wsdl

WSDL folder:

Optionally specify a folder to be appended to META-INF/wsdl

[✓] Enable asynchronous invocation for generated client

[ ] Specify JAX-WS or JAXB binding files

[ ] Customize client proxy class name

[✓] Generate ibm-webservicesclient-bnd.xmi template for overriding the service's endpoint URL

Figure 33 Selecting Enable asynchronous invocation for generated client

4. After the code generation, open **BankPortProxy.java** (Example 16). For each method in the web service, two additional methods are created, the *polling* and *callback* methods, which allow the client to function asynchronously. The `retrieveCustomerNameAsync` method that returns a `Response` is used for polling. The method that returns `Future` is used for callback.

Example 16 BankPortProxy asynchronous methods

```
public Response<RetrieveCustomerNameResponse>
                 retrieveCustomerNameAsync(String ssn) {
    return _getDescriptor().getProxy().retrieveCustomerNameAsync(ssn);
}

public Future<?> retrieveCustomerNameAsync(String ssn,
              AsyncHandler<RetrieveCustomerNameResponse> asyncHandler) {
    return _getDescriptor().getProxy().retrieveCustomerNameAsync
                                                (ssn,asyncHandler);
}
```

5. Create a new class called `BankPollingClient` in the `itso.rad8.bank.test` package. Copy and paste the code from `4884code\webservices\thinclient` (Example 17).

Example 17 BankPollingClient

```
package itso.rad8.bank.test;

import itso.rad8.bank.model.simple.BankPortProxy;
import itso.rad8.bank.model.simple.RetrieveCustomerNameResponse;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.Response;

public class BankPollingClient {

   public static void main(String[] args) {
      try {
         BankPortProxy proxy = new BankPortProxy();
         Response<RetrieveCustomerNameResponse> resp =
                    proxy.retrieveCustomerNameAsync("111-11-1111");
         // Poll for the response.
         while (!resp.isDone()) {
            // You can do some work that does not depend on the customer
               name being available
            // For this example, we just check if the result is available
               every 0.2 seconds.
            System.out.println
                  ("retrieveCustomerName async still not complete.");
            Thread.sleep(200);
         }
         RetrieveCustomerNameResponse rcnr = resp.get();
         System.out.println
                  ("retrieveCustomerName async invocation complete.");
         System.out.println("Customer's name is " +
                           rcnr.getCustomerFullName());
      } catch (InterruptedException e) {
         System.out.println(e.getCause());
      } catch (ExecutionException e) {
         System.out.println(e.getCause());
      }
   }
}
```

6. Right-click **BankPollingClient.java** and select **Run As** → **Java Application**. The output is written to the console:

```
Retrieving document at
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Retrieving schema at 'BankService_schema1.xsd', relative to
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
retrieveCustomerName async still not complete.
retrieveCustomerName async still not complete.
retrieveCustomerName async still not complete.
retrieveCustomerName async still not complete.
retrieveCustomerName async still not complete.
retrieveCustomerName async invocation complete.
Customer's name is Mr. Henry Cui
```

From the results, you can see that the asynchronous call allows you to perform other work while waiting for the response from the server. Eventually, you can obtain the results of the invocation.

Callback client

To implement an asynchronous invocation that uses the callback model, the client provides an `AsynchHandler` callback handler to accept and process the inbound response object. The client callback handler implements the `javax.xml.ws.AsynchHandler` interface, which contains the application code that is run when an asynchronous response is received from the server.

The `AsynchHandler` interface contains the `handleResponse(Response)` method that is called after the run time has received and processed the asynchronous response from the server. The response is delivered to the callback handler in the form of a `javax.xml.ws.Response` object. The response object returns the response content when the `get` method is called.

Additionally, if an error was received, an exception is returned to the client during that call. The response method is then invoked according to the threading model used by the executor method, `java.util.concurrent.Executor`, on the client's `java.xml.ws.Service` instance that was used to create the dynamic proxy or dispatch client instance. The executor is used to invoke any asynchronous callbacks registered by the application. Use the `setExecutor` and `getExecutor` methods to modify and retrieve the executor configured for the service.

To create an asynchronous web service client using the callback model, follow these steps:

1. Create the callback handler class `RetrieveCustomerCallbackHandler` in the `itso.rad8.bank.test` package. Copy and paste the code from `4884code\webservices\thinclient` (Example 18).

Example 18 RetrieveCustomerCallbackHandler

```
package itso.rad8.bank.test;

import itso.rad8.bank.model.simple.RetrieveCustomerNameResponse;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

public class RetrieveCustomerCallbackHandler implements
                        AsyncHandler<RetrieveCustomerNameResponse> {
   private String customerFullName;
   public void handleResponse(Response<RetrieveCustomerNameResponse> resp){
      try {
         RetrieveCustomerNameResponse rcnr = resp.get();
         customerFullName = rcnr.getCustomerFullName();
      } catch (ExecutionException e) {
         System.out.println(e.getCause());
      } catch (InterruptedException e) {
         System.out.println(e.getCause());
      }
   }
   public String getResponse() {
      return customerFullName;
   }
}
```

2. Create the `BankCallbackClient` callback client class in the `itso.rad8.bank.test` package. Copy and paste the code from `4884code\webservices\thinclient` (Example 19).

Example 19 BankCallbackClient

```
package itso.rad8.bank.test;

import itso.rad8.bank.model.simple.BankPortProxy;
import java.util.concurrent.Future;

public class BankCallbackClient {

   public static void main(String[] args) throws Exception {
      BankPortProxy proxy = new BankPortProxy();
      // Set up the callback handler.
      RetrieveCustomerCallbackHandler callbackHandler =
                           new RetrieveCustomerCallbackHandler();
      // Make the Web service call.
      Future<?> response = proxy.retrieveCustomerNameAsync
                              ("111-11-1111", callbackHandler);
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("Customer's full name is "
                           + callbackHandler.getResponse() + ".");
      System.out.println("RetrieveCustomerName async end.");
   }
}
```

3. Right-click **BankCallbackClient.java** and select **Run As** → **Java Application**. The output is written to the console:

```
Retrieving document at
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Retrieving schema at 'BankService_schema1.xsd', relative to
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Wait 5 seconds.
Customer's full name is Mr. Henry Cui.
RetrieveCustomerName async end.
```

Asynchronous message exchange client

By default, asynchronous client invocations do not have asynchronous behavior of the message exchange pattern on the wire. The programming model is asynchronous; however, the exchange of request or response messages with the server is not asynchronous. IBM has provided a feature that goes beyond the JAX-WS specification to provide the asynchronous message exchange support.

In the asynchronous message exchange case, the client listens on a separate HTTP channel to receive the response messages from a service-initiated HTTP channel. The client uses WS-Addressing to provide the ReplyTo endpoint reference (EPR) value to the service. The service initiates a connection to the ReplyTo EPR to send a response. To use an asynchronous message exchange, the `com.ibm.websphere.webservices.use.async.mep` property must be set on the client request context with a boolean value of `true`. When this property is enabled, the messages exchanged between the client and server differ from messages exchanged synchronously.

To create an asynchronous message exchange client, follow these steps:

1. Create the `BankCallbackMEPClient` class in the `itso.rad8.bank.test` package. Copy and paste the code from `4884code\webservices\thinclient` (Example 20).

Example 20 BankCallbackMEPClient

```
package itso.rad8.bank.test;

import itso.rad8.bank.model.simple.BankPortProxy;
import java.util.concurrent.Future;
import javax.xml.ws.BindingProvider;

public class BankCallbackMEPClient {

   public static void main(String[] args) throws Exception {
      BankPortProxy proxy = new BankPortProxy();
      //proxy._getDescriptor().setEndpoint
           ("http://localhost:11487/RAD75WebServiceWeb/BankService");
      // setup the property for asynchronous message exchange
      BindingProvider bp = (BindingProvider)
                          proxy._getDescriptor().getProxy();
      bp.getRequestContext().put
         ("com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
      // Set up the callback handler.
      RetrieveCustomerCallbackHandler callbackHandler =
                       new RetrieveCustomerCallbackHandler();
      // Make the Web service call.
      Future<?> response = proxy.retrieveCustomerNameAsync
                             ("111-11-1111", callbackHandler);
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("Customer's full name is "
                       + callbackHandler.getResponse() + ".");
      System.out.println("RetrieveCustomerName async end.");
      }
}
```

2. Right-click **BankCallbackMEPClient.java** and select **Run As** → **Java Application**. The output is written to the console:

```
Retrieving document at
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
Retrieving schema at 'BankService_schema1.xsd', relative to
'file:/C:/workspaces/WebServices/RAD8WebServiceThinClient/bin/META-INF/wsdl/'.
[WAShttpAsyncResponseListener] listening on port 4553
Wait 5 seconds.
Customer's full name is Mr. Henry Cui.
RetrieveCustomerName async end.
```

   Notice the new line in the WebSphere Application Server Console:

```
[10/22/10 14:43:56:359 PDT] 00000024 WSChannelFram A   CHFW0019I: The Transport
Channel Service has started chain
HttpOutboundChain:wxpsp408.rcsnl.ams.nl.ibm.com:4553.
```

3. Optional: If you want to see the SOAP request message, activate the comment line:

```
proxy._getDescriptor().setEndpoint
          ("http://wxpsp408:12036/RAD75WebServiceWeb/BankService");
```

   You must supply your host name instead of `wxpsp408` and the port `12036` must match the port of the TCP/IP Monitor.

4. Run the application again. Example 21 shows the SOAP request.

Example 21 SOAP request for asynchronous message exchange

```
--MIMEBoundary_3028c58b531c6cb4c683e77e89daa042d5c97cdfa680727e
Content-Type: application/xop+xml; charset=UTF-8; type="text/xml"
Content-Transfer-Encoding: binary
Content-ID: <0.2028c58b531c6cb4c683e77e89daa042d5c97cdfa680727e@apache.org>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
<wsa:To>http://wxpsp408:12036/RAD8WebServiceWeb/BankService</wsa:To><wsa:ReplyT
o>
<wsa:Address>http://wxpsp408.rcsnl.ams.nl.ibm.com:4991/axis2/services/BankServi
ce.BankPort</wsa:Address>
</wsa:ReplyTo>
<wsa:MessageID>urn:uuid:d0da36f9-9623-4d6e-8249-d57a118c43e2</wsa:MessageID>
<wsa:Action>urn:getCustomerFullName</wsa:Action>
</soapenv:Header>
<soapenv:Body>
<a:RetrieveCustomerName xmlns:a="http://simple.model.bank.rad8.itso/">
<ssn>111-11-1111</ssn>
</a:RetrieveCustomerName>
</soapenv:Body>
</soapenv:Envelope>
--MIMEBoundary_3028c58b531c6cb4c683e77e89daa042d5c97cdfa680727e--
```

Because the client listens on a separate HTTP channel to receive the response messages from a service-initiated HTTP channel, the TCP/IP Monitor is unable to capture the SOAP response.

The completed Thin Client project is available in this file:

```
4884codesolution\webservices\RAD8WebServiceThinClient.zip
```

Creating web services from an EJB

You can generate EJB web services by using either the Web Service wizard or annotations.

In this section, you create a JAX-WS web service from an EJB session bean using annotations:

1. Expand the EJB project **RAD8WebServiceEJB** and open the **SimpleBankFacadeBean** (in `ejbModule/itso.rad8.bank.ejb.facade`).
2. Add the `@WebService` annotation on the line over the `@Stateless` annotation (Example 22). Press Ctrl+Shift+O to resolve the import.

   Example 22   Annotating a stateless session EJB

```
@WebService
@Stateless
public class SimpleBankFacadeBean implements SimpleBankFacadeBeanLocal {
   ......
```

3. Wait for the `RAD8WebServiceEAR` application to publish on the server (or force a manual publish). Notice that a new web service named `RAD8WebServiceEJB` is added in the Services view under JAX-WS.

   An HTTP router module is required to allow the transport of SOAP messages over the HTTP protocol.
4. In the Services view (Figure 34), right-click the new **RAD8WebServiceEJB** and select **Create Router Modules (EndpointEnabler)**.

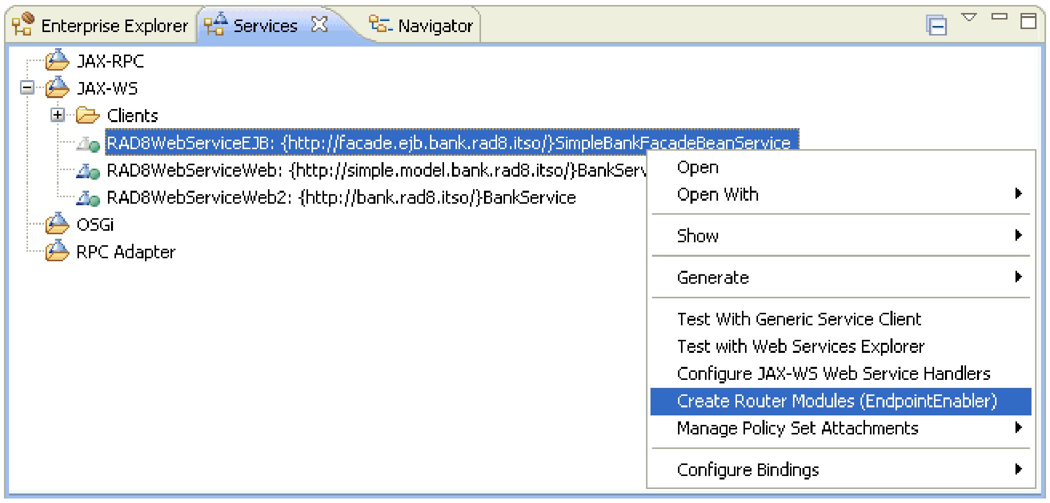
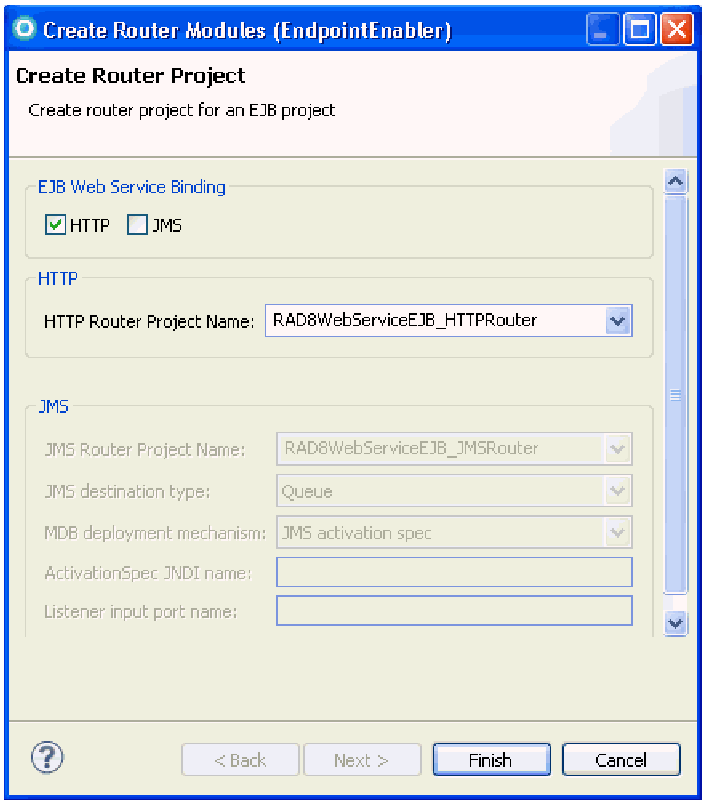

Enterprise Explorer
Services
Navigator
JAX-RPC
JAX-WS
Clients
RAD8WebServiceEJB: {http://facade.ejb.bank.rad8.itso/}SimpleBankFacadeBeanService
RAD8WebServiceWeb: {http://simple.model.bank.rad8.itso/}BankServ
RAD8WebServiceWeb2: {http://bank.rad8.itso/}BankService
OSGi
RPC Adapter
Open
Open With
Show
Generate
Test With Generic Service Client
Test with Web Services Explorer
Configure JAX-WS Web Service Handlers
Create Router Modules (EndpointEnabler)
Manage Policy Set Attachments
Configure Bindings

Figure 34   Selecting Create Router Modules

5. In the Create Router Project window, accept **HTTP** as the default EJB web service binding (Figure 35). Although two EJB bindings are listed, HTTP and JMS, for this example, we use SOAP over HTTP. Click **Finish**.


Create Router Modules (EndpointEnabler)
Create Router Project
Create router project for an EJB project
EJB Web Service Binding
HTTP
JMS
< Back
Next >
Finish
Cancel

HTTP Router Project Name: RAD8WebServiceEJB_HTTPRouter

JMS Router Project Name: RAD8WebServiceEJB_JMSRouter

JMS destination type: Queue

MDB deployment mechanism: JMS activation spec

ActivationSpec JNDI name:

Listener input port name:

Figure 35 Create Router Project window

6. Open the deployment descriptor of the `RAD8WebServiceEJB_HTTPRouter` project to see the generated servlet.
7. In the Services view, right-click **RAD8WebServiceEJB** and select **Test with Generic Service Client**.
8. Select the **getAccountBalance** operation and select the **SimpleBankFacadeBeanPort** under it.
9. In the **Edit Data** → **Message** → **Form** tab, expand **getAccountNumber**.
10. Select **arg0** (which represents the Account Number input parameter).
11. For the Account number, type `001-999000777`.
12. Click **Invoke**. You can see the result of the web service call:

```
getAccountBalanceResponse
12345.67
```

All of the projects that we have completed so far are available in this file:

```
4884codesolution\webservices\RAD8WebServiceEJBService.zip
```

Creating a top-down web service from a WSDL

When creating a web service using a top-down approach, first you design the implementation of the web service by creating a WSDL file. You can do this by using the WSDL editor. You can then use the Web Service wizard to create the web service and skeleton Java classes to which you can add the required code. The top-down approach is the recommended way of creating a web service.

Designing the WSDL by using the WSDL editor

In this section, you create a WSDL with two operations: `getAccount` (using an account ID to retrieve an account) and `getCustomer` (using a customer ID to retrieve a customer).

By using the WSDL editor, you can easily and graphically create, modify, view, and validate WSDL files. To create a WSDL, follow these steps:

1. Create a dynamic web project to host the new web service:
   - Web project: `RAD8TopDownBankWS`
   - EAR project: `RAD8TopDownBankEAR`
2. Create a WSDL file:
   a. Right-click **WebContent** (in `RAD8TopDownBankWS`).

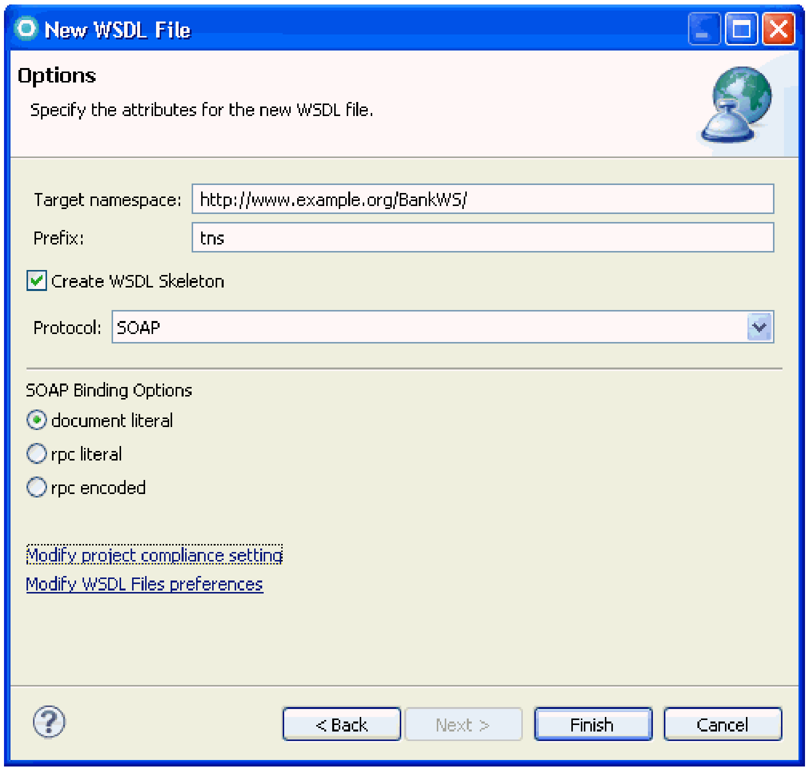
   b. Select **New** → **Other**.

   c. In the New Wizard, select **Web services** → **WSDL**.

   d. Click **Next**.

   e. Change the File name to `BankWS.wsdl` and click **Next**.

   f. In the Options window (Figure 36 on page 56), ensure that the default options **Create WSDL Skeleton** and **document literal** are selected. Then click **Finish**.


New WSDL File

Options

Specify the attributes for the new WSDL file.

Target namespace: http://www.example.org/BankWS/

Prefix: tns

[✓] Create WSDL Skeleton

Protocol: SOAP

SOAP Binding Options

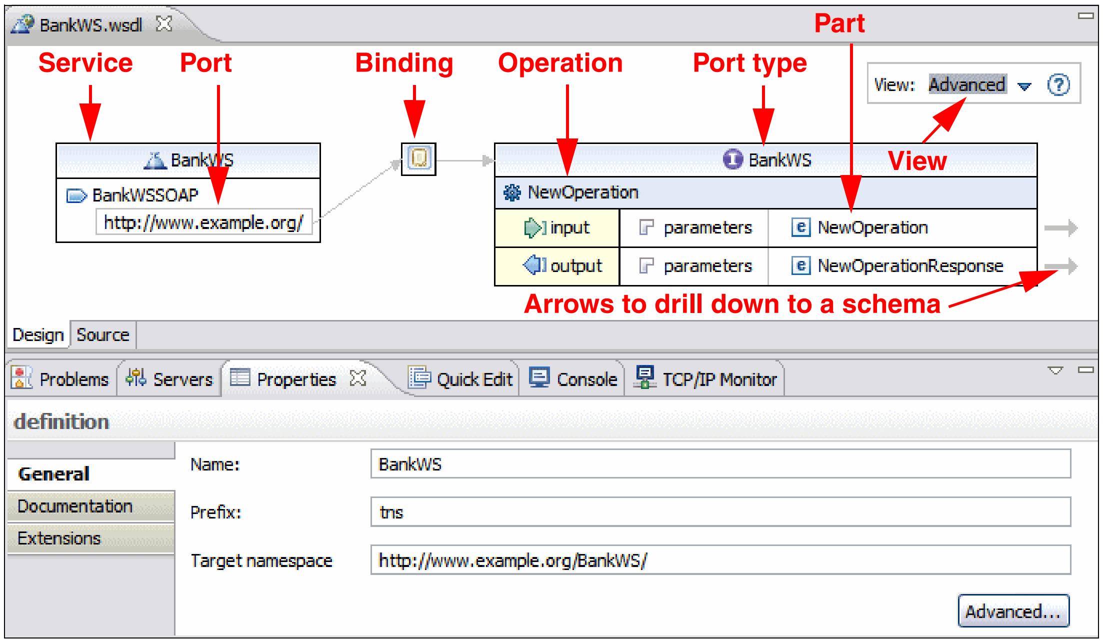
[● document literal] ○ rpc literal ○ rpc encoded

Modify project compliance setting

Modify WSDL Files preferences

Figure 36   New WSDL File wizard: Options window

3. When the WSDL editor opens with the new WSDL file, select the **Design** tab:
   a. In the WSDL editor, for View (in the upper-right corner), select **Advanced**.
   b. Select the **Properties** view. Now you are ready to edit the WSDL file (Figure 37).



Name: BankWS

Prefix: tns

Target namespace: http://www.example.org/BankWS/

Figure 37   WSDL editor

Editing the WSDL file

Define the operations and parameters of the WSDL:

1. Change the operation name by double-clicking **NewOperation** and overtyping the name with `getAccount`. (You can also select the operation and change the name in the Properties view.)

   As a result, the input parameter changes to `getAccount` and the output parameter changes to `getAccountResponse`.

2. Add a new operation. In the Design view, right-click the port type **BankWS**, select **Add Operation**, and name the operation `getCustomer`.

   As a result, the input parameter is set to `getCustomer` and the output parameter changes to `getCustomerResponse`.

3. To change the input type of the WSDL operation `getAccount`, click the **arrow** to the right of the input parameter to drill down to the schema.

4. When the Inline Schema editor opens, select the **Design** tab and switch to the **Detailed** view (Figure 38).

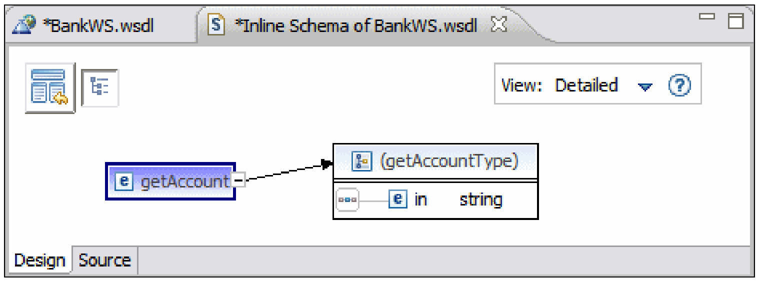


Figure 38 Schema editor: Start

   a. Select the **in** element, and in the Properties view, change the name to `accountId` (leave the type as `xsd:string`).

   b. Click the **Show schema index view** icon ( ) in the upper-left corner to see all the directives, elements, types, attributes, and groups in the WSDL (Figure 39).

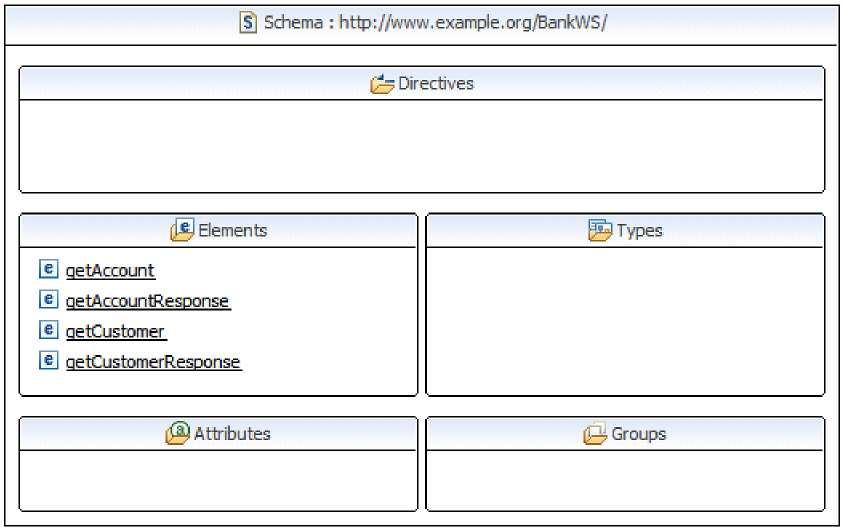


Figure 39 Schema editor: Index view

c. In the Types category, right-click and select **Add Complex Type**. Change the name to `Account`. Follow these steps:

   i. Right-click **Account** and select **Add Sequence**.

   ii. You are taken out of the Index view and back into the Online Schema.

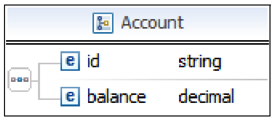
   iii. Right-click **Account** and select **Add Element**.

   iv. Change the Element name to `id`.

   v. Right-click the content model object (▫▫▫) and select **Add Element**.

   vi. Change the name to `balance`.

   vii. Click **Browse** and select the type as **decimal** (Figure 40).



Account

id	string
balance	decimal

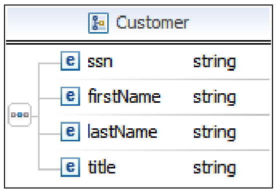
Figure 40 Schema editor: Account

d. Click the **Show schema index view** icon ( ) in the upper-left corner. Perform these steps:

   i. In the Types section, right-click and select **Add Complex Type**.

   ii. Change the name to `Customer`.

   iii. Right-click **Customer** and select **Add Sequence**. Add four elements: **ssn**, **firstName**, **lastName**, and **title**, all of type `string` (Figure 41).



Customer

ssn	string
firstName	string
lastName	string
title	string

Figure 41 Schema editor: Customer

e. Click the **Show schema index view** icon ( ) in the upper-left corner. Add the global elements for the complex types:

   i. Right-click the **Elements** category and click **Add Element**.

   ii. Change the element name to `Account`.

   iii. Right-click **Account** and select **Set Type**.

   iv. Click **Browse** and select **Account**.

   v. Add the global element **Customer**.

   vi. Set type to `Customer` (Figure 42 on page 59).

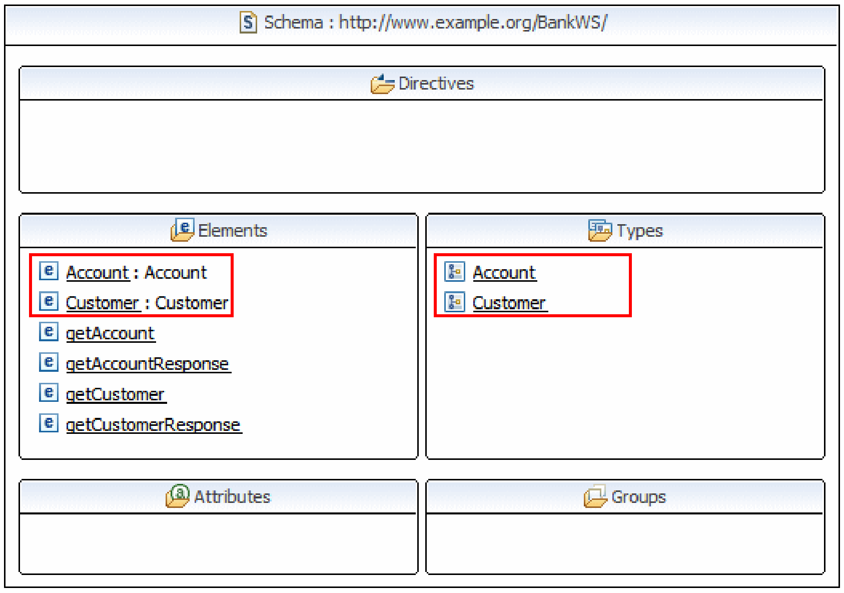


Figure 42   Schema editor: Global elements

5. In the WSDL editor, click the **arrow** to the right of the **getAccountResponse** element to drill down to the schema:
   a. Right-click the **out** element and select **Set Type**.
   b. Click **Browse** and select **Account** (Figure 43).

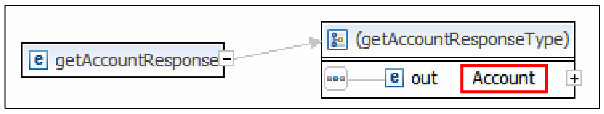
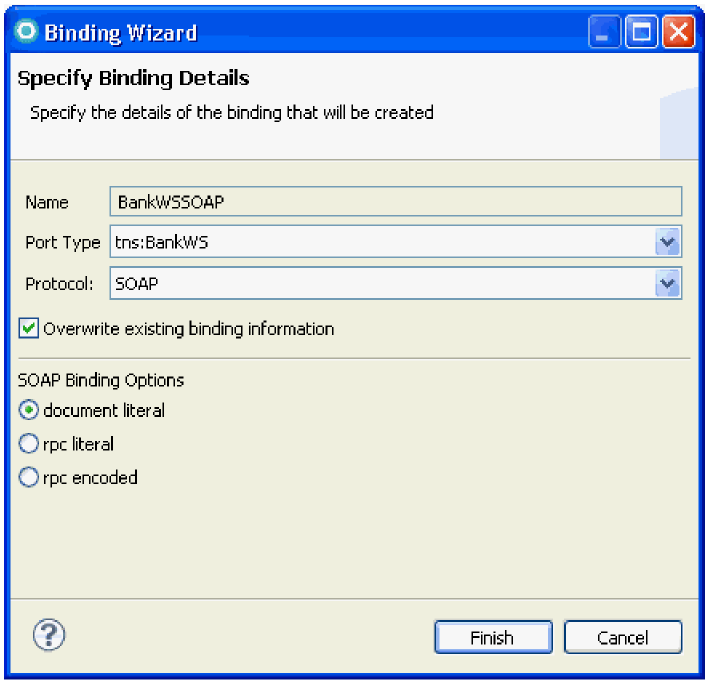


Figure 43   Schema editor: Output message

6. In the WSDL editor, click the **arrow** to the right of the **getCustomer** element to drill down to the schema. Change the element name from `in` to `customerId`.
7. In the WSDL editor, click the **arrow** to the right of the **getCustomerResponse** element to drill down to the schema:
   a. Right-click the **out** element and select **Set Type**.
   b. Click **Browse** and select **Customer**.
8. In the WSDL editor, right-click the **binding** icon (see Figure 37 on page 56). Select **Generate Binding Content**.

9. In the Binding wizard (Figure 44), select **Overwrite existing binding information** and click **Finish**.



Specify Binding Details

Specify the details of the binding that will be created

Name: BankWSSOAP

Port Type: tns:BankWS

Protocol: SOAP

[✓] Overwrite existing binding information

SOAP Binding Options

[✓ document literal]

rpc literal

rpc encoded

Figure 44 Specify Binding Details wizard

10. Save the schema and WSDL file (Figure 45).

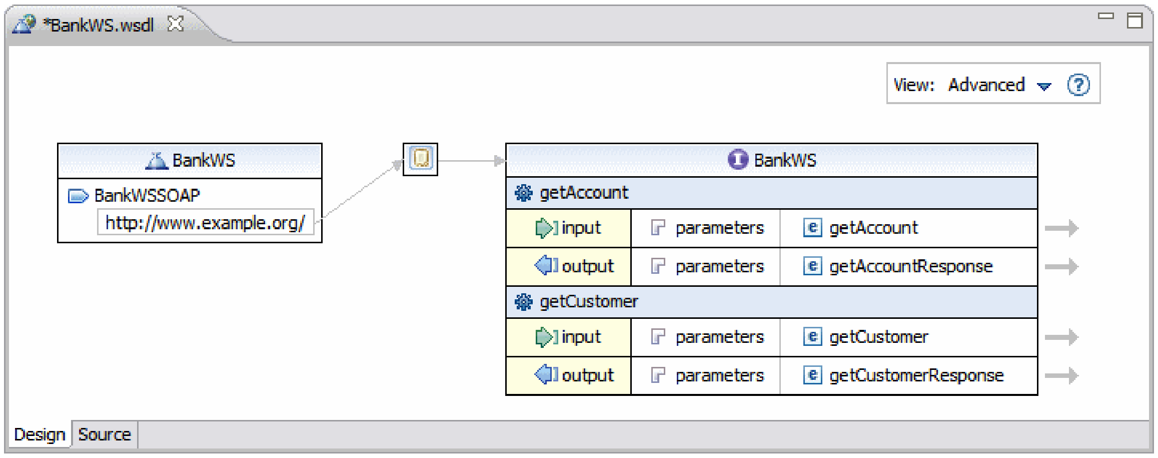


Figure 45 BankWS.wsdl

11. In the Enterprise Explorer, right-click **BankWS.wsdl** and select **Validate**. In the window that opens showing a message that confirms that there are no errors or warnings, click **OK**.

If you have a problem when creating the WSDL file, you can import the `BankWS.wsdl` file from the `4884code\webservices\topdown` directory.

Generating the skeleton JavaBean web service

To generate a skeleton JavaBean web service from a WSDL, follow these steps:

1. Right-click **BankWS.wsdl** and select **Web Services** → **Generate Java bean skeleton**.
2. Keep the slider at the **Start service** level. Notice that the code is generated into the `RAD8TopDownBankWS` project. Click **Finish**.

After the code generation, the skeleton class `BankWSSOAPImpl.java` opens in the Java editor. Notice the annotation of the class:

```
@javax.jws.WebService (endpointInterface="org.example.bankws.BankWS",
targetNamespace="http://www.example.org/BankWS/", serviceName="BankWS",
portName="BankWSSOAP")
```

The `RAD8TopDownBankEAR` is deployed to the server, and the web service is displayed in the Services view.

Implementing the generated JavaBean skeleton

You must provide the business logic for the generated JavaBean skeleton. Use the simple implementation that is shown in Example 23.

Example 23   Implementation of the generated JavaBean skeleton

```
package org.example.bankws;

import java.math.BigDecimal;

@javax.jws.WebService (endpointInterface="org.example.bankws.BankWS",
   targetNamespace="http://www.example.org/BankWS/", serviceName="BankWS",
   portName="BankWSSOAP")
public class BankWSSOAPImpl{

   public Account getAccount(String accountId) {
      Account account = new Account();
      account.setId(accountId);
      account.setBalance(new BigDecimal(1000.00));
      return account;
   }

   public Customer getCustomer(String customerId) {
      Customer customer = new Customer();
      customer.setSsn(customerId);
      customer.setFirstName("Henry");
      customer.setLastName("Cui");
      customer.setTitle("Mr.");
      return customer;
    }
}
```

After you save the `.java` file, wait for the Server to republish or publish manually.

Testing the generated web service

To test the web service, use the Generic Service Client.

BankWS.wsdl file: You cannot use the `BankWS.wsdl` file in the `WebContent` folder to test the web service. The web service endpoint was set to `http://www.example.org/` when we created this WSDL. The dynamic WSDL loading from the Services view sets the endpoint correctly.

Follow these steps:

1. To test the web service, in the Services view, expand **JAX-WS**, right-click **RAD8TopDownBankWS**, and select **Test with Generic Service Client**.
2. Test the **getCustomer** and **getAccount** operations. You see the correct result in the Generic Service Client.

The Top Down Project is available in this file:

`4884codesolution\webservices\RAD8TopDownWebService.zip`

Explore: After you create a web service, you might want to make the following changes to it:

- For example, you might want to add a new WSDL operation `getBalanceByAccountId` to the WSDL. After the WSDL is changed, you have to regenerate the web service code. The existing business logic might be wiped out.
- To retain your changes while updating the web service, you can use the *skeleton merge* feature. With this feature, you can regenerate the web service while keeping your changes intact. Select **Window** → **Preferences** → **Web Services** → **Resource Management** → **Merge generated skeleton file**.

Creating web services with Ant tasks

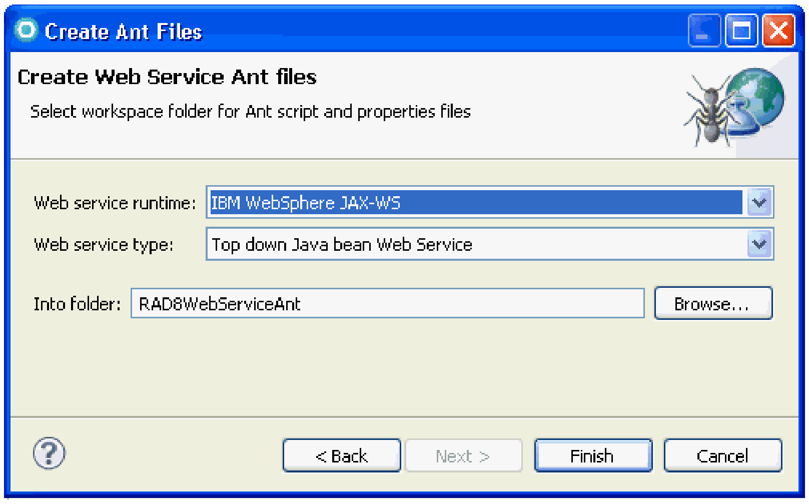
If you prefer not to use the Web Service wizard, you can use Apache Ant tasks to create a web service using the IBM WebSphere JAX-WS runtime environment. The Ant tasks support creating web services using both the top-down and bottom-up approaches. After you have created a web service, you can then deploy it to a server, test it, and publish it as a business entity or business service. Additionally, you can create web service clients using the Ant tasks.

Creation procedure

In this section, we use Ant tasks to automate the top-down code generation process that we did in the last section:

1. Remove `RAD8WebServiceEAR` from the server while we make modifications.
2. Create a dynamic web project to host the web service generated by Ant tasks: `RAD8WebServiceAnt` in `RAD8WebServiceEAR`.
3. Copy the `BankWS.wsdl` file from the `RAD8TopDownBankWS/WebContent` folder to the `RAD8WebServiceAnt` folder (not under WebContent).

4. Right-click **RAD8WebServiceAnt** and select **New** → **Other**. Perform these steps:
   a. In the New wizard, select **Web Services** → **Ant Files**.
   b. Click **Next**.
5. In the Create Ant Files window (Figure 46), perform these steps:
   a. For Web service run time, select **IBM WebSphere JAX-WS**.
   b. For Web service type, select **Top down Java bean Web Service**.
   c. Click **Finish**.


Create Ant Files

Create Web Service Ant files

Select workspace folder for Ant script and properties files

Web service runtime: IBM WebSphere JAX-WS

Web service type: Top down Java bean Web Service

Into folder: RAD8WebServiceAnt

Figure 46 Create Web Service Ant files window

A `wsgenTemplates` folder is created with the `was_jaxws_tdjava.xml` and `was_jaxws_tdjava.properties` files.

6. Open the **was_jaxws_tdjava.properties** file and perform these steps:
   a. Change `InitialSelection=` to `InitialSelection=/RAD8WebServiceAnt/BankWS.wsdl`.
   b. Make sure that the `Service.ServerId` line is equal to `Service.ServerId=com.ibm.ws.ast.st.v8.server.base`.
   c. Save and close the file.

Running the web service Ant task

Perform these steps to run the Ant task:

1. Right-click **was_jaxws_tdjava.xml** and select **Run As** → **Ant Build**.
2. The Configuration Editor opens.
3. In the Edit Configuration window, perform these steps:
   a. Select the **JRE** tab.
   b. Select **Run in the same JRE as the workspace**. This option is required; otherwise, the ANT tasks present in the `was_jaxws_tdjava.xml` file cannot be found at run time.
4. Click **Apply** and then click **Run**.

The web service is generated:

- The web service artifacts are generated in the Java Resources folder.
- You can implement the generated skeleton (`BankWSSOAPImpl`) and test the web service, as we did in the last section.

All projects developed so far are available in this file:

`4884codesolution\webservices\RAD8ANTWebService.zip`

Sending binary data using MTOM

SOAP Message Transmission Optimization Mechanism (MTOM) is a standard that is developed by the World Wide Web Consortium (W3C). MTOM describes a mechanism for optimizing the transmission or wire format of a SOAP message by selectively re-encoding portions of the message while presenting an XML Information Set (Infoset) to the SOAP application.

MTOM uses the XML-binary Optimized Packaging (XOP) in the context of SOAP and Multipurpose Internet Mail Extensions (MIME) over HTTP. XOP defines a serialization mechanism for the XML Infoset with binary content that is not only applicable to SOAP and MIME packaging, but to any XML Infoset and any packaging mechanism. It is an alternate serialization of XML that happens to look like a MIME multipart or related package, with XML documents as the root part.

That root part is similar to the XML serialization of the document, except that base64-encoded data is replaced by a reference to one of the MIME parts, which is not base64 encoded. This reference enables you to avoid the bulk and overhead in processing that are associated with encoding. Encoding is the only way that binary data can work directly with XML.

In this section, we use the top-down approach to create a JAX-WS web service to send binary attachments along with a SOAP request, and to receive binary attachments along with a SOAP response using MTOM.

The web service client sends three types of documents: Microsoft Word, image, and PDF file. We describe several ways to send the documents:

- The client uses `byte[]` to send the Word document.
- The client uses `java.awt.Image` to send the image file.
- The client uses `javax.activation.DataHandler` to send the PDF file.

After the web service receives the binary data from the client, it stores the received document on the local hard disk and then passes the same document back to the client. In an actual scenario, the provider or the consumer can send an acknowledgement message, after it receives the binary data from the other side. For our example, we want to show how to enable the MTOM on both the client and the server side in a compact example.

Creating a web service project and importing the WSDL

To create a web service project, follow these steps:

1. Select **File** → **New** → **Dynamic Web Project**.
2. In the window that opens, complete the following actions:
   a. For Project Name, type `RAD8WSMTOM`.
   b. For EAR Project Name, type `RAD8WSMTOMEAR`.
   c. Click **Finish**.
3. Import the `4884code\webservice\mtom\ProcessDocumentService.wsdl` file into the `RAD8WSMTOM/WebContent` folder.
4. Open the **ProcessDocumentService.wsdl** file. Look at the source, and you see the attributes that are highlighted in Example 24.

Example 24 Extract of the ProcessDocumentService.wsdl file

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
           xmlns:tns="http://mtom.rad8.ibm.com/"
           targetNamespace="http://mtom.rad8.ibm.com/" version="1.0">
<xs:complexType name="sendPDFFile">
   <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="xs:base64Binary"
                                    xmime:expectedContentTypes="*/*"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="sendWordFile">
   <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="xs:base64Binary"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="sendImage">
   <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="xs:base64Binary"
                              xmime:expectedContentTypes="image/jpeg"/>
   </xs:sequence>
</xs:complexType>
```

Default mapping

The default mapping for `xs:base64Binary` is `byte[]` in Java. If you want to use another mapping, you can add the `xmime:expectedContentTypes` attribute to the element containing the binary data. This attribute is defined in the `http://www.w3.org/2005/05/xmlmime` namespace and specifies the MIME types that the element is expected to contain. The setting of this attribute changes how the code generators create the JAXB class for the data. Depending on the `expectedContentTypes` value in the WSDL file, the JAXB artifacts generated are in the Java type, as described in Table 2.

Table 2 Mapping between MIME type and Java type

MIME type	Java type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`

MIME type	Java type
`text/plain`	`java.lang.String`
`text/xml`	`javax.xml.transform.Source`
`application/xml`	`javax.xml.transform.Source`
`*/*`	`javax.activation.DataHandler`

Based on this table, we can make the following predictions:

- `sendWordFile` will be mapped to `byte[]` in Java.
- `sendPDFFile` will be mapped to `javax.activation.DataHandler`.
- `sendImage` will be mapped to `java.awt.Image`.

Generating the web service and client

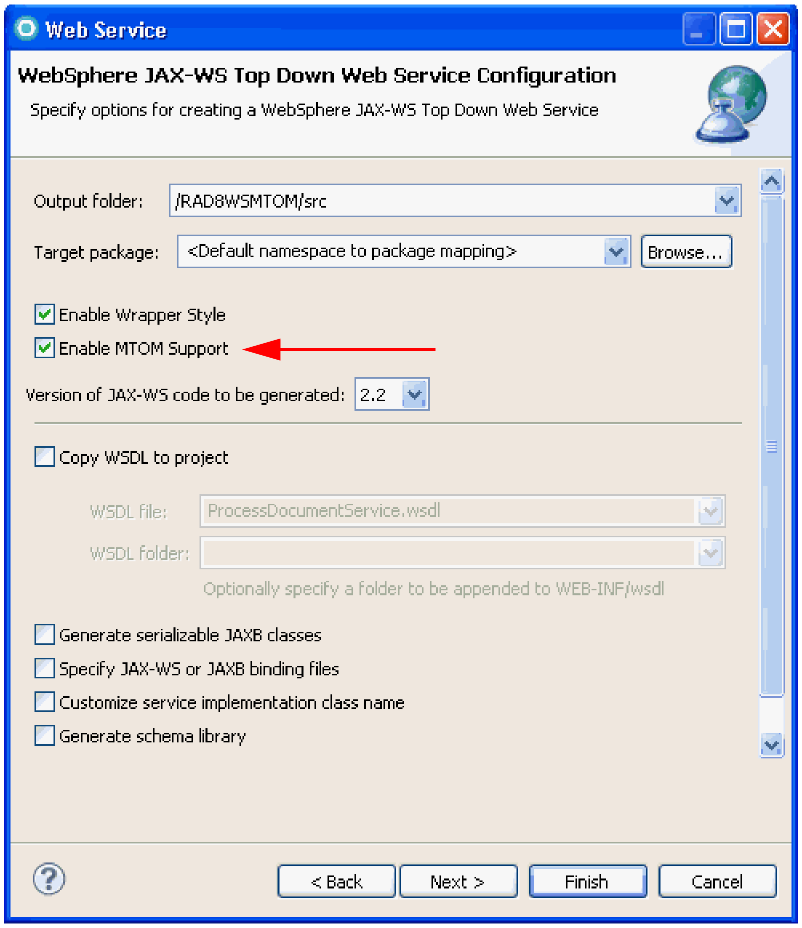
To create the web service and client using the Web Service wizard, follow these steps:

1. Right-click **ProcessDocumentService.wsdl** and select **Web Services** → **Generate Java bean skeleton**.
2. When the Web Service wizard starts with the Web Services page, complete the following steps:
   a. Select the following options for the web service:
      i. For Server, select **WebSphere Application Server v8.0 Beta**.
      ii. For Web service run time, select **IBM WebSphere JAX-WS**.
      iii. For Service project, select **RAD8WSMTOM**.
      iv. For Service EAR project, select **RAD8WSMTOMEAR**.

   b. Select the following options for the web service client:
      i. Move the slider to **Test client**.
      ii. For Server, select **WebSphere Application Server v8.0 Beta**.
      iii. For Web service run time, select **IBM WebSphere JAX-WS**.
      iv. For Client project, select **RAD8WSMTOMClient**.
      v. For Client EAR project, select **RAD8WSMTOMClientEAR**.

      Because the web service client project is not yet in the workspace when you run the Web Service wizard, the wizard creates the project for you.

   c. Select **Monitor the Web service** and then click **Next**.
3. In the WebSphere JAX-WS Top Down Web Service Configuration window (Figure 47 on page 67), perform these steps:
   a. Select **Enable MTOM Support**.
   b. Click **Next**.


Web Service

WebSphere JAX-WS Top Down Web Service Configuration

Specify options for creating a WebSphere JAX-WS Top Down Web Service

Output folder: /RAD8WSMTOM/src

Target package: <Default namespace to package mapping>

[✓] Enable Wrapper Style

[✓] Enable MTOM Support

Version of JAX-WS code to be generated: 2.2

[ ] Copy WSDL to project

WSDL file: ProcessDocumentService.wsdl

WSDL folder:

Optionally specify a folder to be appended to WEB-INF/wsdl

[ ] Generate serializable JAXB classes

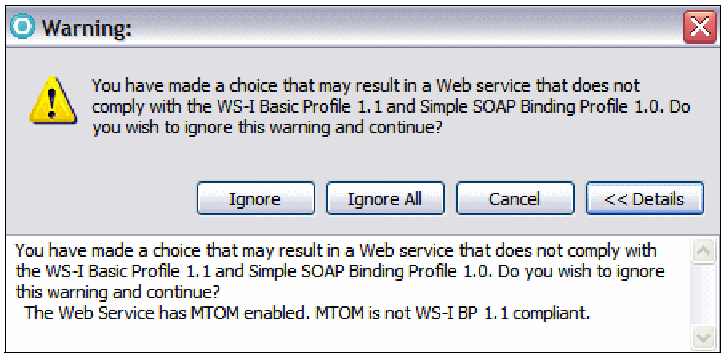
[ ] Specify JAX-WS or JAXB binding files

[ ] Customize service implementation class name

[ ] Generate schema library

Figure 47 Selecting the Enable MTOM Support option

4. In the Warning message window (Figure 48) that opens, click **Details** to view the complete message. Click **Ignore** to continue the code generation.


Warning:

You have made a choice that may result in a Web service that does not comply with the WS-I Basic Profile 1.1 and Simple SOAP Binding Profile 1.0. Do you wish to ignore this warning and continue?

You have made a choice that may result in a Web service that does not comply with the WS-I Basic Profile 1.1 and Simple SOAP Binding Profile 1.0. Do you wish to ignore this warning and continue?
The Web Service has MTOM enabled. MTOM is not WS-I BP 1.1 compliant.

Figure 48 WS-I warning against MTOM

5. In the Test Web Service window, select **Next** (we will not test this way, because it is difficult to supply the required input types, such as an array of bytes).
6. In the WebSphere JAX-WS Web Service Client Configuration window, accept the defaults (**Enable MTOM** is selected) and click **Next**.
7. In the Web Service Client Test window, for the Test Facility, select **JAX-WS JSPs** and click **Finish**. The generated JavaBean skeleton and the sample JSP client open.

Implementing the JavaBean skeleton

Before you test the sample JSP client, you must implement the generated JavaBean skeleton. The web service stores the received document on the local hard drive and then passes the same document back to the client. Follow these steps:

1. Examine the generated skeleton class `ProcessDocumentPortBindingImpl`. We can see that `sendWordFile` is mapped to `byte[]`, `sendPDFFile` is mapped to `javax.activation.DataHandler`, and `sendImage` is mapped to `java.awt.Image`, as we expected.
2. The Wizard has inserted the annotation:

   ```
   @javax.xml.ws.BindingType
         (value=javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
   ```

   You can use the `@BindingType (javax.xml.ws.BindingType)` annotation on an endpoint implementation class to specify that the endpoint supports one of the MTOM binding types so that the response messages are MTOM-enabled. The `javax.xml.ws.SOAPBinding` class defines two constants, `SOAP11HTTP_MTOM_BINDING` and `SOAP12HTTP_MTOM_BINDING`, that you can use for the value of the `@BindingType` annotation.

   To enable MTOM on an endpoint, you can also place the `@MTOM` (`javax.xml.ws.soap.MTOM`) annotation on the endpoint. The `@MTOM` annotation has two parameters, `enabled` and `threshold`. The `enabled` parameter has a boolean value and indicates if MTOM is enabled for the JAX-WS endpoint. The `threshold` parameter has an integer value that must be greater than or equal to zero. When MTOM is enabled, any binary data whose size, in bytes, exceeds the threshold value is XML-binary Optimized Packaging (XOP)-encoded or sent as an attachment. When the message size is less than the threshold value, the message is inlined in the XML document as either base64 or hexBinary data.

   The presence and value of an `@MTOM` annotation overrides the value of the `@BindingType` annotation. For example, if the `@BindingType` indicates that MTOM is enabled, but an `@MTOM` annotation is present with an enabled value of `false`, MTOM is not enabled.
3. Copy the code from `4884code\webservices\mtom` and paste it to `ProcessDocumentPortBindingImpl.java` (Example 25).

Example 25 ProcessDocumentPortBindingImpl.java

```
package com.ibm.rad8.mtom;

import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileOutputStream;
import javax.activation.DataHandler;
import javax.imageio.ImageIO;

@javax.jws.WebService
   (endpointInterface="com.ibm.rad8.mtom.ProcessDocumentDelegate",
   targetNamespace="http://mtom.rad8.ibm.com/",
   serviceName="ProcessDocumentService", portName="ProcessDocumentPort")
@javax.xml.ws.BindingType
      (value=javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class ProcessDocumentPortBindingImpl{

   public byte[] sendWordFile(byte[] arg0) {
```

```
      try {
         FileOutputStream fileOut = new FileOutputStream
            (new File("C:/4884code/webservices/mtomresult/RAD-intro.doc"));
         fileOut.write(arg0);
      } catch (Exception e) {
         e.printStackTrace();
      }
      return arg0;
   }
   public Image sendImage(Image arg0) {
      try {
         File file = new File
            ("C:/4884code/webservices/mntomresult/BlueHills.jpg");
         BufferedImage bi = new BufferedImage(arg0.getWidth(null),
                      arg0.getHeight(null), BufferedImage.TYPE_INT_RGB);
         Graphics2D g2d = bi.createGraphics();
         g2d.drawImage(arg0, 0, 0, null);
         ImageIO.write(bi, "jpeg", file);
      } catch (Exception e) {
         e.printStackTrace();
      }
      return arg0;
   }
   public DataHandler sendPDFFile(DataHandler arg0) {
      try {
         FileOutputStream fileOut = new FileOutputStream(new File(
            "C:/4884code/webservices/mtoresult/JAX-WS.pdf"));
         BufferedInputStream fileIn = new BufferedInputStream
                                   (arg0.getInputStream());
         while (fileIn.available() != 0) {
            fileOut.write(fileIn.read());
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
      return arg0;
   }
}
```

Examine the code in Example 25, and notice the following points:

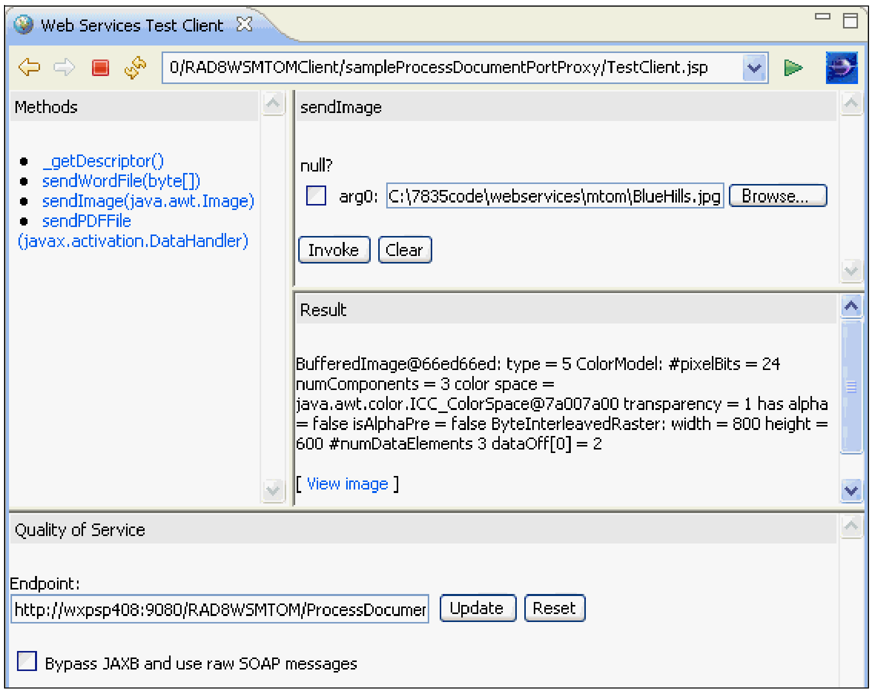
- The `sendWordFile` method takes `byte[]` as input and stores the binary data as `4884code/webservices/mtomresult/RAD-intro.doc`.
- The `sendImage` method takes an image as input and stores the binary data as `4884code/WebServices/mtomresult/BlueHills.jpg`.
- The `sendPDFFile` method takes a `DataHandler` as input and stores the data in `4884code/WebServices/mtomresult/JAX-WS.pdf`.
- All the three methods return the received data to the client after storing it on the local drive.

Testing and monitoring the MTOM-enabled web service

Now it is time to see if MTOM really optimizes the transmission of the data. We use the `4884code\webservices\mtomresult` output folder to store the document received by the web service JavaBean.

1. In the Web Services Test Client view (Figure 49), complete the following actions:
   a. In the sample JSP client, select the **sendImage** method.
   b. Click **Browse** and navigate to `4884code\webservices\mtom`. Select **BlueHills.jpg** and click **Open**.
   c. Click **Invoke** to invoke the `sendImage` method.
   d. In the Result pane, click **View image**. The image is displayed in the Results pane.

Tip: The security settings of your external browser might prevent this sample from being able to access the local files, producing exceptions, such as `FileNotFoundException`. If you encounter this issue, try to use the internal browser (**Window** → **Preferences**: **General** → **Web Browser** → **Use internal Web Browser**).

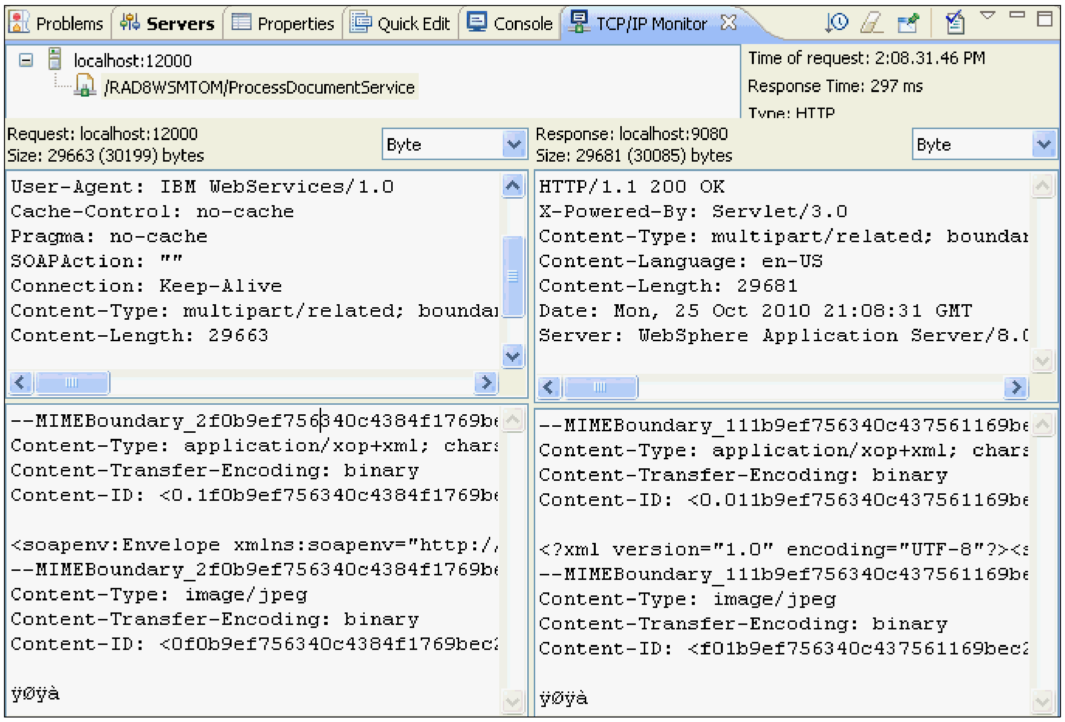

Web Services Test Client
0/RAD8WSMTOMClient/sampleProcessDocumentPortProxy/TestClient.jsp
Methods
_getDescriptor()
sendWordFile(byte[])
sendImage(java.awt.Image)
sendPDFFile (javax.activation.DataHandler)
sendImage
null?
arg0: C:\7835code\webservices\mtom\BlueHills.jpg
Browse...
Invoke
Clear
Result
[View image]
Quality of Service
Endpoint:
http://wxpsp408:9080/RAD8WSMTOM/ProcessDocumer
Update
Reset
Bypass JAXB and use raw SOAP messages

```
BufferedImage@66ed66ed: type = 5 ColorModel: #pixelBits = 24
numComponents = 3 color space =
java.awt.color.ICC_ColorSpace@7a007a00 transparency = 1 has alpha
= false isAlphaPre = false ByteInterleavedRaster: width = 800 height =
600 #numDataElements 3 dataOff[0] = 2
```

Figure 49 Invoking the MTOM web service sendImage method

2. Examine the `4884code\webservices\mtomresult` folder. You can see that the `BlueHills.jpg` file is stored in this folder. The size differs from the size of the same file in the `mtom` folder. Probably, a separate JPEG compression is used.

3. Select the **TCP/IP Monitor** tab to view the SOAP traffic. Click the ▽ icon and then select **Show Header**. Figure 50 shows the HTTP header and the SOAP traffic.



Time of request: 2:08.31.46 PM
Response Time: 297 ms
Type: HTTP

Request: localhost:12000
Size: 29663 (30199) bytes

```
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Connection: Keep-Alive
Content-Type: multipart/related; boundar
Content-Length: 29663
```

```
--MIMEBoundary_2f0b9ef756340c4384f1769be
Content-Type: application/xop+xml; chars
Content-Transfer-Encoding: binary
Content-ID: <0.1f0b9ef756340c4384f1769be

<soapenv:Envelope xmlns:soapenv="http:/
--MIMEBoundary_2f0b9ef756340c4384f1769be
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <0f0b9ef756340c4384f1769bec

ÿØÿà
```

Response: localhost:9080
Size: 29681 (30085) bytes

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: multipart/related; boundar
Content-Language: en-US
Content-Length: 29681
Date: Mon, 25 Oct 2010 21:08:31 GMT
Server: WebSphere Application Server/8.0
```

```
--MIMEBoundary_111b9ef756340c437561169be
Content-Type: application/xop+xml; chars
Content-Transfer-Encoding: binary
Content-ID: <0.011b9ef756340c437561169be

<?xml version="1.0" encoding="UTF-8"?><s
--MIMEBoundary_111b9ef756340c437561169be
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <f01b9ef756340c437561169bec

ÿØÿà
```

Figure 50   SOAP traffic when MTOM is enabled for the web service and client

Look at the SOAP request and response:

- The web service (provider) has MTOM enabled after the code generation. Therefore, the SOAP response has a smaller payload. The web service sends the binary data as a MIME attachment outside of the XML document to realize the optimization.
- The SOAP request also has MTOM enabled and therefore a smaller payload.

Example 26 shows the SOAP response and its HTTP header (manually formatted).

Example 26   HTTP header SOAP response message with MTOM enabled

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: multipart/related;
boundary="MIMEBoundary_111b9ef756340c437561169bec2d4285933d44b19cd7a882";
type="application/xop+xml";
start="<0.011b9ef756340c437561169bec2d4285933d44b19cd7a882@apache.org>";
start-info="text/xml"
Content-Language: en-US
Content-Length: 29681
Date: Mon, 25 Oct 2010 21:08:31 GMT
Server: WebSphere Application Server/8.0
==========================================================================
--MIMEBoundary_111b9ef756340c437561169bec2d4285933d44b19cd7a882
Content-Type: application/xop+xml; charset=UTF-8; type="text/xml"
Content-Transfer-Encoding: binary
Content-ID: <0.011b9ef756340c437561169bec2d4285933d44b19cd7a882@apache.org>
```

```
<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><a:sendIma
geResponse xmlns:a="http://mtom.rad8.ibm.com/"><return><xop:Include
xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:f01b9ef756340c437561169bec2d4285933d44b19cd7a882@apache.org"/></return><
/a:sendImageResponse></soapenv:Body></soapenv:Envelope>
--MIMEBoundary_111b9ef756340c437561169bec2d4285933d44b19cd7a882
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <f01b9ef756340c437561169bec2d4285933d44b19cd7a882@apache.org>
```

The `type` and `content-type` attributes have the value `application/xop+xml`, which indicates that the message was successfully optimized using XOP when MTOM was enabled.

To see the difference when MTOM is not enabled, let us test the new `@MTOM` annotation on the server.

Open the file **ProcessDocumentPortBindingImpl.java** and replace the following line:

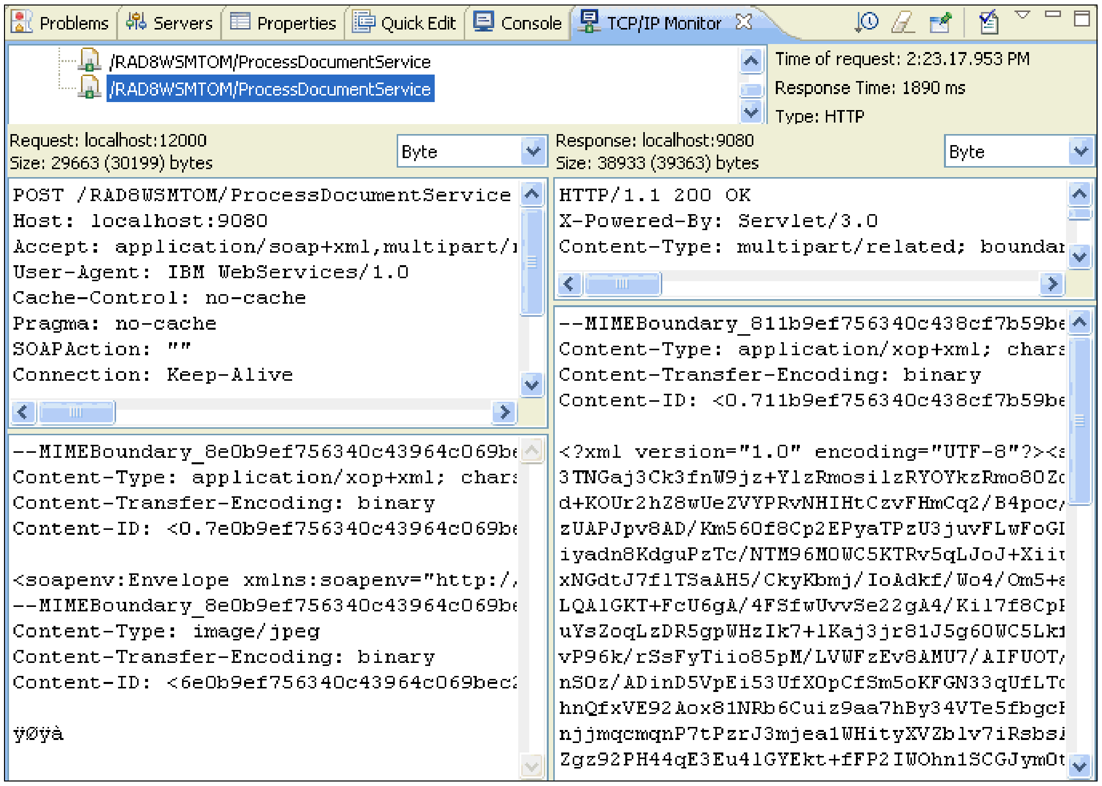
```
@javax.xml.ws.BindingType
(value=javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
```

Replace it with this line:

```
@MTOM(enabled=true,threshold=200000)
```

Then use CTRL+SHIFT+O to arrange the import statements. This new line means that if the attachment is smaller in size than the threshold (in bytes), the service needs to send binary data as base64 encoded data within the XML document, instead of optimizing it using XOP.

After you save the file and republish the server, perform the same test as before and look at the TCP/IP monitor. Figure 51 shows that, in this case, the image is embedded as base64 encoded data in the response.



/RAD8WSMTOM/ProcessDocumentService
/RAD8WSMTOM/ProcessDocumentService

Time of request: 2:23.17.953 PM
Response Time: 1890 ms
Type: HTTP

Request: localhost:12000
Size: 29663 (30199) bytes
Byte

Response: localhost:9080
Size: 38933 (39363) bytes
Byte

```
POST /RAD8WSMTOM/ProcessDocumentService
Host: localhost:9080
Accept: application/soap+xml,multipart/
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Connection: Keep-Alive
```

```
--MIMEBoundary_8e0b9ef756340c43964c069b
Content-Type: application/xop+xml; char
Content-Transfer-Encoding: binary
Content-ID: <0.7e0b9ef756340c43964c069b

<soapenv:Envelope xmlns:soapenv="http:/
--MIMEBoundary_8e0b9ef756340c43964c069b
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <6e0b9ef756340c43964c069bec

ÿØÿà
```

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: multipart/related; bounda
```

```
--MIMEBoundary_811b9ef756340c438cf7b59b
Content-Type: application/xop+xml; char
Content-Transfer-Encoding: binary
Content-ID: <0.711b9ef756340c438cf7b59b

<?xml version="1.0" encoding="UTF-8"?><
3TNGaj3Ck3fnW9jz+YlzRmosilzRYOYkzRmo8OZ
d+KOUr2hZ8wUeZVYPRvNHIHtCzvFHmCq2/B4poc
zUAPJpv8AD/Km56Of8Cp2EPyaTPzU3juvFLwFoG
iyadn8KdguPzTc/NTM96MOWC5KTRv5qLJoJ+Xii
xNGdtJ7flTSaAH5/CkyKbmj/IoAdkf/Wo4/Om5+
LQAlGKT+FcU6gA/4FSfwUvvSe22gA4/Kil7f8Cp
uYsZoqLzDR5gpWHzIk7+lKaj3jr81J5g6OWC5Lk
vP96k/rSsFyTiio85pM/LVWFzEv8AMU7/AIFUOT
nSOz/ADinD5VpEi53UfXOpCfSm5oKFGN33qUfLT
hnQfxVE92Aox81NRb6Cuiz9aa7hBy34VTe5fbgc
njjmqcmqnP7tPzrJ3mjea1WHityXVZblv7iRsbs
Zgz92PH44qE3Eu41GYEkt+fFP2IWOhn1SCGJymOt
```

Figure 51   Using @MTOM with a threshold higher than the attachment size

Example 27 shows the HTTP Header and response body in this case, with the binary data omitted.

Example 27   HTTP Header and response when using @MTOM with a high threshold

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0
Content-Type: multipart/related;
boundary="MIMEBoundary_811b9ef756340c438cf7b59bec2d4285833d44b19cd7a882";
type="application/xop+xml";
start="<0.711b9ef756340c438cf7b59bec2d4285833d44b19cd7a882@apache.org>";
start-info="text/xml"
Content-Language: en-US
Transfer-Encoding: chunked
Date: Mon, 25 Oct 2010 21:23:19 GMT
Server: WebSphere Application Server/8.0
======================================================================
--MIMEBoundary_811b9ef756340c438cf7b59bec2d4285833d44b19cd7a882
Content-Type: application/xop+xml; charset=UTF-8; type="text/xml"
Content-Transfer-Encoding: binary
Content-ID: <0.711b9ef756340c438cf7b59bec2d4285833d44b19cd7a882@apache.org>

<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><a:sendIma
geResponse xmlns:a="http://mtom.rad8.ibm.com/"><return>
...omitted encoded binary data....
</return></a:sendImageResponse></soapenv:Body></soapenv:Envelope>
--MIMEBoundary_811b9ef756340c438cf7b59bec2d4285833d44b19cd7a882--
```

How MTOM was enabled on the client

Let us review how the Web Services wizard has enabled MTOM on the client:

1. In the Enterprise Explorer, expand **RAD8MTOMClient** → **Java Resources** → **src** → **com.ibm.rad8.mtom** and open **ProcessDocumentPortProxy.java**.
2. Review the method `setMTOMEnabled` that is shown in Example 28.

Example 28 How MTOM was enabled on the client

```
public class ProcessDocumentPortProxy{

    protected Descriptor _descriptor;

    public class Descriptor {
.....
      public void setMTOMEnabled(boolean enable) {
            SOAPBinding binding = (SOAPBinding) ((BindingProvider)
            _proxy).getBinding();
            binding.setMTOMEnabled(enable);
        }
    }//end of class Descriptor

    public ProcessDocumentPortProxy() {
        _descriptor = new Descriptor();
        _descriptor.setMTOMEnabled(true);
    }

    public ProcessDocumentPortProxy(URL wsdlLocation, QName serviceName) {
        _descriptor = new Descriptor(wsdlLocation, serviceName);
        _descriptor.setMTOMEnabled(true);
    }
...
}
```

There are now other ways of enabling MTOM on the client. For example, as of JAX-WS 2.2, you can use `@MTOM`. As of JAX-WS 2.1, you can use `MTOMFeature`.

The completed MTOM example is available in the file:

`4884codesolution\webservices\RAD8WSMTOM.zip`

JAX-RS programming model

The JAX-RS programming model is based on the principles of Representational State Transfer (REST) architectures, which were introduced by Roy Fielding in his dissertation *Architectural Styles and the Design of Network-based Software Architectures*:

http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Resources

Resources are the key concept in REST. Resources must be addressable, for example, using Uniform Resource Identifiers (URIs).

Resource representations

Resources can have multiple representations that are suitable for being served to various types of clients. Examples of representations are HTML (to be consumed by web browsers), XML (to be consumed by Java clients), and JavaScript Object Notation (JSON) (to be consumed by JavaScript clients). These representations offered to clients are independent of the way that the actual data referenced by the resources is stored on the server, which can be in a relational database.

Uniform interface

Contrary to SOAP-based web services, where each service defines its own interface, introducing the need for WSDLs to expose the specific interface to clients, in RESTful architectures the set of methods that can be invoked on the resources is limited and well-known. In particular, JAX-RS is based on the HTTP protocol and restricts the possible methods to the HTTP methods, as described in the Hypertext Transfer Protocol -- HTTP/1.1 Request for Comments (RFC):

http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

GET

GET is used to read the resource. It does not change the resource (it is *safe*). Therefore, it can be called multiple times, and it continues to produce the same side effects (it is *idempotent*).

PUT

PUT is used to either insert or update a resource. The client provides the identity (URI) of the resource to update or create. Because the identity of the resource is already known, this method is idempotent, although obviously, it is not safe.

DELETE

Delete is used to delete the resource. It is idempotent.

POST

POST is used to create a new resource on the server. The important difference compared to PUT is that the client of a POST request sends the URI of the resource that *includes* the new resource to be created, while the client of PUT provides the URI of the resources to be created. POST is not idempotent, and it is not safe.

HEAD

HEAD returns the same information as GET, apart from the actual response body. The client receives a status code and headers, if any.

OPTIONS

OPTIONS is used to retrieve the communication options associated with a resource, or the capabilities of a server, without actually retrieving the resource.

Statelessness

The requirement for statelessness communication is present to allow applications to scale, and it is based on the success of the HTTP protocol.

Hypermedia as the Engine of Application State (HATEOAS)

The server response to a request for a given resource must consist of *hypermedia* containing *links* to other resources that the client can access.

Implementation of JAX-RS in WebSphere Application Server

The IBM JAX-RS implementation is based on Apache Wink. Wink is a project developed within the Apache Software Foundation that provides a lightweight framework for developing RESTful applications. Wink supports REST services implemented using JAX-RS to describe the resources on the server. However, a client API is also provided by Wink. This client API is specific to the Wink runtime environment, because there is no JAX-RS-defined client API.

The IBM implementation of JAX-RS 1.1 is an extension of the base Wink 1.1.1 runtime environment. IBM JAX-RS includes the following features:

- JAX-RS server run time
- Stand-alone client API with the option to use Apache HttpClient 4.0 as the underlying client
- Built-in entity provider support for JSON4J
- An Atom JAXB model in addition to Apache Abdera support
- Multipart content support
- A handler system to integrate user handlers into the processing of requests and responses

JAX-RS is supported by the following run times (Table 3).

Table 3 Versions of WebSphere Application Server that support JAX-RS

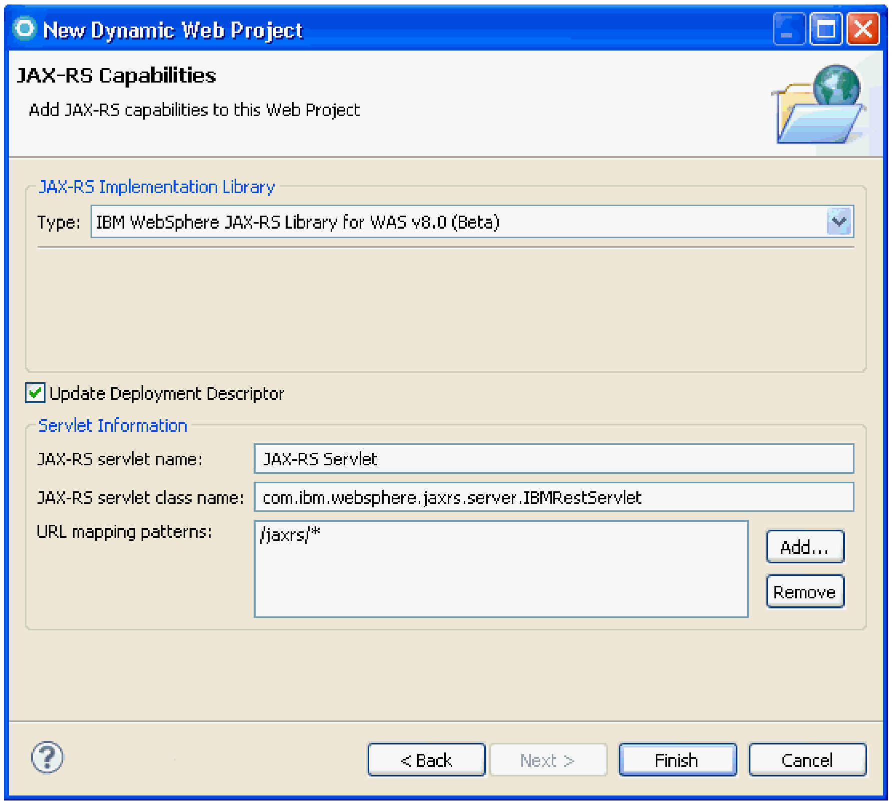
JAX-RS	Notes	WebSphere Application Server
1.0	Requires Dynamic Web Module 2.3 or higher and Java 1.5 or higher	7.0 with Feature Pack for Web 2.0 8.0
1.1	Requires Dynamic Web Module 2.4 or higher and Java 1.6 or higher. Based on Apache Wink 1.1.1	7.0 with Feature Pack for Web 2.0 8.0

JAX-RS project setup

You can set up a JAX-RS enabled project in one of the following two ways, depending on whether you use a new project or an existing one.

Follow these steps to configure a new Dynamic Web project:

1. Select **File** → **New** → **Dynamic Web project**.
2. Perform these steps:
   a. In the Project Name field, enter `RAD8JAX-RSWeb`.
   b. In the Dynamic Web Module version field, select **3.0**.
   c. In the Configuration field, select **IBM JAX-RS Configuration**.
   d. In the EAR field, enter the name `RAD8JAX-RSEAR`.
   e. Select **Next** three times until you reach the window that is shown in Figure 52.


New Dynamic Web Project

JAX-RS Capabilities

Add JAX-RS capabilities to this Web Project

JAX-RS Implementation Library

Type: IBM WebSphere JAX-RS Library for WAS v8.0 (Beta)

☑ Update Deployment Descriptor

Servlet Information

JAX-RS servlet name: JAX-RS Servlet

JAX-RS servlet class name: com.ibm.websphere.jaxrs.server.IBMRestServlet

URL mapping patterns: /jaxrs/*

Figure 52 Adding JAX-RS Capabilities to the web project

3. Following these steps, add the JAX-RS facet. If the target server is V7.0, these steps add the Ajax Proxy and server-side technology facets. The JAX-RS facet adds the library, servlet information, and support for JAX-RS annotations processing and JAX-RS quick fixes.
4. Select **Finish.**

Follow these steps to configure an existing project:

1. Right-click the project and select **Properties**.
2. Select **Project Facets**.
3. In the Configuration field, select **IBM JAX-RS Configuration**.
4. Select **Further configuration required**.
5. In the JAX-RS Implementation Library field, select **IBM WebSphere JAX-RS Library for WAS v8.0 (Beta)**.
6. Select **Update Deployment Descriptor** and select **OK**.

Example 29 shows the servlet and servlet mapping that are added to the Deployment Descriptor (`web.xml`).

Example 29 JAX-RS servlet and servlet mapping in web.xml

```
<servlet>
   <description>
   JAX-RS Tools Generated - Do not modify</description>
   <servlet-name>JAX-RS Servlet</servlet-name>
   <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet
```

```
</servlet-class>
      <load-on-startup>1</load-on-startup>
      <enabled>true</enabled>
      <async-supported>false</async-supported>
   </servlet>
   <servlet-mapping>
      <servlet-name>JAX-RS Servlet</servlet-name>
      <url-pattern>
      /jaxrs/*</url-pattern>
   </servlet-mapping>
```

For more information about the configuration of the IBM JAX-RS servlet `com.ibm.websphere.jaxrs.server.IBMRestServlet`, see these websites:

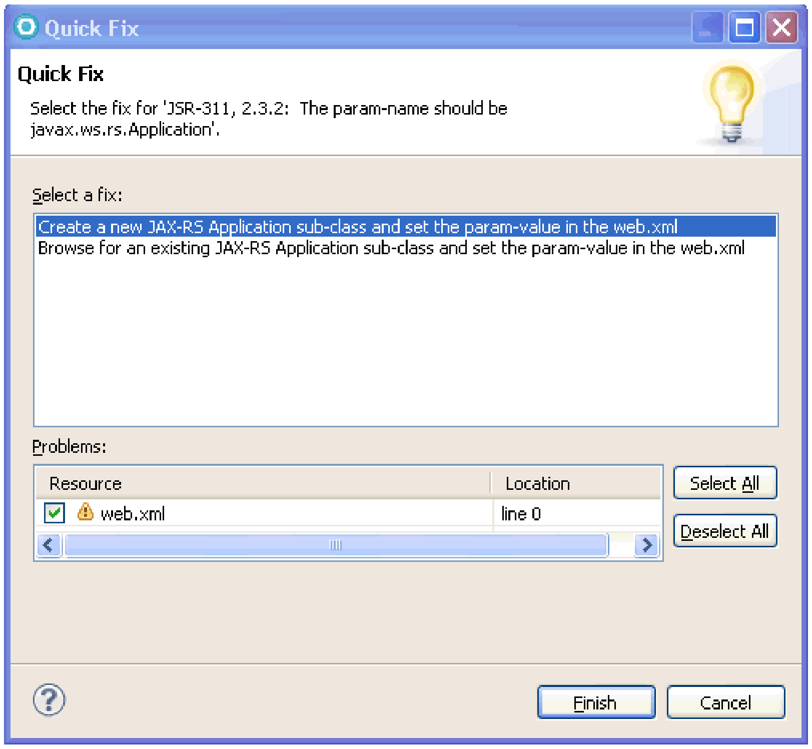
http://www14.software.ibm.com/webapp/wsbroker/redirect?version=matt&product=was-base-dist&topic=twbs_jaxrs_configwebxml

http://www14.software.ibm.com/webapp/wsbroker/redirect?version=v700web&product=was-base-dist&topic=twbs_jaxrs_configwebxml

For both a new and an existing project, we must add a class that extends `javax.ws.rs.core.Application`. This class indicates which classes with the `@Path` and `@Provider` annotations need to be deployed by the JAX-RS run time.

Rational Application Developer provides a quick fix to help you create this class:

1. Open the Deployment Descriptor (**web.xml**).
2. In the Design tab, select **Servlet (JAX-RS Servlet)**.
3. Click **Add**.
4. Select **Initialization Parameter**.
5. Save the `web.xml` file.
6. In the Problems view, the following warning appears: "`JSR-311, 2.3.2: The param-name should be javax.ws.rs.Application`."
7. Right-click the warning and select **Quick Fix**.
8. In the Quick Fix window, select **Create a new JAX-RS Application sub-class and set the param-value in the web.xml**, as shown in Figure 53.
9. In the Class creation wizard, perform these tasks:
   a. For Package, type `itso.bank.jaxrs`.
   b. For Name, type `BankJAXRSApplication`.
   c. Accept the **javax.ws.rs.core.Application** superclass.
   d. Select **Finish**.
10. In the Java editor, right-click and select **Source** → **Override/Implement Methods**.
11. Select the **getClasses()** method from the Application class.
12. Modify the method body of `getClasses()`, as shown in Example 30 on page 79.



Quick Fix

Select the fix for 'JSR-311, 2.3.2: The param-name should be javax.ws.rs.Application'.

Select a fix:

Create a new JAX-RS Application sub-class and set the param-value in the web.xml
Browse for an existing JAX-RS Application sub-class and set the param-value in the web.xml

Problems:

Resource	Location
✓ web.xml	line 0

Figure 53 Quick fix to create a new JAX-RS Application sub-class

Example 30 Sub-class of javax.ws.rs.core.Application

```
package itso.bank.jaxrs;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class BankJAXRSApplication extends Application {

   @Override
   public Set<Class<?>> getClasses() {
      Set<Class<?>> classes = new HashSet<Class<?>>();
      return classes;
   }

}
```

We will add classes that represent resources (annotated with `@Path`) to the collection returned by `getClasses()`. Instances of these classes will be created by the run time with a per-request lifecycle.

You can get the project, which we have developed so far, in this file:

`4884code\webservices\RAD8JAX-RSStart.zip`

Exposing a JPA application as a RESTful service

This section shows how the JPA application can be exposed as a JAX-RS service. First, we import the JPA project `RAD8JPA`:

1. Select **File** → **Import**.
2. Select **General** → **Existing project into Workspace**.
3. Choose **Select archive file** and click **Browse** for `4884codesolution\jpa\RAD8JPA.zip`.
4. Import the project **RAD8JPA**.

Before we can use this project, we must add it to our EAR:

1. Right-click the EAR project **RAD8JAX-RSWeb**.
2. Select **Properties**.
3. Select **Deployment Assembly**.
4. Select **Add**.
5. Select **Project**.
6. Select **Next**.
7. Select **RAD8JPA**.
8. Select **Finish** and then click **OK**.

Now we configure the project for deploying the JPA entities on the server.

Before you continue: To configure this project, you need a connection called `ITSOBANKDerby`. If you do not have this connection, refer to Chapter 9 in *Rational Application Developer for WebSphere Software V8 Programming Guide*, SG24-7835, for detailed information about creating this connection. This book is available at this website:

`http://www.redbooks.ibm.com/abstracts/sg247835.html?Open`

To configure the project, we follow these steps:

1. Make sure that you have the connection called `ITSOBANKDerby`.
2. Configure a data source called `jdbc/itsobank`, either on WebSphere Application Server or by using the enhanced EAR file. You can also use JPA Tools to perform this task:
   a. Right-click **RAD8JPA**.
   b. Select **JPA Tools** → **Configure project for JDBC Deployment**.
   c. In Connection, select **ITSOBANKDerby**.
   d. Clear **Set Up persistence.xml**, because it is already configured.
   e. Select **Deploy connection information to server**.
   f. The data source will be called in the same way that the connection is called, so you need to rename the data source:
      i. Right-click **RAD8JAX-RSEAR**.
      ii. Select **Java EE** → **Open WebSphere Application Server Deployment**.
      iii. Rename `jdbc/ITSOBANKDerby` to `jdbc/itsobank`.

We are now ready to start exposing the JPA entities as JAX-RS resources:

1. In project `RAD8JAX-RSWeb`, create a Java package called `itso.bank.resources`.
2. Add to this package three Java classes:
   - `AccountResource`
   - `CustomerResource`
   - `TransactionResource`
3. In the class `BankJAXRSApplication`, add these three classes to the `HashSet` that is returned by the method `getClasses`, as shown inExample 31.

Example 31 Completed BankJAXRSApplication class

```
package itso.bank.jaxrs;

import itso.bank.resources.AccountResource;
import itso.bank.resources.CustomerResource;
import itso.bank.resources.TransactionResource;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class BankJAXRSApplication extends Application {

   @Override
   public Set<Class<?>> getClasses() {
      Set<Class<?>> classes = new HashSet<Class<?>>();
      classes.add(CustomerResource.class);
      classes.add(AccountResource.class);
      classes.add(TransactionResource.class);
      return classes;
   }
}
```

In order to define the URIs for the Resources, we use the `@Path` annotation.

@Path(@javax.ws.rs.Path)

`@Path` can be applied to Java classes and methods. It is used to define the URI (relative to the web project context-root) for an incoming HTTP request that must be answered by that class/method.

Table 4 shows how the values of the `@Path` annotation map to URIs, based on the previously chosen project name (`RAD8JAX-RSWeb`) and on the JAX-RS servlet mapping already declared in `web.xml`.

Table 4 Mapping of @Path annotations to URIs

Annotation	URL
`@Path(“/customers”)`	`http://hostname:portname/RAD8JAX-RSWeb/jaxrs/customers`
`@Path(“/accounts”)`	`http://hostname:portname/RAD8JAX-RSWeb/jaxrs/accounts`
`@Path(“/transaction”)`	`http://hostname:portname/RAD8JAX-RSWeb/jaxrs/transaction`

In our first implementation of `CustomerResource` (Example 32 on page 82), we only want to return a JSON representation of an array of `Customer` entities. To access the entities, we reuse the JPA Manager Beans (`CustomerManager` in this case) and we make use of JSON4J to convert the Java Representation to JSON.

Example 32 First implementation of CustomerResource.java

```
package itso.bank.resources;

import itso.bank.entities.Customer;
import itso.bank.entities.controller.CustomerManager;

import java.io.IOException;
import java.util.List;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

@Path("/customers")
public class CustomerResource {

   private CustomerManager manager;
   private EntityManagerFactory emf;

   public CustomerResource() {
      super();
      emf = Persistence.createEntityManagerFactory("RAD8JPA");
      manager = new CustomerManager(emf);

   }

   @GET
   @Produces("application/json")
   public JSONArray getAllCustomers() throws IOException {
      final List<Customer> allCustomers = manager.getCustomers();
      JSONArray jsonArray = jsonCustomerArray(allCustomers);
      return jsonArray;

   }
   private static JSONArray jsonCustomerArray(final List<Customer> allCustomers) {
      JSONArray jsonArray = new JSONArray(allCustomers.size());
      for (Customer customer : allCustomers) {
         jsonArray.add(jsonCustomer(customer));
      }
      return jsonArray;
   }
   private static JSONObject jsonCustomer(Customer customer) {
      JSONObject obj = new JSONObject();
      obj.put("title", customer.getTitle());
      obj.put("firstName", customer.getFirstName());
```

```
        obj.put("lastName", customer.getLastName());
        obj.put("ssn", customer.getSsn());
        return obj;
    }

}
```

We have introduced two additional annotations.

@Get(@javax.ws.rs.Get)

This annotation is used to denote a method that performs the safe, idempotent HTTP GET operation.

@Produces(@javax.ws.rs.Produces)

This annotation indicates which media type is returned by a method annotated with `@Get`.

Testing the first implementation of CustomerResource

To test this code, publish the EAR to the server and then simply open a web browser and enter the following URL:

```
http://localhost:9080/RAD8JAX-RSWeb/jaxrs/customers
```

Typically, web browsers do not know how to handle the type “application/json”, which is meant to be consumed by JavaScript, and you are likely prompted to save the file or open it with an application of your choosing. After you open the file in any text editor, you see the contents, as shown in Example 33.

Example 33 JSONArray produced by the method getAllCustomers

```
[{"lastName":"Cui","title":"Mr","firstName":"Henry","ssn":"111-11-1111"},{"lastNam
e":"Fleming","title":"Ms","firstName":"Craig","ssn":"222-22-2222"},{"lastName":"Co
utinho","title":"Mr","firstName":"Rafael","ssn":"333-33-3333"},{"lastName":"Sollam
i","title":"Mr","firstName":"Salvatore","ssn":"444-44-4444"},{"lastName":"Hainey",
"title":"Mr","firstName":"Brian","ssn":"555-55-5555"},{"lastName":"Baber","title":
"Mr","firstName":"Steve","ssn":"666-66-6666"},{"lastName":"Venkatraman","title":"M
r","firstName":"Sundaragopal","ssn":"777-77-7777"},{"lastName":"Ziosi","title":"Mr
","firstName":"Lara","ssn":"888-88-8888"},{"lastName":"Lippmann","title":"Mr","fir
stName":"Sylvi","ssn":"999-99-9999"},{"lastName":"Kumari","title":"Mr","firstName"
:"Venkata","ssn":"000-00-0000"},{"lastName":"Keen","title":"Mr","firstName":"Marti
n","ssn":"000-00-1111"}]
```

We can also test with a stand-alone Java client:

1. Create a new Java project called `RAD8JAX-RSClient`.
2. Right-click the project and select **Properties** → **Java Build Path**.
3. In Libraries, select **Add External Jar** and browse for `<`*`WAS_HOME`*`>\runtimes\com.ibm.jaxrs.thinclient_8.0.0.jar`, which is the redistributable WebSphere Application Server JAX-RS Thin Client that you can use in Java stand-alone applications.

4. Add a new class in the `itso.bank.jaxrs.client` package with the code that is shown in Example 34.

Example 34 GetAllCustomersClient stand-alone JAX-RS Client

```
package itso.bank.jaxrs.client;

import java.io.IOException;
import org.apache.wink.client.ClientResponse;
import com.ibm.json.java.JSONArray;

public class GetAllCustomersClient {

 private org.apache.wink.client.ClientConfig clientConfig = new 
org.apache.wink.client.ClientConfig();
 private org.apache.wink.client.RestClient client = new 
org.apache.wink.client.RestClient(clientConfig);
 private final String base = "http://localhost:9080/RAD8JAX-RSWeb/jaxrs";

 public static void main(String args[]) throws IOException {
  GetAllCustomersClient getAllCustomersClient = new GetAllCustomersClient();
  getAllCustomersClient.getResource(getAllCustomersClient.base + "/customers");

 }

 public JSONArray getResource(String URI) {
  org.apache.wink.client.Resource resource = client.resource(URI);
  ClientResponse response = resource.get();
  System.out.println("Getting: " + URI);
  System.out.println("Received Message:\n" + response.getMessage());
  System.out
    .println("Received Entity:\n" + response.getEntity(JSONArray.class));
  return response.getEntity(JSONArray.class);
 }

}
```

5. Right-click this class and select **Run As** → **Java Application**.
6. You obtain similar output to the output that is shown in the browser.

Example 35 Java application output

```
Getting: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/customers
Received Message:
OK
Received Entity:
[{"lastName":"Cui","title":"Mr","firstName":"Henry","ssn":"111-11-1111"},......
.
```

Finishing the implementation of CustomerResource

We can now implement additional methods for `CustomerResource`, such as `getCustomerByPartialName`, which use a NamedQuery defined in the class `CustomerManager`, as shown in Example 36.

Example 36 Method that invokes a JPA NamedQuery

```
@Path("pname/{pname}")
 @GET
 @Produces("application/json")
 public JSONArray getCustomersByPartialName(@PathParam(value = "pname") String
pname)  {
  final List<Customer> allCustomers = manager.getCustomersByPartialName(pname);
JSONArray jsonArray = jsonCustomerArray(allCustomers);
  return jsonArray;
 }
```

We have introduced a new annotation.

@PathParam(@javax.ws.rs.PathParam)

`@PathParam` is placed in front of an operation parameter. It takes the `value` attribute, which is also referenced in the `@Path` annotation and is enclosed in braces ({}), which allows the JAX-RS run time to inject the (converted) segment of the URL into the Java method parameter.

To test the new method, point the browser at this website:

```
http://localhost:9080/RAD8JAX-RSWeb/jaxrs/customers/pname/Ziosi
```

The browser will return a file that contains this information:

```
[{"lastName":"Ziosi","title":"Mr","firstName":"Lara","ssn":"888-88-8888"}]
```

Mapping Account and Transaction to JSONObject

The next step is to create methods for other NamedQueries that are exposed by `CustomerManager`. These queries can return other types of entities, so we need to first define how all other entities map to `JSONObject`. We define static methods in `AccountResource` (Example 37) and `TransactionResource` (Example 38) to convert `Account` and `Transaction` to `JSONObject`.

Example 37 Code to add to AccountResource

```
public static JSONArray jsonAccountArray(final List<Account> allAccounts) {
  JSONArray jsonArray = new JSONArray(allAccounts.size());
  for (Account account : allAccounts) {
   jsonArray.add(jsonAccount(account));
  }
  return jsonArray;
 }

 public static JSONObject jsonAccount(Account account) {
  JSONObject obj = new JSONObject();
  obj.put("id", account.getId());
  obj.put("balance", account.getBalance());
  return obj;
 }
```

Example 38 Code to add to TransactionResource

```
public static JSONArray jsonTransactionArray(
   final List<Transaction> allTransactions) {
  JSONArray jsonArray = new JSONArray(allTransactions.size());
  for (Transaction transaction : allTransactions) {
   jsonArray.add(jsonTransaction(transaction));
  }
  return jsonArray;
 }

 public static JSONObject jsonTransaction(Transaction transaction) {
  JSONObject obj = new JSONObject();
  obj.put("id", transaction.getId());
  obj.put("amount", transaction.getAmount().toPlainString());
  obj.put("transTime", transaction.getTransTime().toString());
  obj.put("transType", transaction.getTransType());
  return obj;
 }
```

We have chosen to represent `Debit` and `Credit` entities with only one type of resource called `TransactionResource`, which is mapped to a `JSONObject` that has a transaction type (`transType`).

We can now define the additional method of class `CustomerResource`.

Example 39 getAccountsForSSN in CustomerResource

```
@Path("accounts/{ssn}")
 @GET
 @Produces("application/json")
 public JSONArray getAccountsForSsn(@PathParam(value = "ssn") String ssn)  {
  final List<Account> allAccounts = manager.getAccountsForSSN(ssn);
JSONArray jsonArray = AccountResource.jsonAccountArray(allAccounts);
  return jsonArray;
 }
```

To test the new method, point the browser at this website:

```
http://localhost:9080/RAD8JAX-RSWeb/jaxrs/customers/accounts/111-11-1111
```

This step returns a JSONArray with the accounts corresponding to the `ssn`:

```
[{"id":"001-111001","balance":12645.67},{"id":"001-111002","balance":6843.21},{"id
":"001-111003","balance":398.76}]
```

The complete implementation of `AccountResource` conceptually does not differ from `CustomerResource`. You can inspect it in the completed project.

Posting data

`TransactionResource` is more interesting, because you see how to create new transactions (Example 40).

Example 40 Implementation of createTransaction in TransactionResource

```
@POST
 @Consumes("application/json")
 @Produces("application/json")
 public JSONObject createTransaction(JSONObject inputObj)
   throws WebApplicationException {
  String transType = (String) inputObj.get("transType");
  Transaction transaction = null;
  EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("RAD8JPA");
  AccountManager accountManager = new AccountManager(emf);
  Account account = accountManager.findAccountById((String) inputObj
    .get("accountId"));
  if(account ==null)
   throw new WebApplicationException(jSONObjectResponse(Status.BAD_REQUEST,"No
Account Found"));
  Transaction persistedTransaction = null;
  if (transType.equals("Credit")) {
   transaction = new Credit(BigDecimal.valueOf(Double
     .parseDouble(inputObj.get("amount").toString())));
   try {
    transaction.setAccount(account);
    creditManager.createCredit((Credit) transaction);
    persistedTransaction = creditManager
    .findCreditById(transaction.getId());
   } catch (Exception e) {
     throw new
WebApplicationException(jSONObjectResponse(Status.INTERNAL_SERVER_ERROR,e.getMessa
ge()));
   }
  } else if (transType.equals("Debit")) {
   transaction = new Debit(BigDecimal.valueOf(Double
     .parseDouble(inputObj.get("amount").toString())));
   try {
    transaction.setAccount(account);
    debitManager.createDebit((Debit)transaction);
    persistedTransaction = debitManager
    .findDebitById(transaction.getId());
   } catch (Exception e) {
     throw new
WebApplicationException(jSONObjectResponse(Status.INTERNAL_SERVER_ERROR,e.getMessa
ge()));
   }
  } else {
    throw new
WebApplicationException(jSONObjectResponse(Status.BAD_REQUEST,transType+ " should
be Debit or Credit"));
}
  return jsonTransaction(persistedTransaction);
 }
```

This method takes as input a `JSONObject` representing a `Transaction`. It tries first to find the `Account` corresponding to the `accountId` field of the input `JSONObject`. Then it tries to determine whether the input corresponds to a `Credit` or `Debit` transaction, based on the value of the `transType` field of the input `JSONObject`. If both pieces of information can be retrieved successfully, it tries to create a new `Credit` or `Debit` object, and finally it tries to persist it.

We have introduced two new annotations.

@Post(@javax.ws.rs.Post)

The JAX-RS run time directs the HTTP POST requests that match the URL of the enclosing `@PATH` annotation to the method annotated with `@Post`. The method annotated with `@Post` is typically used to create new elements that will have the URL of the enclosing `@PATH` annotation.

@Consumes(@javax.ws.rs.Consumes)

The `@Consumes` annotation defines what media type the method expects to receive as input from the HTTP request.

Managing exceptions in JAX-RS

Many possible error conditions can cause the transaction creation to fail, such as the supplied `accountId` might not match any existing account or the `transType` might be spelled incorrectly, and so on.

The class `javax.ws.rs.WebApplicationException` can be used to represent exceptions in JAX-RS. The constructor that we have used takes as input a `javax.ws.rs.core.Response` object. Because we want to always return a `JSONObject` or a `JSONArray`, even when an error condition occurs, we have constructed a special `Response` object with the help of the following utility class (Example 41).

Example 41 Utility class to return Response based on JSONObject or JSONArray

```
package itso.bank.resources;

import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

public class ErrorUtil {
   public static JSONObject errorObject(String msg){
      JSONObject object = new  JSONObject();
      object.put("error", msg);
      return object;
   }
   public static JSONArray errorArray(String msg){
      JSONObject object = errorObject(msg);
      JSONArray array = new JSONArray();
      array.add(object);
      return array;
   }
   public static Response jSONObjectResponse(Status status, String msg){
  return 
Response.status(Status.BAD_REQUEST).type("application/json").entity(errorObject(ms
g)).build();
```

```
   }
 public static Response jSONArrayResponse(Status status, String msg){
  return
Response.status(Status.BAD_REQUEST).type("application/json").entity(errorArray(msg
)).build();
 }
}
```

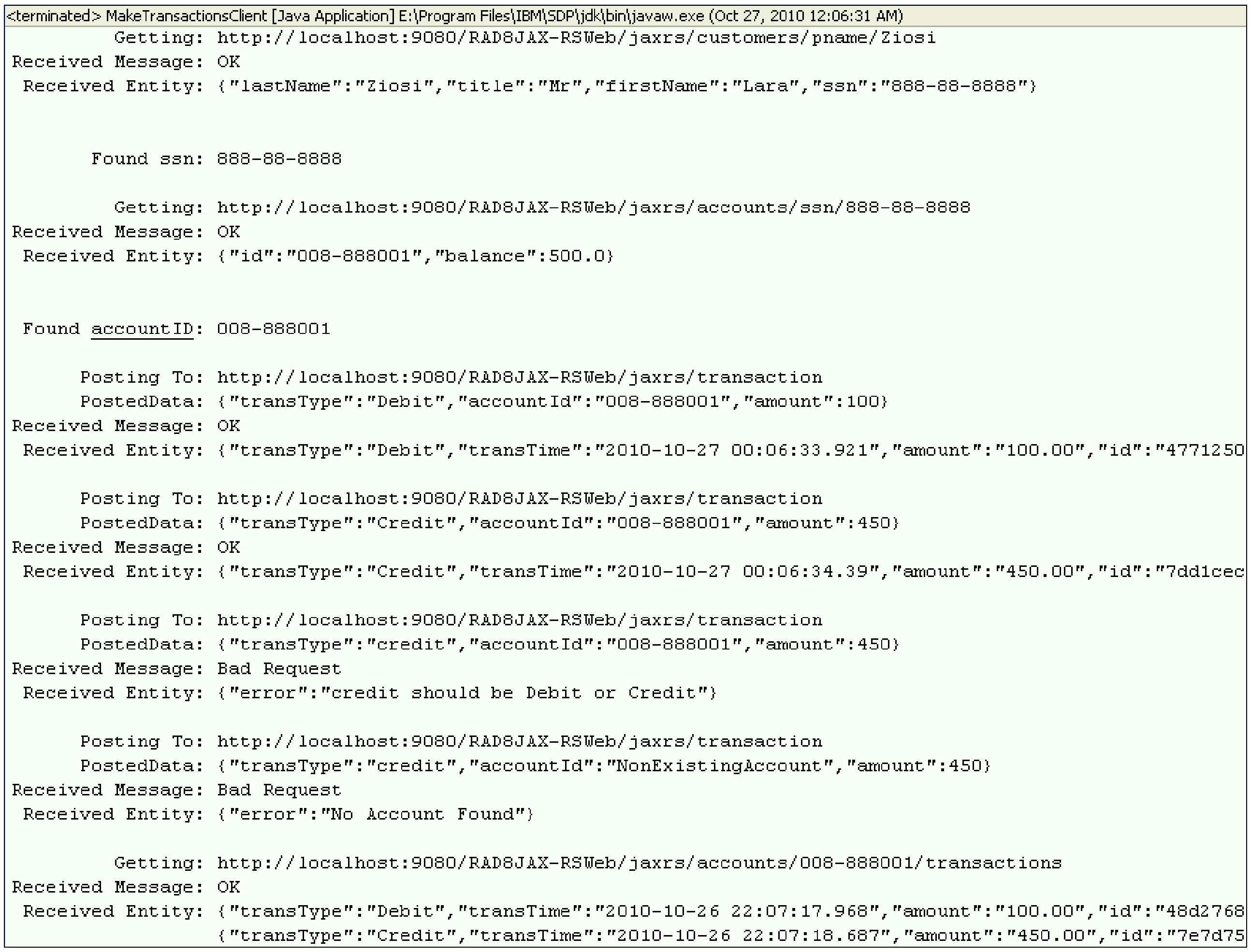
Testing the completed JAX-RS application

After introducing suitable exceptions in all other methods and implementing a method to get all transactions in the `TransactionResource` class, as documented in the provided completed application sources, we are ready to test the complete application with the following code, which must be added to the Java Project `RAD8JAX-RSClient`.

Example 42 Complete application

```
package itso.bank.jaxrs.client;

import java.io.IOException;
import java.math.BigDecimal;
import org.apache.wink.client.ClientResponse;
import org.apache.wink.client.Resource;
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

public class MakeTransactionsClient {

   private org.apache.wink.client.ClientConfig clientConfig = new
org.apache.wink.client.ClientConfig();
   private org.apache.wink.client.RestClient client = new
org.apache.wink.client.RestClient(clientConfig);
   private final String base = "http://localhost:9080/RAD8JAX-RSWeb/jaxrs";

   public static void main(String args[]) throws IOException {
      MakeTransactionsClient makeTransactionsClient = new
MakeTransactionsClient();
      String[] ssn = makeTransactionsClient.getCustomerSSN("Ziosi");
      for (int i = 0; i < ssn.length; i++) {
         String[] accountId = makeTransactionsClient.getAccountId(ssn[i]);
         for (int j = 0; j < accountId.length; j++) {
            makeTransactionsClient.makeTransaction(new BigDecimal(100.00),"Debit",
accountId[j]);
            makeTransactionsClient.makeTransaction(new
BigDecimal(450.00),"Credit", accountId[j]);
            //Bad Request, mis-spelled Credit
            makeTransactionsClient.makeTransaction(new
BigDecimal(450.00),"credit", accountId[j]);
            //Non-existing account
            makeTransactionsClient.makeTransaction(new
BigDecimal(450.00),"credit", "NonExistingAccount");
            makeTransactionsClient.listTransactionsForAccount(accountId[j]);
         }
      }
   }
   private JSONArray getResource(String URI){
        org.apache.wink.client.Resource resource = client.resource(URI);
        resource.contentType("application/json");
        ClientResponse response =resource.get();
        System.out.printf("\n%20s %s\n","Getting:",URI);
        System.out.printf("%20s %s\n","Received Message:",response.getMessage());
        System.out.printf("%20s %s\n","Received
Entity:",prettyPrint(response.getEntity(JSONArray.class)));
        return response.getEntity(JSONArray.class);
   }
   private String[] getCustomerSSN(String pname) throws IOException {
      String customerURI = this.base + "/customers/pname/" + pname;
      JSONArray jsonArray = getResource(customerURI);
      String[] ssn = new String[jsonArray.size()];
      for (int i = 0; i < jsonArray.size(); i++) {
         JSONObject jsonObject = (JSONObject) jsonArray.get(i);
         ssn[i] = (String) jsonObject.get("ssn");
         System.out.printf("\n%20s %s\n","Found ssn:",ssn[i]);
      }
      return ssn;
   }
   private String[] getAccountId(String ssn) throws IOException {
      String accountsURI = base + "/accounts/ssn/" + ssn;
      JSONArray jsonArray = getResource(accountsURI);
      String accountId[] = new String[jsonArray.size()];
      for (int i = 0; i < jsonArray.size(); i++) {
         JSONObject jsonObject = (JSONObject) jsonArray.get(i);
         accountId[i] = (String) jsonObject.get("id");
         System.out.printf("\n%20s %s\n","Found accountID:",accountId[i]);
      }
      return accountId;
   }
   private void makeTransaction(BigDecimal amount, String transType,
         String accountId) throws IOException {
      String transactionURI=base + "/transaction";
      Resource resource = client.resource(transactionURI);
      resource.contentType("application/json");
      JSONObject jsonObj = new JSONObject();
      jsonObj.put("amount", amount);
      jsonObj.put("transType", transType);
      jsonObj.put("accountId", accountId);
      String jsonStr = jsonObj.serialize();
      System.out.printf("\n%20s %s\n","Posting To:",transactionURI);
      System.out.printf("%20s %s\n","PostedData:",jsonStr);
      ClientResponse response = resource.post(jsonObj);
      System.out.printf("%20s %s\n","Received Message:",response.getMessage());
      System.out.printf("%20s %s\n","Received Entity:",
response.getEntity(JSONObject.class));
   }
   private void listTransactionsForAccount(String accountId) {
      String transactionsURI = base + "/accounts/" + accountId
            + "/transactions";
      getResource(transactionsURI);
   }
private String prettyPrint(JSONArray entity) {
  Object[]objects =entity.toArray();
  String pretty="";
  String spaces="                    ";
  for(int i=0;i<objects.length;i++){
   pretty=pretty+objects[i]+"\n"+spaces;
  }
  return pretty;
 }
}
```

Figure 54 shows the results of running this code.



```
<terminated> MakeTransactionsClient [Java Application] E:\Program Files\IBM\SDP\jdk\bin\javaw.exe (Oct 27, 2010 12:06:31 AM)
         Getting: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/customers/pname/Ziosi
Received Message: OK
 Received Entity: {"lastName":"Ziosi","title":"Mr","firstName":"Lara","ssn":"888-88-8888"}


       Found ssn: 888-88-8888

         Getting: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/accounts/ssn/888-88-8888
Received Message: OK
 Received Entity: {"id":"008-888001","balance":500.0}


 Found accountID: 008-888001

      Posting To: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/transaction
      PostedData: {"transType":"Debit","accountId":"008-888001","amount":100}
Received Message: OK
 Received Entity: {"transType":"Debit","transTime":"2010-10-27 00:06:33.921","amount":"100.00","id":"4771250

      Posting To: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/transaction
      PostedData: {"transType":"Credit","accountId":"008-888001","amount":450}
Received Message: OK
 Received Entity: {"transType":"Credit","transTime":"2010-10-27 00:06:34.39","amount":"450.00","id":"7dd1cec

      Posting To: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/transaction
      PostedData: {"transType":"credit","accountId":"008-888001","amount":450}
Received Message: Bad Request
 Received Entity: {"error":"credit should be Debit or Credit"}

      Posting To: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/transaction
      PostedData: {"transType":"credit","accountId":"NonExistingAccount","amount":450}
Received Message: Bad Request
 Received Entity: {"error":"No Account Found"}

         Getting: http://localhost:9080/RAD8JAX-RSWeb/jaxrs/accounts/008-888001/transactions
Received Message: OK
 Received Entity: {"transType":"Debit","transTime":"2010-10-26 22:07:17.968","amount":"100.00","id":"48d2768
                  {"transType":"Credit","transTime":"2010-10-26 22:07:18.687","amount":"450.00","id":"7e7d75
```

Figure 54 Results of invoking MakeTransactionsClient.java

You can locate the completed application in this file:

`4884codesolution\webservices\RAD8JAX-RS.zip.`

Web services security

Web services security for WebSphere Application Server V8.0 is based on standards included in the Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security (WS-Security) Version 1.0/1.1 specification, the Username Token Profile 1.0/1.1, and the X.509 Certificate Token Profile 1.0/1.1.

WS-Security addresses three major issues involved in securing SOAP message exchanges: authentication, message integrity, and message confidentiality.

Authentication

Authentication is used to ensure that parties within a business transaction are really who they claim to be; thus, proof of identity is required. This proof can be claimed in the following ways:

- Presenting a user identifier and password (referred to as a *username token* in the WS-Security domain)
- Using an X.509 certificate issued by a trusted certificate authority

The *certificate* contains identity credentials and has a pair of private and public keys associated with it. The proof of identity presented by a party includes the certificate itself and a separate piece of information that is digitally signed using the certificate's private key. By validating the signed information using the public key associated with the party's certificate, the receiver can authenticate the sender as being the owner of the certificate, thereby validating the sender's identity.

Two WS-Security specifications, the Username Token Profile 1.0/1.1 and the X.509 Certificate Token Profile 1.0/1.1, explain how to use these authentication mechanisms with WS-Security.

Message integrity

To validate that a message has not been tampered with or corrupted during its transmission over the Internet, the message can be digitally signed by using security keys. The sender uses the private key of its X.509 certificate to digitally sign the SOAP request. The receiver uses the sender's public key to check the signature and identity of the signer. The receiver signs the response with its private key. The sender can validate that the response has not been tampered with or corrupted by using the receiver's public key to check the signature and identity of the responder.

Message confidentiality

To keep the message safe from eavesdropping, encryption technology is used to scramble the information in web services requests and responses. The encryption ensures that no one accesses the data in transit, in memory, or after it has been persisted, unless that person has the private key of the recipient. The WS-Security: SOAP Message Security 1.0/1.1 specification describes enhancements to SOAP messaging to provide message confidentiality.

Two options are available to configure WS-Security for JAX-WS web services:

- Policy sets
- Programming API for securing SOAP message with the WS-Security API and Service Programming Interfaces (SPI) for a service provider

We use policy sets in our examples.

Policy set

You can use policy sets to simplify configuring the qualities of service for web services and clients. *Policy sets* are assertions about how web services are defined. By using policy sets, you can combine configurations for separate policies. You can use policy sets with JAX-WS applications, but not with JAX-RPC applications.

A policy set is identified by a unique name. An instance of a policy set consists of a collection of policy types. An empty policy set has no policy instance defined.

Policies are defined on the basis of a quality of service (QoS). Policy definitions are typically based on the WS-Policy standards language. For example, the WS-Security policy is based on the current WS-SecurityPolicy language from the OASIS standards.

Policy sets omit application or user-specific information, such as keys for signing, keystore information, or persistent store information. Instead, application and user-specific information is defined in the bindings. Typically, bindings are specific to the application or the user, and bindings are not normally shared. On the server side, if you do not specify a binding for a policy set, a default binding is used for that policy set. On the client side, you must specify a binding for each policy set.

A *policy set attachment* defines which policy set is attached to service resources, and which bindings are used for the attachment. The bindings define how the policy set is attached to the resources. An attachment is defined outside of the policy set, as metadata associated with the application. To enable a policy set to work with an application, a binding is required.

Applying WS-Security to a web service and client

In this section, we apply the Username WS-Security default policy set to a web service and client. This policy set provides the following features:

- Message integrity by digital signature (using the Rivest-Shamir-Adleman (RSA) algorithm public-key cryptography) to sign the body, time stamp, and WS-Addressing headers using the WS-Security specifications.
- Message confidentiality by encryption (using RSA public-key cryptography) to encrypt the body, signature, and signature confirmation elements using the WS-Security specifications.
- A username token included in the request message to authenticate the client to the service. The username token is encrypted in the request.

Sample bindings for JAX-WS applications

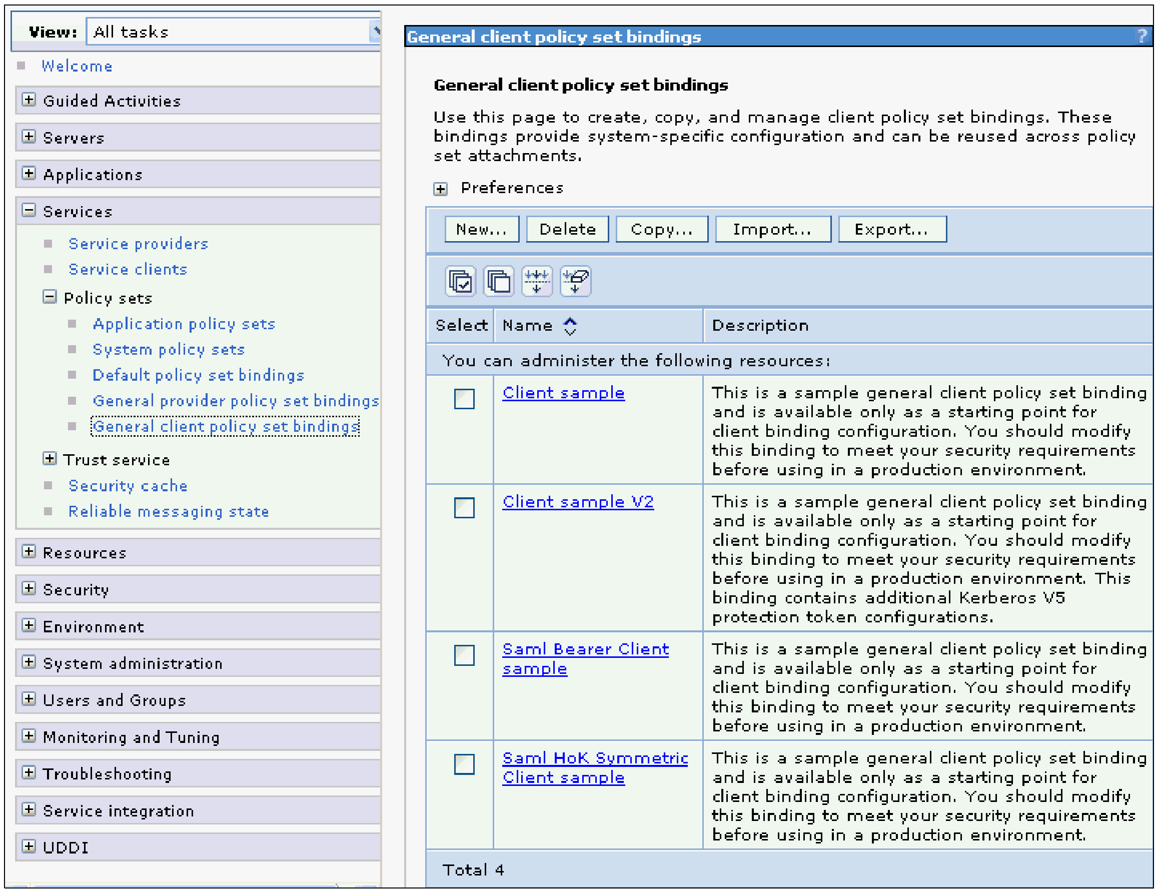
WebSphere Application Server V8.0 includes provider and client sample bindings for testing purposes. In the bindings, the product provides sample values for supporting tokens for various token types, such as the X.509 token and the username token. The bindings also include sample values for message protection information for token types, such as X.509. Both provider and client sample bindings can be applied to the applications attached with a policy set.

In a production environment, you must modify the bindings to meet your security needs before using them in a production environment by making a copy of the bindings and then modifying the copy. For example, you must change the key and keystore settings to ensure security, and you must modify the binding settings to match your environment.

Configuring the username token

When using the Username WS-Security default policy set, you must configure the user name and password for username token authentication separately from the security settings defined in the bindings. The sample binding does not include a user name or password for token authentication, because it is specific to the target deployed system. You must specify a valid user name and password in your environment using the WebSphere administrative console:

1. In the Servers view, right-click **WebSphere Application Server v8.0 Beta** and select **Administration** → **Run administrative console**.
2. Log in with the user ID and password (`admin`). We assume that your WebSphere Profile is secured and that the administrator user is called `admin`.
3. Select **Services** → **Policy sets** → **General client policy set bindings** (Figure 55).



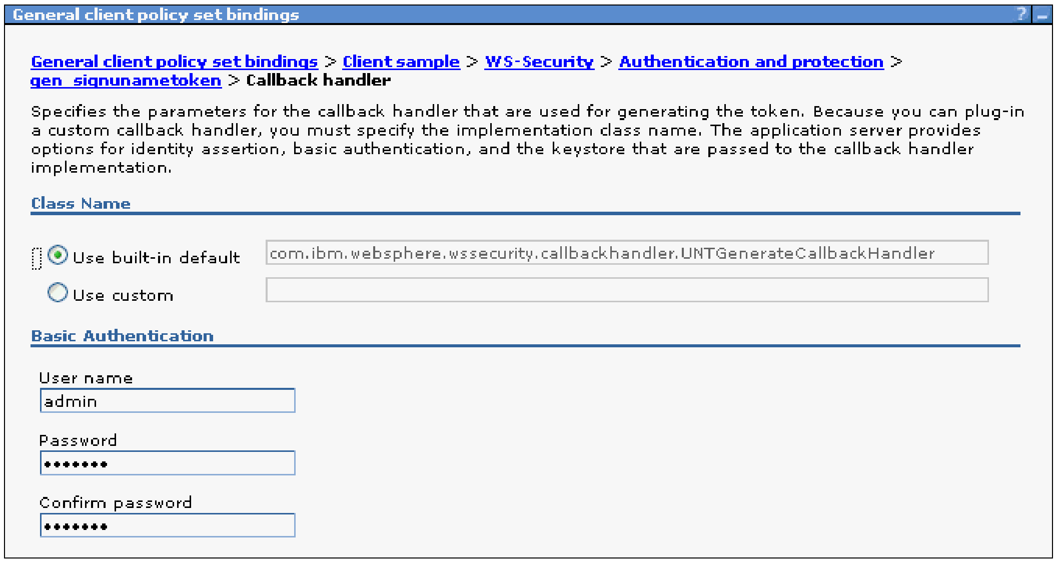
General client policy set bindings

Use this page to create, copy, and manage client policy set bindings. These bindings provide system-specific configuration and can be reused across policy set attachments.

Select	Name	Description
You can administer the following resources:		
☐	Client sample	This is a sample general client policy set binding and is available only as a starting point for client binding configuration. You should modify this binding to meet your security requirements before using in a production environment.
☐	Client sample V2	This is a sample general client policy set binding and is available only as a starting point for client binding configuration. You should modify this binding to meet your security requirements before using in a production environment. This binding contains additional Kerberos V5 protection token configurations.
☐	Saml Bearer Client sample	This is a sample general client policy set binding and is available only as a starting point for client binding configuration. You should modify this binding to meet your security requirements before using in a production environment.
☐	Saml HoK Symmetric Client sample	This is a sample general client policy set binding and is available only as a starting point for client binding configuration. You should modify this binding to meet your security requirements before using in a production environment.
Total 4		

Figure 55 General Client policy set bindings

4. Click **Client sample** to edit the binding.
5. Click **WS-Security**.
6. Click **Authentication and protection**.
7. In the Authentication tokens list, select **gen_signunametoken** to edit the username token settings.
8. In the Additional Bindings section (bottom), click **Callback handler**.
9. For User name, enter `admin`. For password, enter `admin` and confirm the password. Click **Apply** (Figure 56).
10. Click **Save** and then click **Logout**.


General client policy set bindings

General client policy set bindings > Client sample > WS-Security > Authentication and protection > gen_signunametoken > Callback handler

Specifies the parameters for the callback handler that are used for generating the token. Because you can plug-in a custom callback handler, you must specify the implementation class name. The application server provides options for identity assertion, basic authentication, and the keystore that are passed to the callback handler implementation.

Class Name

(●) Use built-in default: com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler

( ) Use custom

Basic Authentication

User name: admin

Password: •••••••

Confirm password: •••••••

Figure 56   Setting Basic Authentication for gen_signunametoken Callback Handler

Attaching the Username WS-Security policy set to the web service

To attach the Username WS-Security default policy set to the web service, follow these steps:

1. In the Java EE perspective, in the Services view (Figure 57), expand the **JAX-WS** node. Right-click **RAD8WebServiceWeb:{...}BankService** and select **Manage Policy Set Attachments** → **Server Side**.

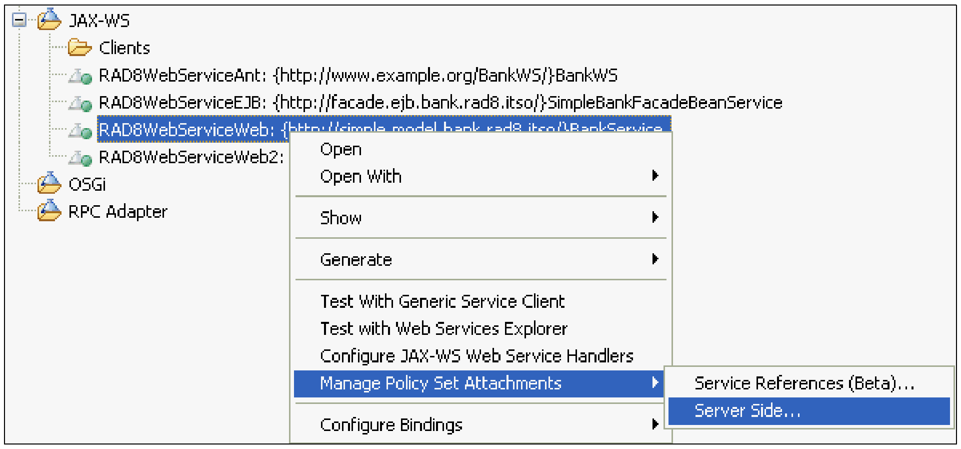
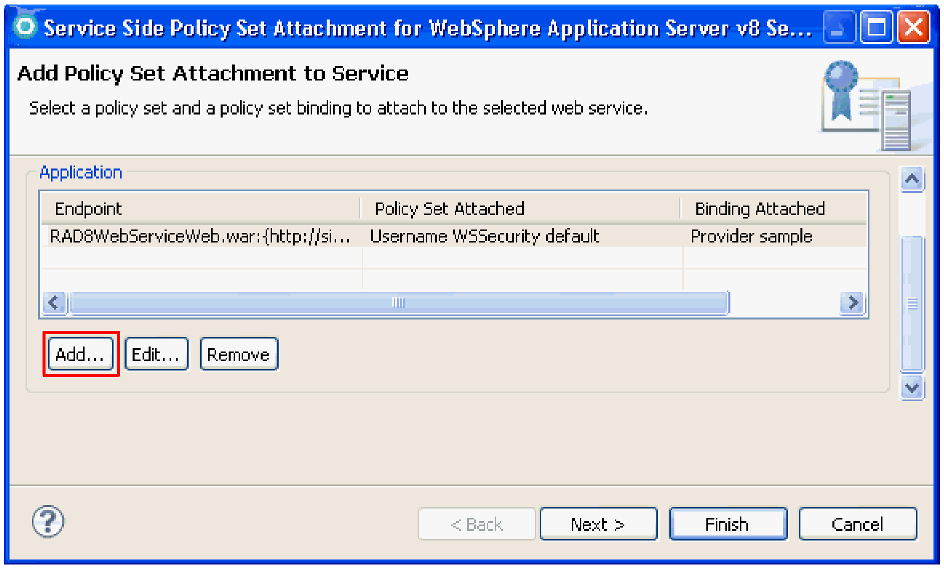

JAX-WS
Clients
RAD8WebServiceAnt: {http://www.example.org/BankWS/}BankWS
RAD8WebServiceEJB: {http://facade.ejb.bank.rad8.itso/}SimpleBankFacadeBeanService
RAD8WebServiceWeb: {http://simple.model.bank.rad8.itso/}BankService
RAD8WebServiceWeb2:
OSGi
RPC Adapter
Open
Open With
Show
Generate
Test With Generic Service Client
Test with Web Services Explorer
Configure JAX-WS Web Service Handlers
Manage Policy Set Attachments
Configure Bindings
Service References (Beta)...
Server Side...

Figure 57   Manage Policy Set Attachments

2. In the Add Policy Set Attachment to Service window (Figure 58 on page 96), click **Add**.
3. In the End Point Definition Dialog window (Figure 59 on page 96), for Policy Set, select **Username WSSecurity default**, and for Binding, ensure that **Provider Sample** is selected. This binding is a service-side general binding packaged with WebSphere Application Server. Click **OK**.



Service Side Policy Set Attachment for WebSphere Application Server v8 Se...

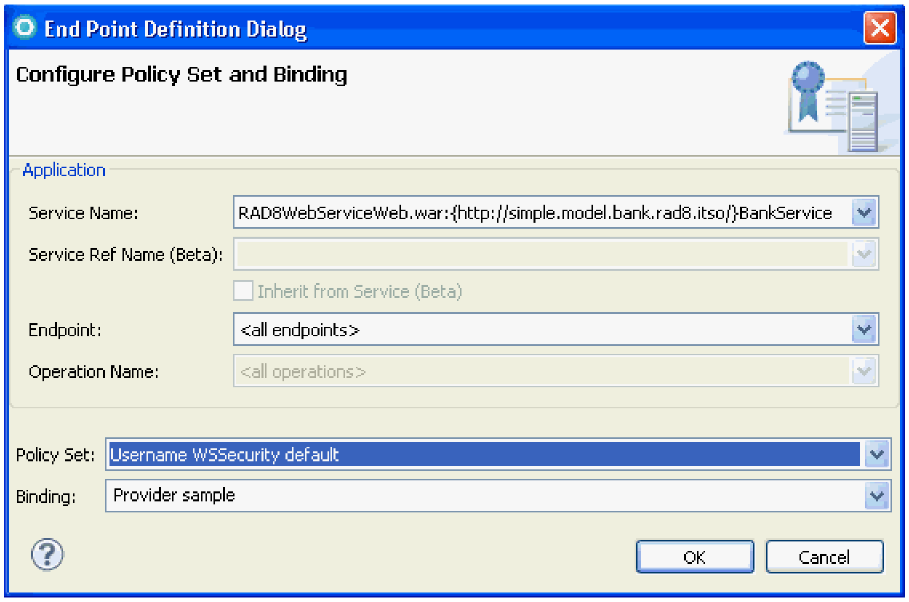
Add Policy Set Attachment to Service

Select a policy set and a policy set binding to attach to the selected web service.

Application

Endpoint	Policy Set Attached	Binding Attached
RAD8WebServiceWeb.war:{http://si...	Username WSSecurity default	Provider sample

Figure 58 Add Policy Set attachment to service



End Point Definition Dialog

Configure Policy Set and Binding

Application

Service Name: RAD8WebServiceWeb.war:{http://simple.model.bank.rad8.itso/}BankService

Service Ref Name (Beta):

Inherit from Service (Beta)

Endpoint: <all endpoints>

Operation Name: <all operations>

Policy Set: Username WSSecurity default

Binding: Provider sample

Figure 59 Configure Policy Set and Binding

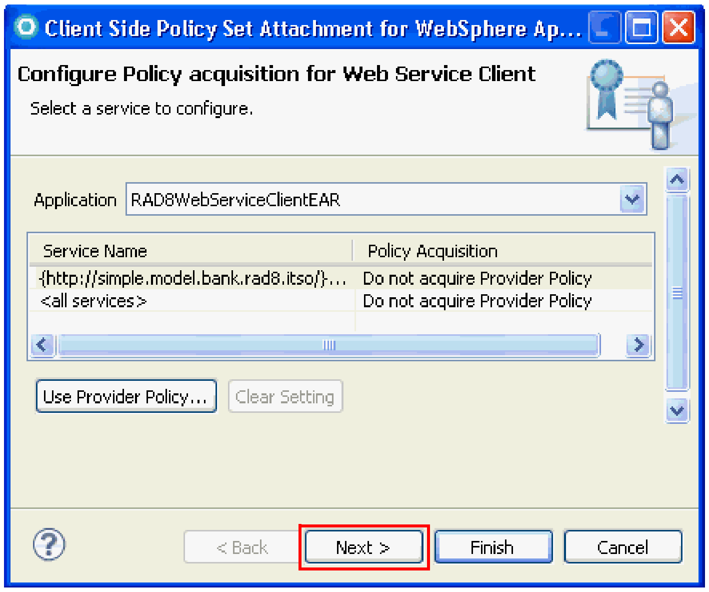
You can apply a policy set at the service, port, or operation level. Separate policy sets can be applied to various endpoints and operations within a single web service. However, the service and client must have the same policy set settings. For this example, we apply the policy set to the entire service, so the Endpoint and Operation Name fields are left blank.

4. When the warning message is displayed, click **Ignore**. WS-Security was included in the WS-I Basic Security Profile. The WS-I Basic Security Profile Version 1.0 was in Final Material status.
5. Back in the Add Policy Set Attachment to Service window, click **Finish**. Notice that the service application is republished to the server.

Attaching the policy set to the web service client

To attach the Username WS-Security default policy set to the web service client, follow these steps:

1. In the Services view, expand the **JAX-WS** → **Clients**. Right-click **RAD8WebServiceClient: service/BankService** and select **Manage Policy Set Attachment**.
2. In the Configure Policy acquisition for Web Service Client window, click **Next** (Figure 60). In this scenario, we do not want to acquire the policy from the Provider, so do not select Use Provider Policy.

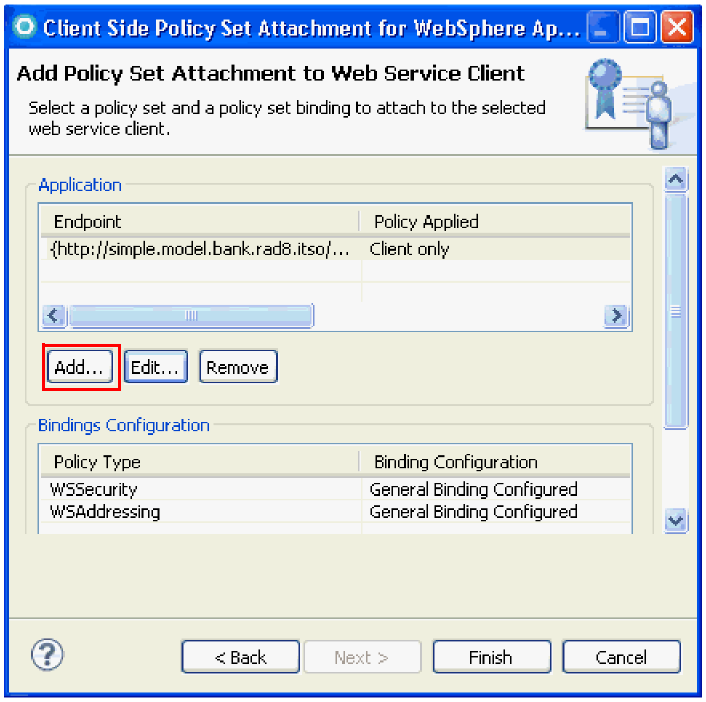

Client Side Policy Set Attachment for WebSphere Ap...
Configure Policy acquisition for Web Service Client
Select a service to configure.
Use Provider Policy...
Clear Setting
< Back
Next >
Finish
Cancel

Application: RAD8WebServiceClientEAR

Service Name	Policy Acquisition
{http://simple.model.bank.rad8.itso/}...	Do not acquire Provider Policy
<all services>	Do not acquire Provider Policy

Figure 60 Configure Policy Acquisition for Web Service Client window

3. In the Add Policy Set Attachment to Client window, click **Add** (Figure 61 on page 98) to attach a policy set to the endpoint and to specify the bindings. The dialog window initially has no entries, and Figure 61 on page 98 shows the result after the addition is complete.

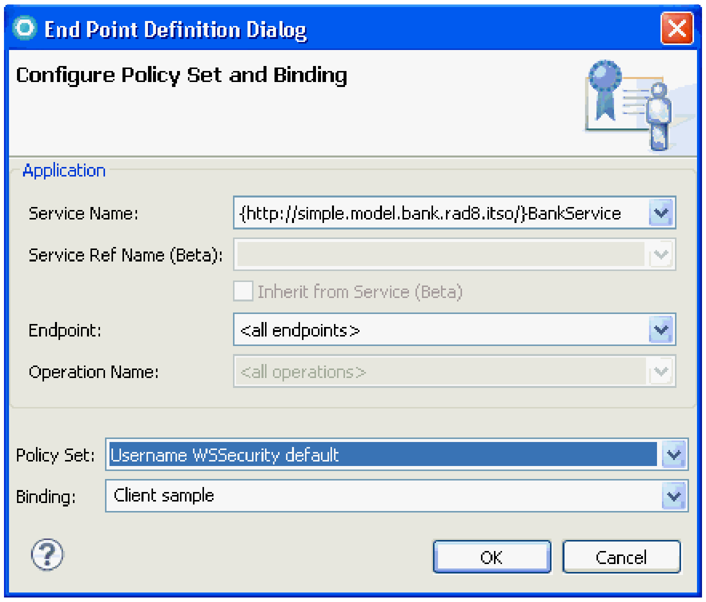

Client Side Policy Set Attachment for WebSphere Ap...
Add Policy Set Attachment to Web Service Client
Select a policy set and a policy set binding to attach to the selected web service client.
Application
Add...
Edit...
Remove
Bindings Configuration
< Back
Next >
Finish
Cancel

Endpoint	Policy Applied
{http://simple.model.bank.rad8.itso/...	Client only

Policy Type	Binding Configuration
WSSecurity	General Binding Configured
WSAddressing	General Binding Configured

Figure 61 Add Policy Set Attachment to Web Service Client window

4. In the End Point Definition Dialog: Configure Policy Set and Binding window (Figure 62), accept the settings for Service Name (**BankService**), Endpoint (**all**), Policy Set (**Username WSSecurity default**), and Binding (**Client sample**). This binding is a client-side general binding packaged with WebSphere Application Server.
5. Click **OK**.


End Point Definition Dialog
Configure Policy Set and Binding
Application
Inherit from Service (Beta)
OK
Cancel

Service Name: {http://simple.model.bank.rad8.itso/}BankService
Service Ref Name (Beta):
Endpoint: <all endpoints>
Operation Name: <all operations>
Policy Set: Username WSSecurity default
Binding: Client sample

Figure 62 Configure Policy Set and Binding window

6. In the message window that opens, click **Ignore**.

   The policy types contained by the policy set that you selected are listed in the Bindings Configuration table. The configurations for these policy types are already complete.

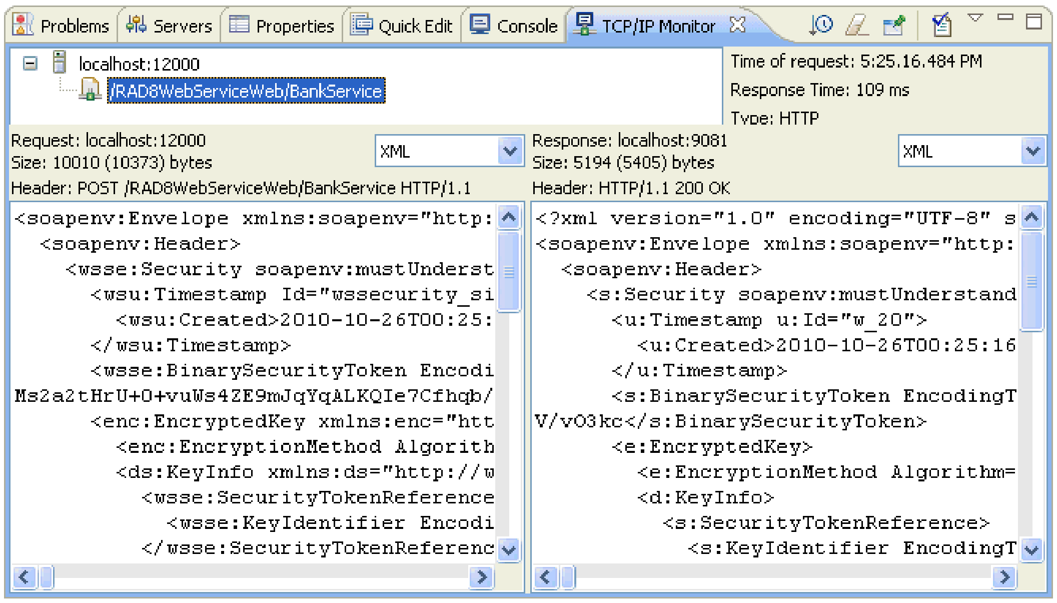
7. Click **Finish** to complete the wizard.

Testing the secured web service

To test the secured web service, follow these steps:

1. Select **Window** → **Preferences**: **Run/Debug** → **TCP/IP Monitor**. Make sure the TCP/IP Monitor is started. Make a note of the monitor port, because you need to reuse it in step 4. Let us call the monitor port *xxxx* for future reference.
2. In the Enterprise Explorer, expand the **RAD8WebServiceClient** project, right-click **TestClient.jsp**, and select **Run As** → **Run on Server**.
3. Select the **v8.0 Beta** server and click **Finish**.
4. In the sample JSP client, Quality of Service pane, change the endpoint to the monitor port and click **Update**:

   `http://localhost:`***`xxxxx`***`/RAD8WebServiceWeb/BankService`
5. Invoke the **retrieveCustomerName** with a customer number of `111-11-1111`.
6. In the TCP/IP Monitor view (Figure 63), verify that the message is signed and encrypted and that the username token in the SOAP header is encrypted.



```
localhost:12000
  /RAD8WebServiceWeb/BankService
```

Time of request: 5:25.16.484 PM
Response Time: 109 ms
Type: HTTP

Request: localhost:12000
Size: 10010 (10373) bytes
Header: POST /RAD8WebServiceWeb/BankService HTTP/1.1

```
<soapenv:Envelope xmlns:soapenv="http:
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderst
      <wsu:Timestamp Id="wssecurity_si
        <wsu:Created>2010-10-26T00:25:
      </wsu:Timestamp>
      <wsse:BinarySecurityToken Encodi
Ms2a2tHrU+0+vuWs4ZE9mJqYqALKQIe7Cfhqb/
      <enc:EncryptedKey xmlns:enc="htt
        <enc:EncryptionMethod Algorith
        <ds:KeyInfo xmlns:ds="http://w
          <wsse:SecurityTokenReference
            <wsse:KeyIdentifier Encodi
          </wsse:SecurityTokenReferenc
```

Response: localhost:9081
Size: 5194 (5405) bytes
Header: HTTP/1.1 200 OK

```
<?xml version="1.0" encoding="UTF-8" s
<soapenv:Envelope xmlns:soapenv="http:
  <soapenv:Header>
    <s:Security soapenv:mustUnderstand
      <u:Timestamp u:Id="w_20">
        <u:Created>2010-10-26T00:25:16
      </u:Timestamp>
      <s:BinarySecurityToken EncodingT
V/vO3kc</s:BinarySecurityToken>
      <e:EncryptedKey>
        <e:EncryptionMethod Algorithm=
        <d:KeyInfo>
          <s:SecurityTokenReference>
            <s:KeyIdentifier EncodingT
```

Figure 63 TCP/IP Monitor showing signed and encrypted message

Figure 43 shows a snippet of the Request.

Example 43 Request snippet

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-sec
ext-1.0.xsd">
      <wsu:Timestamp Id="wssecurity_signature_id_21"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd">
        <wsu:Created>2010-10-26T00:25:16.453Z</wsu:Created>
      </wsu:Timestamp>
```

```
     <wsse:BinarySecurityToken
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message
-security-1.0#Base64Binary"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-prof
ile-1.0#X509v3" Id="x509bst_23"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd">MIICQzCCAaygAwIBA.....
</wsse:BinarySecurityToken>
.....
```

You can obtain the complete request and response at these locations:

- `4884codesolution\webservices\UserNameTokenRequest.txt`
- `4884codesolution\webservices\UserNameTokenResponse.txt`

You can obtain the secured projects archive (where the irrelevant projects have been removed from the two EAR files) in this folder:

`4884codesolution\webservices\RAD8WSUsernameToken.zip`

WS-I Reliable Secure Profile

Continuing from the Reliable Asynchronous Messaging Profile (RAMP) Version 1.0 specification, the Reliable Secure Profile (RSP) working group of the WS-I organization has developed Version 1.0 of an interoperability profile that works with secure and reliable messaging capabilities for web services.

WS-I Reliable Secure Profile 1.0 provides secure reliable session-oriented web services interactions. WS-I Reliable Secure Profile 1.0 builds on WS-I Basic Profile 1.2, WS-I Basic Profile 2.0, WS-I Basic Security Profile 1.0, and WS-I Basic Security Profile 1.1. It adds support for WS-Reliable Messaging 1.1, WS-Make Connection 1.0, and WS-Secure Conversation 1.3:

- WS-Reliable Messaging 1.1 is a session-based protocol that provides message-level reliability for web services interactions.
- WS-Make Connection 1.0 was developed by the WS-Reliable Messaging workgroup to address scenarios in which a web services endpoint is behind a firewall or the endpoint has no visible endpoint reference. If a web services endpoint loses connectivity during a reliable session, WS-Make Connection provides an efficient method to re-establish the reliable session.
- WS-Secure Conversation 1.3 is a session-based security protocol that uses an efficient symmetric key-based encryption algorithm for message-level security.

The configuration steps to apply the WS-I RSP policy set are similar to the steps for the Username WS-Security policy set. Select WS-I RSP for Policy Set when adding a policy set attachment to the service. We leave it as an exercise for you to explore this functionality.

WS-Policy

The Web Services Policy Framework (WS-Policy) specification is an interoperability standard that is used to describe and communicate the policies of a web service so that service providers can export policy requirements in a standard format. Clients can combine the service provider requirements with their own capabilities to establish the policies that are required for a specific interaction.

WebSphere Application Server conforms to the Web Services Policy Framework (WS-Policy) specification. You can use the WS-Policy protocol to exchange policies in a standard format. A policy represents the capabilities and requirements of a web service, for example, whether a message is secure and how to secure it, and whether a message is delivered reliably and how to deliver a message reliably. You can communicate the policy configuration to any other client, service registry, or service that supports the WS-Policy specification, including non-WebSphere Application Server products in a heterogeneous environment.

For a service provider, the policy configuration can be shared in a published WSDL that is obtained by a client using an HTTP get request or by using the Web Services Metadata Exchange (WS-MetadataExchange) protocol. The WSDL is in the standard WS-PolicyAttachments format.

For a client, the client can obtain the policy of the service provider in the standard WS-PolicyAttachments format and use this information to establish a configuration that is acceptable to both the client and the service provider. That is, the client can be configured dynamically, based on the policies supported by its service provider. The provider policy can be attached at the application or service level.

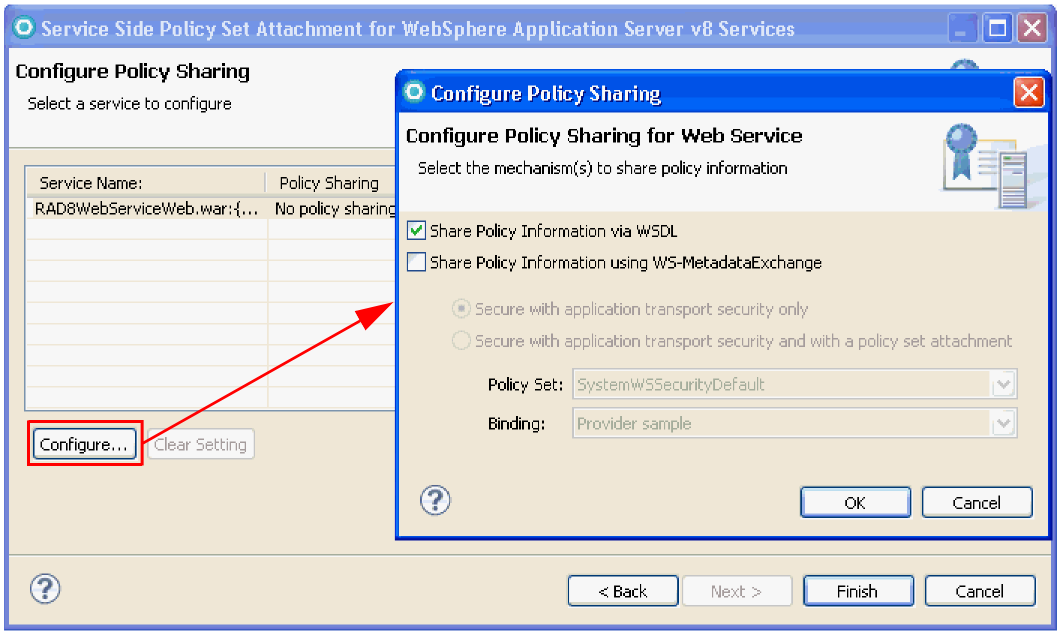
Relationship to policy set: Policy sets are not inherently concerned with the WS-Policy specification, but work with the configuration of web services and need to be considered as a front end to WS-Policy. Policy sets provide a mechanism to specify a policy within a WebSphere environment. They do not provide a mechanism to communicate this policy to non-WebSphere partners in a heterogeneous environment. In addition, policy set functionality does not provide a mechanism for the client to calculate effective policy (that is, a policy that is acceptable to both client and provider) based the intersection of a list of client and provider policies.

Configuring a service provider to share its policy configuration

Configure a service provider to share its policy configuration:

1. In the Services view, right-click **RAD8WebServiceWeb:{...}BankService** and select **Manage Policy Set Attachment**.
2. In the next window, verify that the *username* WS-Security default is listed as the attached policy set from the last section. Click **Next**.

3. In the Configure Policy Sharing window (left window in Figure 64), select the service and click **Configure**. In the Configure Policy Sharing for Web Service window (right window in Figure 64), select **Share Policy Information via WSDL** and click **OK**.


Service Side Policy Set Attachment for WebSphere Application Server v8 Services
Configure Policy Sharing
Select a service to configure
Configure...
Clear Setting
< Back
Next >
Finish
Cancel
Configure Policy Sharing
OK
Cancel

Configure Policy Sharing for Web Service

Select the mechanism(s) to share policy information

Service Name:	Policy Sharing
RAD8WebServiceWeb.war:{...	No policy sharing

- [✓] Share Policy Information via WSDL
- [ ] Share Policy Information using WS-MetadataExchange

- ● Secure with application transport security only
- ○ Secure with application transport security and with a policy set attachment

Policy Set: SystemWSSecurityDefault

Binding: Provider sample

Figure 64 Configure Policy Sharing windows

4. Click **Ignore** for the warning and then click **Finish**.
5. After the server is published, open a browser and enter the following URL in the browser (908*x* is the port number, which is likely 9080):

   `http://localhost:908x/RAD8WebServiceWeb/BankService?wsdl`

The WS-Policy information is embedded in the WSDL document (Example 44). You can see that the policy configured for the input message includes `UsernameToken`.

Example 44 WS-Policy in WSDL

```
......
<binding name="BankPortBinding" type="tns:Bank">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsp:PolicyReference URI="#b2670004-122a-4028-ab78-bf5d286647d6"/>
    <operation name="RetrieveCustomerName">
      <soap:operation soapAction="urn:getCustomerFullName"/>
      <input>
        <soap:body use="literal"/>
    <wsp:PolicyReference URI="#402c7f57-35bb-435d-98bb-f9e3575f4d3e"/>
      </input>
      <output>
        <soap:body use="literal"/>
    <wsp:PolicyReference URI="#b7dd0e57-5b38-4ecd-9f67-10d6c32da5b5"/>
      </output>
      <fault name="CustomerDoesNotExistException">
        <soap:fault name="CustomerDoesNotExistException" use="literal"/>
      </fault>
    </operation>
```

```
  </binding>
....
<wsp:Policy wsu:Id="402c7f57-35bb-435d-98bb-f9e3575f4d3e">
   <ns2:SignedParts
      xmlns:ns2="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
      <ns2:Body />
      <ns2:Header Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing" />
      <ns2:Header Namespace="http://www.w3.org/2005/08/addressing" />
   </ns2:SignedParts>
   <ns2:EncryptedParts
      xmlns:ns2="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
      <ns2:Body />
   </ns2:EncryptedParts>
   <ns2:SignedEncryptedSupportingTokens
      xmlns:ns2="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
      <wsp:Policy>
         <ns2:UsernameToken

ns2:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Includ
eToken/AlwaysToRecipient">
            <wsp:Policy>
               <ns2:WssUsernameToken10 />
            </wsp:Policy>
         </ns2:UsernameToken>
      </wsp:Policy>
   </ns2:SignedEncryptedSupportingTokens>
</wsp:Policy>
....
</definitions>
```

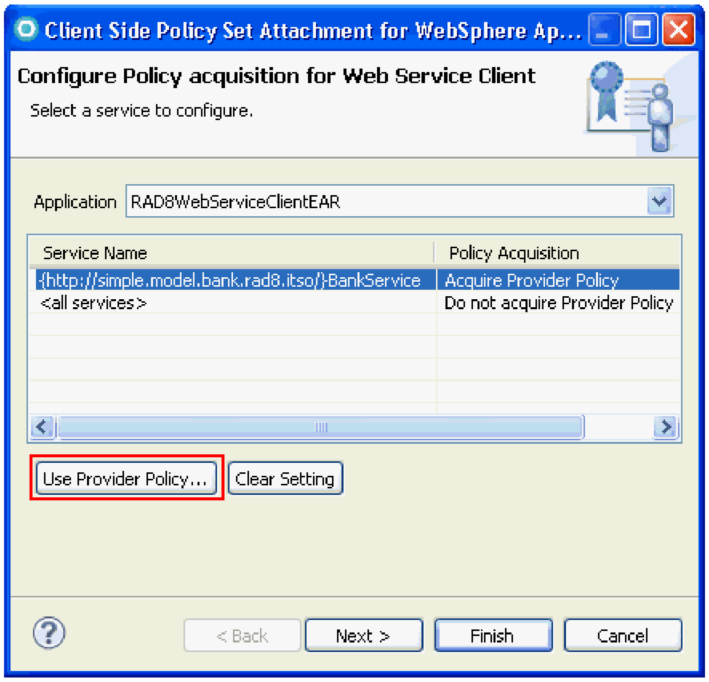
You can see the complete WSDL here:

`4884codesolution\webservices\BankServiceWithUserNameTokenPolicy.wsdl`

Configuring the client policy using a service provider policy

To configure the client policy using a service provider policy, follow these steps:

1. Remove the policy that you applied in the previous section, because we use WS-Policy to request the service provider's policy information:
   a. In the Services view, right-click **RAD8WebServiceClient:service/BankService** and select **Manage Policy Set Attachment**.
   b. Click **Next**.
   c. Click **Remove** and then click **Finish**.
2. Right-click **RAD8WebServiceClient: service/BankService** and select **Manage Policy Set Attachment**.
3. In the first Configure Policy acquisition for Web Service Client window (Figure 65 on page 104), click **Use Provider Policy**.



Configure Policy acquisition for Web Service Client

Select a service to configure.

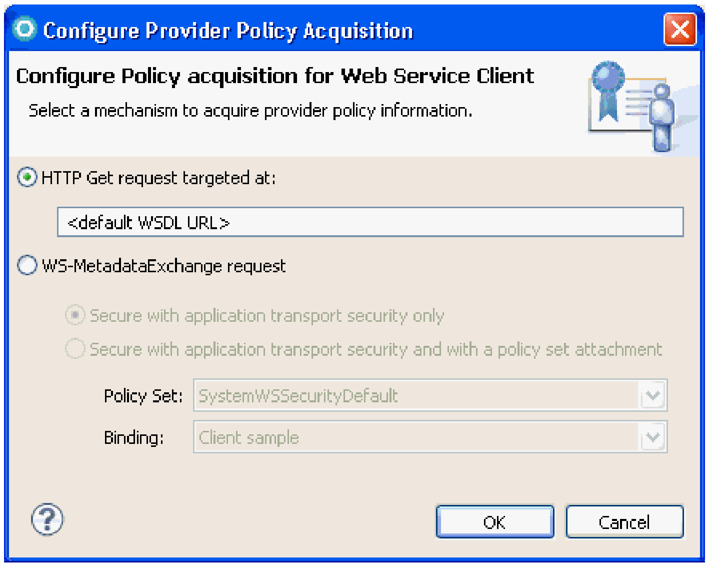
Application: RAD8WebServiceClientEAR

Service Name	Policy Acquisition
{http://simple.model.bank.rad8.itso/}BankService	Acquire Provider Policy
<all services>	Do not acquire Provider Policy

Use Provider Policy... | Clear Setting

Figure 65   Use Provider Policy window

4. In the Configure Policy acquisition for Web Service Client window (Figure 66 on page 104), select **HTTP Get request targeted at <default WSDL URL>** and click **OK**. The Policy Acquisition field for the service changes to `Acquire Provider Policy` in Figure 65.



Configure Policy acquisition for Web Service Client

Select a mechanism to acquire provider policy information.

[✓ HTTP Get request targeted at:] <default WSDL URL>

WS-MetadataExchange request

[✓ Secure with application transport security only]

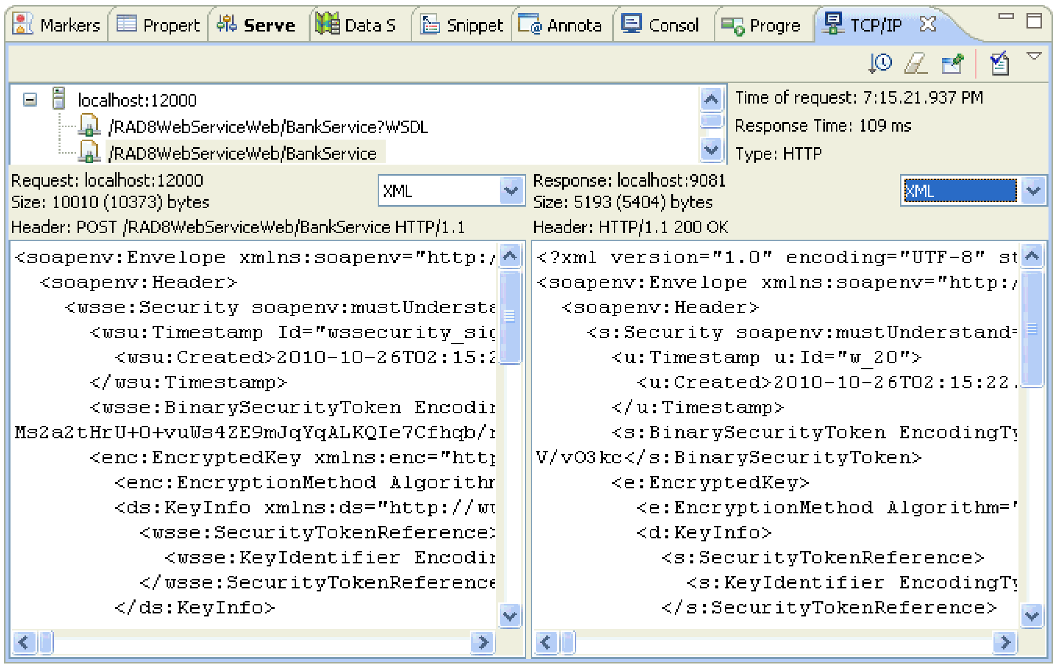
Secure with application transport security and with a policy set attachment

Policy Set: SystemWSSecurityDefault

Binding: Client sample

Figure 66   Configure Policy Acquisition for Web Service Client window

5. In the warning message window that opens, click **Ignore** and then click **Finish**.
6. Test the web service again. In the TCP/IP Monitor, you can see that the client first acquires the WSDL through the HTTP GET (Figure 67 on page 105). The client policy calculations for a service are performed at the first invocation on that service. Calculated policies are cached in the client for performance.



localhost:12000
/RAD8WebServiceWeb/BankService?WSDL
/RAD8WebServiceWeb/BankService

Time of request: 7:15.21.937 PM
Response Time: 109 ms
Type: HTTP

Request: localhost:12000
Size: 10010 (10373) bytes
Header: POST /RAD8WebServiceWeb/BankService HTTP/1.1

Response: localhost:9081
Size: 5193 (5404) bytes
Header: HTTP/1.1 200 OK

```
<soapenv:Envelope xmlns:soapenv="http:/
  <soapenv:Header>
    <wsse:Security soapenv:mustUndersta
      <wsu:Timestamp Id="wssecurity_sic
        <wsu:Created>2010-10-26T02:15:2
      </wsu:Timestamp>
      <wsse:BinarySecurityToken Encodir
Ms2a2tHrU+O+vuWs4ZE9mJqYqALKQIe7Cfhqb/r
      <enc:EncryptedKey xmlns:enc="http
        <enc:EncryptionMethod Algorithr
        <ds:KeyInfo xmlns:ds="http://wu
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier Encodir
          </wsse:SecurityTokenReference
        </ds:KeyInfo>
```

```
<?xml version="1.0" encoding="UTF-8" st
<soapenv:Envelope xmlns:soapenv="http:/
  <soapenv:Header>
    <s:Security soapenv:mustUnderstand=
      <u:Timestamp u:Id="w_20">
        <u:Created>2010-10-26T02:15:22.
      </u:Timestamp>
      <s:BinarySecurityToken EncodingTy
V/vO3kc</s:BinarySecurityToken>
      <e:EncryptedKey>
        <e:EncryptionMethod Algorithm='
        <d:KeyInfo>
          <s:SecurityTokenReference>
            <s:KeyIdentifier EncodingTy
          </s:SecurityTokenReference>
```

Figure 67 TCP/IP Monitor showing retrieval of WSDL

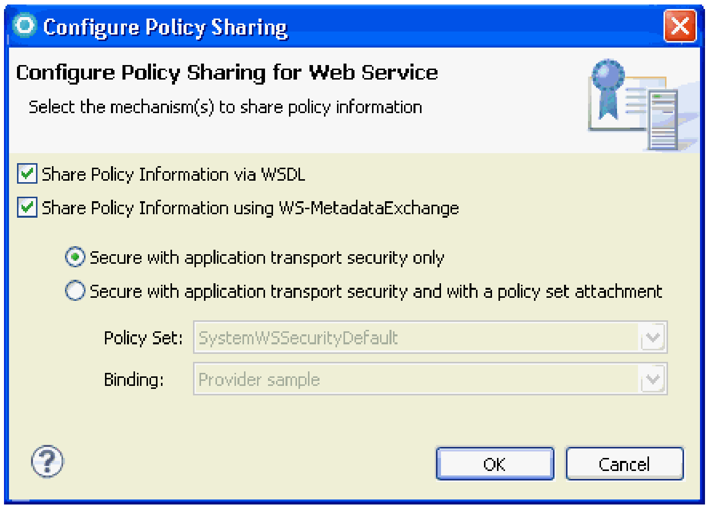
WS-MetadataExchange (WS-MEX)

In WebSphere Application Server V7.0, with JAX-WS, you can enable the Web Services Metadata Exchange (WS-MetadataExchange) protocol so that the policy configuration of the service provider is included in the WSDL and is available to a WS-MetadataExchange `GetMetadata` request. A service provider can use a WS-MetadataExchange request to share its policies. A service client can use a WS-MetadataExchange request to apply the policies of a provider.

One advantage of using the WS-MetadataExchange protocol is that you can apply transport-level or message-level security to WS-MetadataExchange `GetMetadata` requests by using a suitable system policy set. Another advantage is that the client does not have to match the provider configuration, or have a policy set attached. The client only needs the binding information. Then the client can operate based on the provider policy, or based on the intersection of the client and provider policies.

To configure a service provider to share its policy configuration using WS-MEX, follow these steps:

1. In the Services view, right-click **RAD8WebServiceWeb:{...}BankService** and select **Manage Policy Set Attachment**.
2. Verify that the *username* WSSecurity default is listed as the attached policy set from the previous section. Click **Next**.
3. In the Configure Policy Sharing window, select the service and click **Configure**.
4. In the Configure Policy Sharing for Web Service window (Figure 68), select **Share Policy Information using WS-MetadataExchange** and click **OK**.


Configure Policy Sharing

Configure Policy Sharing for Web Service

Select the mechanism(s) to share policy information

[✓] Share Policy Information via WSDL

[✓] Share Policy Information using WS-MetadataExchange

(●) Secure with application transport security only

( ) Secure with application transport security and with a policy set attachment

Policy Set: SystemWSSecurityDefault

Binding: Provider sample

Figure 68 Sharing policy set using WS-MetadataExchange

5. In the warning message window that opens, click **Ignore** and click **Finish**.

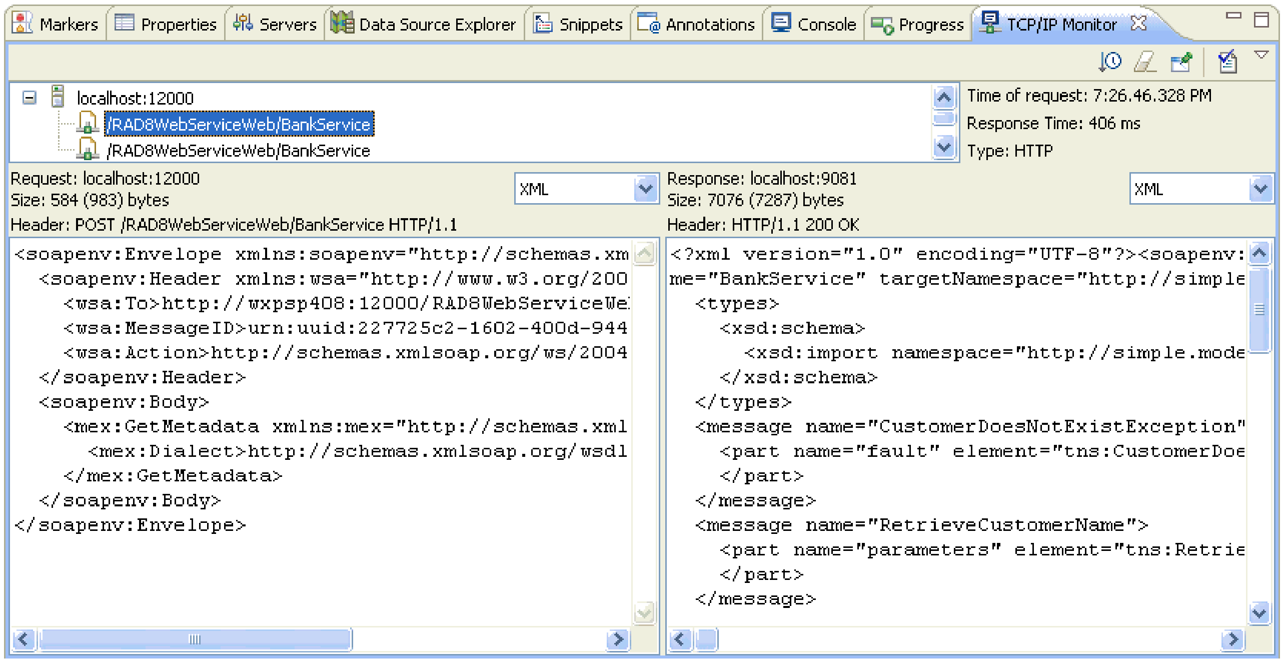
To configure the client policy configuration using WS-MEX, follow these steps:

1. Right-click **RAD8WebServiceClient: service/BankService**, select **Manage Policy Set Attachment** and click **Use Provider Policy**.
2. In the Configure Policy acquisition for Web Service Client window, select **WS-MetadataExchange** and click **OK**.
3. In the warning message window that opens, click **Ignore** and click **Finish**.
4. Test the web service again. In the TCP/IP Monitor, you can see that the client first issues a WS-MEX `GetMetadata` request to the actual web service endpoint and that the dialect of the request is WSDL (Example 45).

Example 45 WS-MEX request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsa:To>http://wxpsp408:12000/RAD8WebServiceWeb/BankService</wsa:To>

<wsa:MessageID>urn:uuid:227725c2-1602-400d-9449-50a0e966058b</wsa:MessageID>

<wsa:Action>http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Request</w
sa:Action>
  </soapenv:Header>
  <soapenv:Body>
    <mex:GetMetadata xmlns:mex="http://schemas.xmlsoap.org/ws/2004/09/mex">
      <mex:Dialect>http://schemas.xmlsoap.org/wsdl/</mex:Dialect>
    </mex:GetMetadata>
  </soapenv:Body>
</soapenv:Envelope>
```

The `GetMetadata` response returns the WSDL with the policy information (Figure 69 on page 107). Then you see a second request in the TCP/IP Monitor with the actual request and response.



localhost:12000
/RAD8WebServiceWeb/BankService
/RAD8WebServiceWeb/BankService

Time of request: 7:26.46.328 PM
Response Time: 406 ms
Type: HTTP

Request: localhost:12000
Size: 584 (983) bytes
Header: POST /RAD8WebServiceWeb/BankService HTTP/1.1

XML

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xm
  <soapenv:Header xmlns:wsa="http://www.w3.org/200
    <wsa:To>http://wxpsp408:12000/RAD8WebServiceWe
    <wsa:MessageID>urn:uuid:227725c2-1602-400d-944
    <wsa:Action>http://schemas.xmlsoap.org/ws/2004
  </soapenv:Header>
  <soapenv:Body>
    <mex:GetMetadata xmlns:mex="http://schemas.xml
      <mex:Dialect>http://schemas.xmlsoap.org/wsdl
    </mex:GetMetadata>
  </soapenv:Body>
</soapenv:Envelope>
```

Response: localhost:9081
Size: 7076 (7287) bytes
Header: HTTP/1.1 200 OK

XML

```
<?xml version="1.0" encoding="UTF-8"?><soapenv:
me="BankService" targetNamespace="http://simple
  <types>
    <xsd:schema>
      <xsd:import namespace="http://simple.mode
    </xsd:schema>
  </types>
  <message name="CustomerDoesNotExistException"
    <part name="fault" element="tns:CustomerDoe
    </part>
  </message>
  <message name="RetrieveCustomerName">
    <part name="parameters" element="tns:Retrie
    </part>
  </message>
```

Figure 69   TCP/IP Monitor showing WS-MEX Request

Security Assertion Markup Language (SAML) support

The Security Assertion Markup Language (SAML) is an XML-based OASIS standard for exchanging user identity and security attributes information.

Using the product SAML function, you can apply policy sets to JAX-WS applications to use SAML assertions in web services messages and in web services usage scenarios. You can use SAML assertions to represent user identity and user security attributes, and optionally, to sign and to encrypt SOAP message elements. WebSphere Application Server supports SAML assertions using the bearer subject confirmation method and the holder-of-key subject confirmation method as defined in the OASIS Web Services Security SAML Token Profile Version 1.1 specification. Policy sets and general bindings that support SAML are included with the product SAML function. To use SAML assertions, you must modify the provided sample general binding.

The SAML function also provides a set of application programming interfaces (APIs) that can be used to request SAML tokens from a Security Token Service (STS) using the WS-Trust protocol. APIs are also provided to locally generate and validate SAML tokens.

SAML assertions defined in the SAML Token Profile standard

The Web Services Security SAML Token Profile OASIS standard specifies how to use Security Assertion Markup Language (SAML) assertions with the Web Services Security SOAP Message Security specification.

WebSphere Application Server Version 7.0.0.7 and later supports two versions of the OASIS SAML standard: Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1, and Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0.

The standard describes the use of SAML assertions as security tokens in the `<wsse:Security>` header, as defined by the WSS: SOAP Message Security specification. An XML signature can be used to bind the subjects and statements in the SAML assertion to the SOAP message.

Subject confirmation methods

Subject confirmation methods define the mechanism by which an entity provides evidence (proof) of the relationship between the subject and the claims of the SAML assertions. The WSS: SAML Token Profile describes the usage of three subject confirmation methods: *bearer*, *holder-of-key*, and *sender-vouches*. WebSphere Application Server Version 7.0.0.9 and later versions support all three confirmation methods.

Bearer

When using the bearer subject confirmation method, proof of the relationship between the subject and claims is implicit. No specific steps are taken to establish the relationship.

Because no key material is associated with a bearer token, protection of the SOAP message, if required, must be performed by using a transport-level mechanism or another security token, such as an X.509 or Kerberos token, for message-level protection.

Holder-of-key

When using the holder-of-key subject confirmation method, proof of the relationship between the subject and claims is established by signing part of the SOAP message with the key specified in the SAML assertion. Because there is key material associated with a holder-of-key token, this token can be used to provide message-level protection (signing and encryption) of the SOAP message.

Sender-vouches

The sender-vouches confirmation method is used when a server needs to propagate the client identity with SOAP messages on behalf of the client. This method is similar to identity assertion, but it has the added flexibility of using SAML assertions to propagate not only the client identity, but also propagate client attributes. The attesting entity must protect the vouched for SAML assertions and SOAP message content so that the receiver can verify that it has not been altered by another party.

SAML APIs

The `SAMLTokenFactory` API is the major SAML token programming interface. Using this API, you can create SAML tokens, insert SAML attributes, parse and validate SAML assertions as XML representations for the SAML tokens, and create Java Authentication and Authorization Service (JAAS) subjects that represent user identity and attributes as defined in SAML tokens. For more information, refer to the WebSphere Application Server Information Center and look for these classes:

- `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory`
- `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken`

http://www14.software.ibm.com/webapp/wsbroker/redirect?version=compass&product=was-base-dist&topic=cwbs_overviewsamlapis

SAML Bearer sample: Prerequisites

This sample shows how you can bind the SAML11 Bearer WSHTTPS default policy set, which contains the items listed in Table 5.

Table 5 SAML11 Bearer WSHTTPS default policy set

SAML11 Bearer WSHTTPS default	
Policies	HTTP transport, SSL transport, WS-Addressing, and WS-Security
Transport security	Using SSL for HTTP
Message authentication	Using SAML 1.1 token with bearer confirmation method

The client is configured to generate a SAML Bearer Token, and the service is configured to consume it. The client makes use of the SAML APIs to create the SAML Token programmatically.

The following steps are required if you use WebSphere Application Server V8:

1. Launch the administrative console.
2. Select **Services** → **Policy Sets** → **Application Policy Sets**.
3. Select **SAML11 Bearer WSHTTPS default**.
4. Select **Import** → **From Default Repository** and select **OK.**
5. Select **Save.**
6. Optional: Explore the other relevant features that are predefined:
   - Verify that in **Services** → **General Client Policy Set Bindings**, you have these bindings:
     - `Saml Bearer Client sample`
     - `Saml HoK Symmetric Client sample`
   - Verify that **Security** → **Global security** → **Java Authentication and Authorization Service** → **System logins** contains these logins:
     - `wss.consume.saml`
     - `wss.generate.saml`
7. Log out of the administrative console.
8. Restart WebSphere Application Server.

If you use WebSphere Application Server V7, additional steps are required as described here:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/twbs_setupsamlconfig.html`

Additionally, you must export the same policy set from WebSphere Application Server and import it into your Rational Application Developer workspace with the following steps:

1. Launch the administrative console.
2. Select **Services** → **Policy Sets** → **Application Policy Sets**.
3. Select **SAML11 Bearer WSHTTPS default**.
4. Select **Export**.
5. A new page opens with a hyperlink to the file `SAML11 Bearer WSHTTPS default.zip`.
6. Right-click the hyperlink and use your browser menu to save the target locally (typically, **Save target as**). Make note of where you save the file.
7. In Rational Application Developer, select **File** → **Import**.

8. Select **WebServices** → **WebSphere Policy Sets** and select **Next**.
9. Browse to the **SAML11 Bearer WSHTTPS default.zip** file.
10. Select **Finish.**

You must repeat the import operation in each workspace where you want to be able to associate this policy set to a service or client.

SAML Bearer sample: Bindings

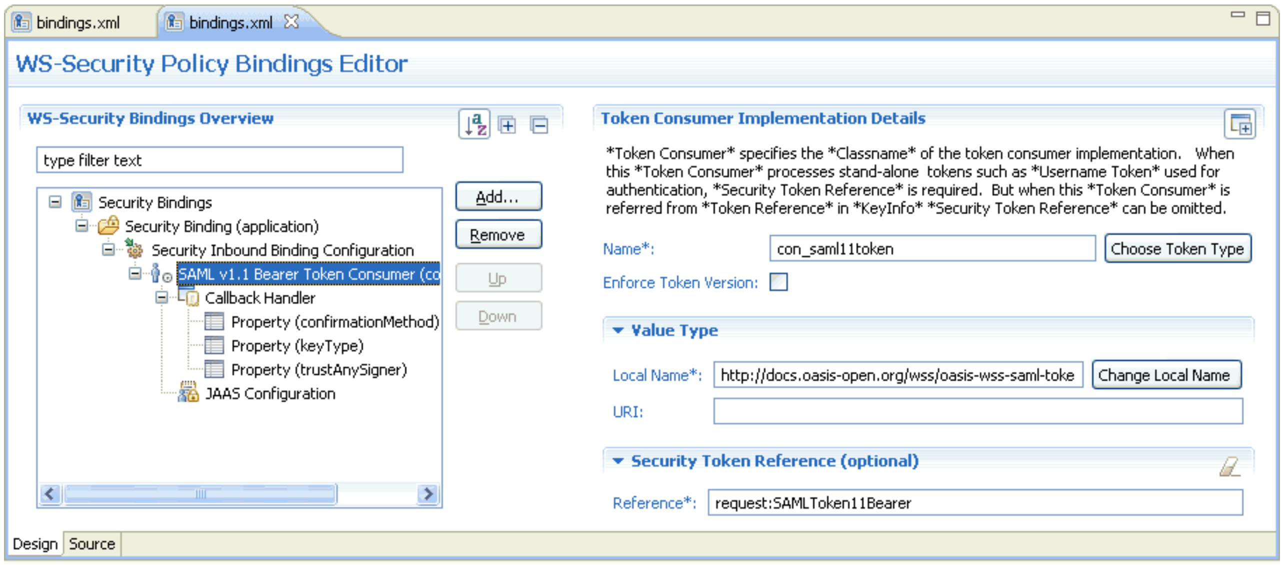
This sample is based on the sample binding that is contained the product. Select **Help** → **Help Contents: Samples** → **WebServices** → **WebSphere JAX-WS address book SAML Web service**. From here, we reuse the supplied bindings and the code to generate a SAML token in the client application:

1. Import the **WebSphere JAX-WS address book SAML Web service** sample, which creates the following projects:
   - `SAMLBearer_AddressBook`
   - `SAMLBearer_AddressBookClient`
   - `SAMLBearer_AddressBookEAR`

   We leave it for you to test this sample. In the remainder of this section, we show how to reuse the bindings for the `RAD8TopDownBankEAR` application, which we developed in , “Creating a top-down web service from a WSDL” on page 55.
2. If you no longer have it in your workspace, import the archive:

   `4884codesolution\webservices\RAD8TopDownWebService.zip`
3. Generate a service client for `RAD8TopDownWebService`:
   a. In the Services view, right-click **RAD8TopDownBankWS**.
   b. Select **Generate** → **Client**.
   c. Move the slider to the **Test** position, because we want to generate a test of the JAX-WS JSP.
   d. Change the client project to `RAD8TopDownBankWSClient`.
   e. Change the EAR project to `RAD8TopDownBankWSClientEAR`.
   f. Select **Next**.
   g. Select **Next**.
   h. On the Web Service Client Test page, make sure that you accept **JAX-WS JSPs** for Test Facility and accept all defaults.
   i. Select **Finish**.
4. Expand **SAMLBearer_AddressBookEAR** in the Enterprise Explorer.
5. Expand the **META-INF** folder.
6. You see two folders that contain application-specific bindings for the service provider and for the service client:
   - `SAMLBearerProviderBinding`
   - `SAMLBearerClientBinding`
7. Copy the complete folder `SAMLBearerProviderBinding` into `RAD8TopDownBankWSEAR\META-INF`. (The folder `META-INF` needs to be added if it does not exist.)

8. Open the file named **RAD8TopDownBankWSEAR\META-INF\SAMLBearerProvider Binding\PolicyTypes\WSSecurity\bindings.xml**. The WS-Security Policy Binding Editor (Figure 70) shows that the Security Inbound Configuration is configured with the SAML V1.1 Bearer Token Consumer.


WS-Security Policy Bindings Editor

WS-Security Bindings Overview

- Security Bindings
  - Security Binding (application)
    - Security Inbound Binding Configuration
      - SAML v1.1 Bearer Token Consumer (co
        - Callback Handler
          - Property (confirmationMethod)
          - Property (keyType)
          - Property (trustAnySigner)
        - JAAS Configuration

Token Consumer Implementation Details

Token Consumer specifies the *Classname* of the token consumer implementation. When this *Token Consumer* processes stand-alone tokens such as *Username Token* used for authentication, *Security Token Reference* is required. But when this *Token Consumer* is referred from *Token Reference* in *KeyInfo* *Security Token Reference* can be omitted.

Name*: con_saml11token

Enforce Token Version: ☐

Value Type

Local Name*: http://docs.oasis-open.org/wss/oasis-wss-saml-toke

URI:

Security Token Reference (optional)

Reference*: request:SAMLToken11Bearer

Figure 70 WS-Security Policy Binding Editor: Provider WSSecurity Binding

The Callback Handler has the following class name:

`com.ibm.websphere.wssecurity.callbackhandler.SAMLConsumerCallbackHandler`

The `SAMLConsumerCallBackHandler` uses three properties (Table 6 on page 111).

Table 6 SAMLConsumerCallBack Handler properties

Name	Value
Bearer	confirmationMethod
keyType	`http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer`
trustAnySigner	true

The JAAS Configuration uses the `system.wss.consume.saml` JAAS Login Name.

9. Example 46 shows the complete binding file.

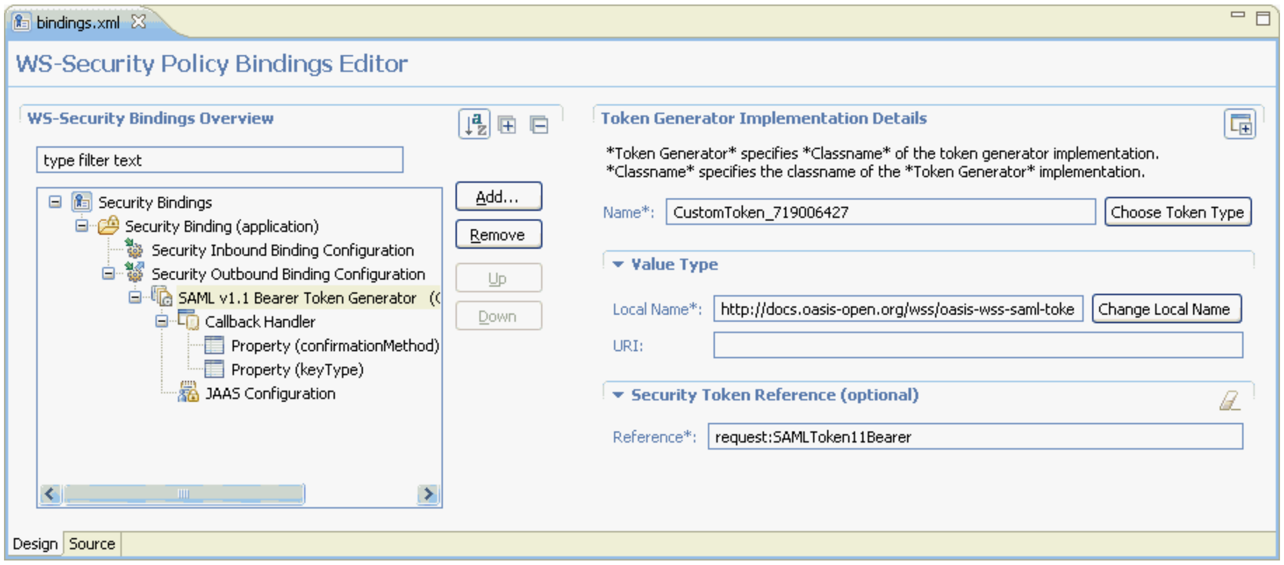
Example 46 Provider WSSecurity Binding

```
<?xml version="1.0" encoding="UTF-8"?>
<securityBindings
xmlns="http://www.ibm.com/xmlns/prod/websphere/200710/ws-securitybinding">
<securityBinding name="application">
  <securityInboundBindingConfig>
    <tokenConsumer name="con_saml11token"
classname="com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenConsumer">
      <valueType
localName="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAML
V1.1" uri="" />
```

```
      <callbackHandler
classname="com.ibm.websphere.wssecurity.callbackhandler.SAMLConsumerCallbackHan
dler">
        <properties value="Bearer" name="confirmationMethod"/>
        <properties
value="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer"
name="keyType"/>
        <properties name="trustAnySigner" value="true" />
</callbackHandler>
      <jAASConfig configName="system.wss.consume.saml"/>
      <securityTokenReference reference="request:SAMLToken11Bearer"/>
   </tokenConsumer>
  </securityInboundBindingConfig>
 </securityBinding>
</securityBindings>
```

10. Copy the complete folder `SAMLBearerClientBinding` into `RAD8TopDownBankWSClientEAR\META-INF`.

11. Open the **RAD8TopDownBankWSClientEAR\META-INF\SAMLBearerClientBinding\PolicyTypes\WSSecurity\bindings.xml** file. The WS-Security Policy Bindings Editor (Figure 71) shows that the Security Outbound Binding Configuration is configured with the SAML V1.1 Bearer Token Generator.


bindings.xml
WS-Security Policy Bindings Editor
WS-Security Bindings Overview
type filter text
Security Bindings
Security Binding (application)
Security Inbound Binding Configuration
Security Outbound Binding Configuration
SAML v1.1 Bearer Token Generator
Callback Handler
Property (confirmationMethod)
Property (keyType)
JAAS Configuration
Add...
Remove
Up
Down
Token Generator Implementation Details
Value Type
Security Token Reference (optional)
Choose Token Type
Change Local Name
Design
Source

Token Generator specifies *Classname* of the token generator implementation.
Classname specifies the classname of the *Token Generator* implementation.

Name*: CustomToken_719006427

Local Name*: http://docs.oasis-open.org/wss/oasis-wss-saml-toke

URI:

Reference*: request:SAMLToken11Bearer

Figure 71 WS-Security Policy Bindings Editor for WSSecurity client binding

The JAAS Configuration uses the `system.wss.generate.saml` JAAS Login Name.

The CallBack Handler, this time, is `com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler`, and it takes two properties (Table 7).

Table 7 SAMLGenerateCallbackHandler properties in Client Binding

Name	Value
Bearer	confirmationMethod
keyType	`http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer`

12. Example 47 shows the complete source of the client binding.

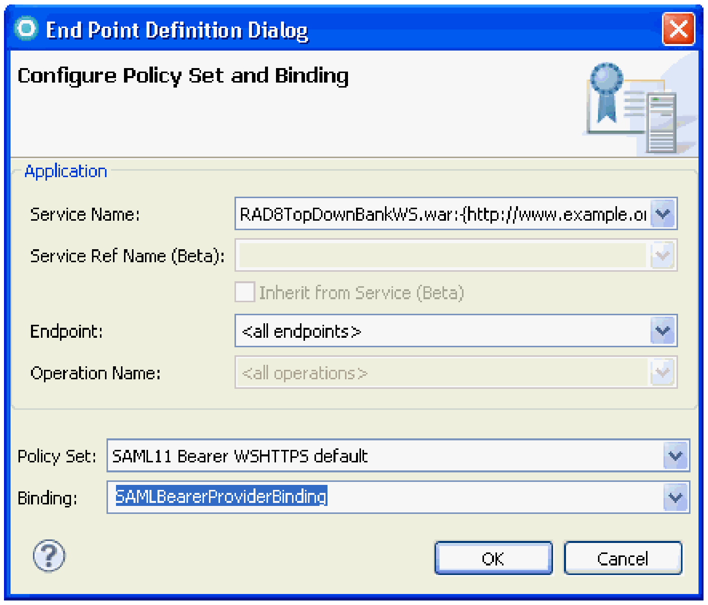
Example 47 WSSecurity client binding

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<securityBindings
xmlns="http://www.ibm.com/xmlns/prod/websphere/200710/ws-securitybinding">
    <securityBinding name="application">
           <securityOutboundBindingConfig wsuNameSpace="" wsseNameSpace="">
            <tokenGenerator
classname="com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenGenerator"
name="CustomToken_719006427">
                <valueType uri=""
localName="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAML
V1.1"/>
                <securityTokenReference reference="request:SAMLToken11Bearer"/>
                <jAASConfig configName="system.wss.generate.saml"/>
                <callbackHandler
classname="com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHan
dler">
                    <properties value="Bearer" name="confirmationMethod"/>
                    <properties
value="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer"
name="keyType"/>
                </callbackHandler>
            </tokenGenerator>
        </securityOutboundBindingConfig>
        <securityInboundBindingConfig/>
    </securityBinding>
</securityBindings>
```

We have seen how the service provider was configured to consume a SAML V1.1 Bearer Token and how the service client was configured to generate a SAML V1.1 Bearer Token.

We now associate these bindings to the corresponding policy set on the service:

1. In the Service view, right-click **RAD8TopDownBankWS**. Complete these steps:
   a. Select **Manage Policy Set Attachments** → **Server Side**.
   b. Select **Add**.
2. You see the dialog window that is shown in Figure 72. Complete these steps:
   a. For Policy set, select **SAML Bearer WSHTTPS default**.
   b. For Binding, select **SAMLBearerProviderBinding**.



Configure Policy Set and Binding

Application

Service Name: RAD8TopDownBankWS.war:{http://www.example.o

Service Ref Name (Beta):

Inherit from Service (Beta)

Endpoint: <all endpoints>

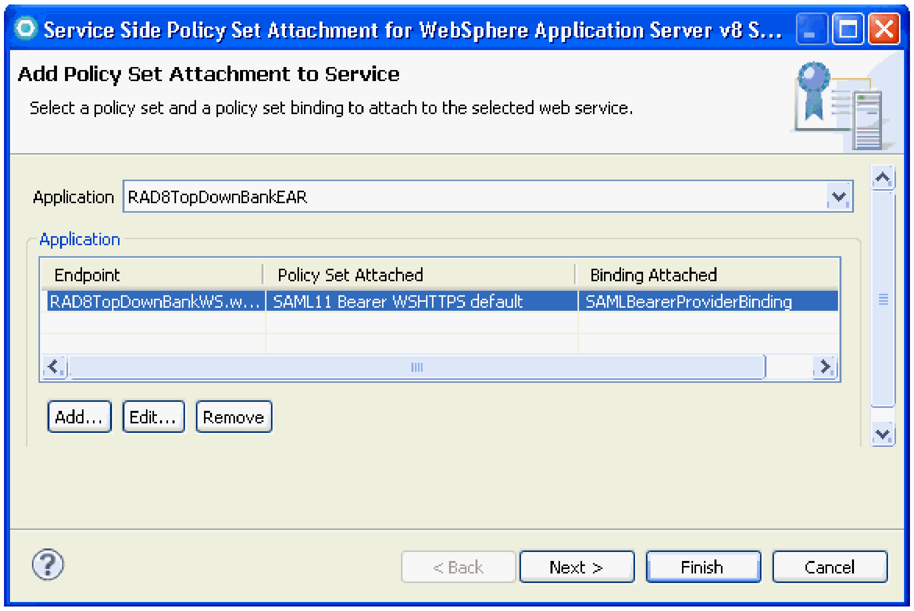
Operation Name: <all operations>

Policy Set: SAML11 Bearer WSHTTPS default

Binding: SAMLBearerProviderBinding

Figure 72 Configure Policy Set and Binding (SAMLBearerProviderBinding

3. Figure 73 shows the resulting endpoint policy set and binding.


Service Side Policy Set Attachment for WebSphere Application Server v8 S...

Add Policy Set Attachment to Service

Select a policy set and a policy set binding to attach to the selected web service.

Application RAD8TopDownBankEAR

Application

Endpoint	Policy Set Attached	Binding Attached
RAD8TopDownBankWS.w...	SAML11 Bearer WSHTTPS default	SAMLBearerProviderBinding

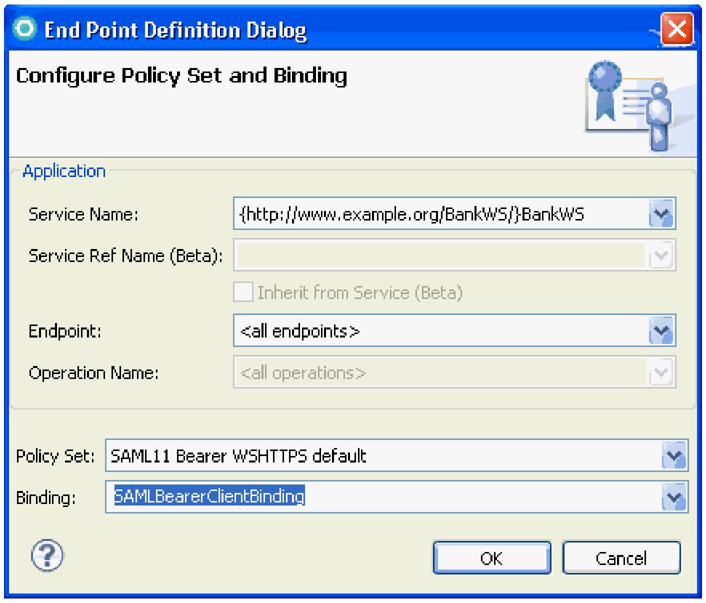
Add...
Edit...
Remove
< Back
Next >
Finish
Cancel

Figure 73 Add Policy Set Attachment to Service window

4. Select **Next**.
5. Select **Configure**.
6. Select **Share Policy Information via WSDL**.
7. Select **OK**, click **Ignore**, and click **Finish**.

We now configure the policy set and binding on the client:

1. In the Services view, right-click **RAD8TopDownBankWSClient**. Complete these steps:
   a. Select **Manage Policy Set Attachments**.
   b. Select **Add**.
2. You see the dialog window that is shown in Figure 74. Complete these steps:
   a. For Policy set, select **SAML11 Bearer WSHTTPS default**.
   b. For Binding, select **SAMLBearerClientBinding**.



End Point Definition Dialog

Configure Policy Set and Binding

Application

Service Name: {http://www.example.org/BankWS/}BankWS

Service Ref Name (Beta):

Inherit from Service (Beta)

Endpoint: <all endpoints>

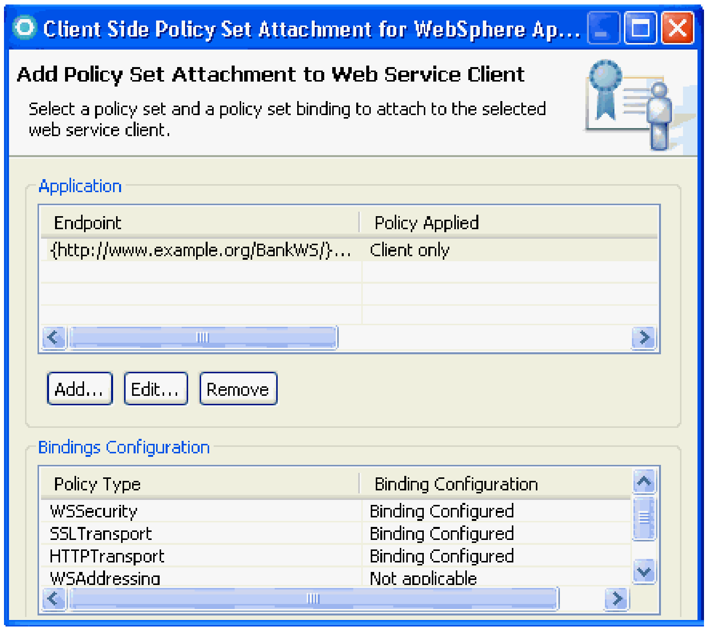
Operation Name: <all operations>

Policy Set: SAML11 Bearer WSHTTPS default

Binding: SAMLBearerClientBinding

Figure 74 Configure Policy Set and Binding (SAMLBearerClientBinding)

3. Select **OK**.
4. The Add Policy Set Attachment to Web Service Client window opens, as shown in Figure 75 on page 116. Select **Finish**.


Client Side Policy Set Attachment for WebSphere Ap...

Add Policy Set Attachment to Web Service Client

Select a policy set and a policy set binding to attach to the selected web service client.

Application

Endpoint	Policy Applied
{http://www.example.org/BankWS/}...	Client only

Bindings Configuration

Policy Type	Binding Configuration
WSSecurity	Binding Configured
SSLTransport	Binding Configured
HTTPTransport	Binding Configured
WSAddressing	Not applicable

Figure 75 Add Policy Set Attachment to Web Service Client

We have completed the configuration of the policy sets and bindings on the client.

SAML Bearer sample: Programmatic token generation

In order for the client to generate the SAML Bearer Token programmatically, add the Java file `SAMLBearerTokenSetup.java`, as shown in Example 48, to the `src` folder of the client project `RAD8TopDownBankWSClient`, in the package `org.example.bankws.saml`.

Example 48 SAMLBearerTokenSetup.java

```
package org.example.bankws.saml;

import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
import com.ibm.wsspi.wssecurity.saml.config.CredentialConfig;
import com.ibm.wsspi.wssecurity.saml.config.ProviderConfig;
import com.ibm.wsspi.wssecurity.saml.config.RequesterConfig;

public class SAMLBearerTokenSetup {

   /**
    * This method generates an instance of a version 1.1 SAML Bearer token
    */
   public static SAMLToken generateSAMLToken() {
      try {

         // Create a SAMLTokenFactory instance for a version 1.1 SAML token
         SAMLTokenFactory samlFactory =
**SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);**

         // OR: Create a SAMLTokenFactory instance for a version 2.0 SAML token
         //SAMLTokenFactory samlFactory =
SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

```
         // Create the RequesterConfig instance for the bearer token
         RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();
         ProviderConfig samlIssuerCfg = 
samlFactory.newDefaultProviderConfig(null);

         CredentialConfig cred = samlFactory.newCredentialConfig();

         // Create the SAML Token  using the SAML Token Factory
         SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, 
samlIssuerCfg);
         return samlToken;
      }
      catch (Exception e) {
         e.printStackTrace();
         return null;
      }
   }
}
```

Edit the file `BankWSSoapProxy.java` by adding the method `generateAndAttachSAMLBearerToken()`, as shown in Example 49, and call this method from both constructors.

Example 49 Generating and adding the SAML Token in BankWSSoapProxy.java

```
public void generateAndAttachSAMLBearerToken(){
      // Generate a version 1.1 SAML Bearer Token (self issuance)
      SAMLToken _samlToken = SAMLBearerTokenSetup.generateSAMLToken();

      // attaching the SAML Bearer Token to the Request Context.

((BindingProvider)_getDescriptor().getProxy()).getRequestContext().put(SamlConstan
ts.SAMLTOKEN_IN_MESSAGECONTEXT, _samlToken);
      System.out.println("$$ Generated a SAML Token  $$");
      System.out.println("SAML Token id is : " + _samlToken.getSamlID());
      System.out.println("Attached to the Request Context as property " + 
SamlConstants.SAMLTOKEN_IN_MESSAGECONTEXT);
    }
    public BankWSSOAPProxy() {
        _descriptor = new Descriptor();
        _descriptor.setMTOMEnabled(true);
        generateAndAttachSAMLBearerToken();
    }

    public BankWSSOAPProxy(URL wsdlLocation, QName serviceName) {
        _descriptor = new Descriptor(wsdlLocation, serviceName);
        _descriptor.setMTOMEnabled(true);
        generateAndAttachSAMLBearerToken();
    }
```

This point terminates the setup of the sample.

SAML Bearer sample: Testing

Perform these steps to test the sample:

1. Add both **RAD8TopDownBankWSClientEAR** and **RAD8TopDownBankWSEAR** to the server.
2. In the Enterprise Explorer view, right-click **RAD8TopDownBankWSClient\WebContent\sampleBankWSSOAPProxy\TestClient.jsp**.
3. Select **Run as** → **Run on Server**.
4. In the bottom pane of the Test Client that is opened in the browser, change the Endpoint so that it uses https. If the initial Endpoint was `http://localohost:9080/RAD8TopDownBankWS/BankWS`, it becomes `https://localhost:9443/RAD8TopDownBankWS/BankWS`. (If you have generated additional profiles with the recommended ports, both port numbers are typically increased by the same number of units.)
5. Select **Update**.
6. Select the method **getAccount**.
7. Enter any number in the accountId field.
8. Select **Invoke**.
9. You see the following results:

```
returnp:

    id: <accountId>
    balance: 1000
```

In the console, you see output indicating that the client generated the SAML Bearer Token (Example 50).

Example 50 Console showing the creation of the SAML Token

```
00000023 servlet       I com.ibm.ws.webcontainer.servlet.ServletWrapper init
SRVE0242I: [RAD8TopDownBankWSClientEAR] [/RAD8TopDownBankWSClient]
[/sampleBankWSSOAPProxy/Input.jsp]: Initialization successful.
00000024 SystemOut     O Retrieving document at
'file:/C:/workspaces/WebServices/RAD8TopDownBankWSClient/WebContent/WEB-INF/wsdl/'
.
00000024 SystemOut     O Retrieving schema at 'BankWS_schema1.xsd', relative to
'file:/C:/workspaces/WebServices/RAD8TopDownBankWSClient/WebContent/WEB-INF/wsdl/'
.
00000024 SystemOut     O $$ Generated a SAML Token  $$
00000024 SystemOut     O SAML Token id is : _ECAE899D4F56A343AC1288118909489
00000024 SystemOut     O Attached to the Request Context as property
com.ibm.wsspi.wssecurity.saml.put.SamlToken
```

If you want to see the actual SOAP message, you cannot use the TCP/IP Monitor, because the message is transmitted using HTTPS. The Generic Service Client supports HTTPS, but it does not use the modified proxy client code to generate the SAML Bearer Token. You can, however, configure tracing in WebSphere Application Server that allows you to see how the server interprets the message:

1. Right-click the server in the Server view.
2. Select **Administration** → **Run Administrative Console**.

3. Select **Troubleshooting → Logs and Trace**.
4. Select **server1**.
5. Select **Diagnostic Trace**.
6. Select **Change Log level details**.
7. Right-click **com.ibm.ws.wssecurity.saml**.
8. Select **All Messages and traces**.
9. Select **OK**, which results in the following trace string:

   `*=info: com.ibm.ws.wssecurity.saml.*=all.`

10. Select **Save**.
11. Log out of the administrative console.
12. Restart WebSphere Application Server for the changes to take effect.
13. Perform the same test as described previously.
14. Open the trace file, which, by default, is located in `<WAS_HOME>\profiles\<profile_name>\logs\server1\trace.log`.
15. You see entries, as shown in Example 51. These entries were manually formatted.

Example 51 Trace including com.ibm.ws.wssecurity.saml.=all*

```
00000017 EnvelopedSign 3   ResourceShower logs
verify-#_DD551E9C7189EE6A931288120072735:
00000017 EnvelopedSign 3
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
AssertionID="_DD551E9C7189EE6A931288120072735"
IssueInstant="2010-10-26T19:07:52.734Z"
Issuer="WebSphere" MajorVersion="1" MinorVersion="1">
  <saml:Conditions NotBefore="2010-10-26T19:07:52.750Z"
        NotOnOrAfter="2010-10-26T20:07:52.750Z">
  </saml:Conditions>
  <saml:AttributeStatement>
    <saml:Subject>
      <saml:NameIdentifier>
      </saml:NameIdentifier>
      <saml:SubjectConfirmation>
        <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer
       </saml:ConfirmationMethod>
      </saml:SubjectConfirmation>
    </saml:Subject>
  </saml:AttributeStatement>
</saml:Assertion>
00000017 EnvelopedSign 3   ResourceShower logs verify-SignedInfo:
00000017 EnvelopedSign 3
<ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-excc14n#">
  </ds:CanonicalizationMethod>
  <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
  </ds:SignatureMethod>
  <ds:Reference URI="#_DD551E9C7189EE6A931288120072735">
    <ds:Transforms>
      <ds:Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature">
      </ds:Transform>
      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      </ds:Transform>
    </ds:Transforms>
    <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
    </ds:DigestMethod>
    <ds:DigestValue>crkHuGw2LI4ZXeniAdyh9ggJ5sA=</ds:DigestValue>
  </ds:Reference>
</ds:SignedInfo>
```

You can get the complete trace log at this location:

`4884codesolution\webservices\SAMLTrace.log`

You can get the SAML Secured project in this file:

`4884codesolution\webservices\RAD8TopDownWebServiceSAML.zip`

More information

For more information about web services, see the following resources:

- For information about JAX-WS, Reliable Messaging, Secure Conversation, policy sets, and RSP profiles, see these publications:
  - *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618
  - *IBM WebSphere Application Server V7.0 Web Services Guide*, SG24-7758
- For JAX-RPC web services tools that ship with Rational Application Developer V7.0, see the *Rational Application Developer V7 Programming Guide*, SG24-7501.
- IBM developerWorks® section about SOA and web services

  http://www.ibm.com/developerworks/webservices
- List of current and emerging web services standards on developerWorks (under **SOA and Web services** → **Standards**)

  http://www.ibm.com/developerworks/webservices/standards/
- The JAX-WS specification

  http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html
- The JAXB specification

  http://jcp.org/en/jsr/detail?id=222
- The MTOM specification

  http://www.w3.org/TR/soap12-mtom/
- JAX-WS annotations:
  - https://jax-ws.dev.java.net/jax-ws-ea3/docs/annotations.html
  - http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_jaxwsannotations.html
- The WS-Policy specification

  http://www.w3.org/Submission/WS-Policy/
- The WS-MetadataExchange specification

  http://www.ibm.com/developerworks/webservices/library/specification/ws-mex/

- JAX-RS resources and examples:
  - `http://www.ibm.com/developerworks/web/library/wa-apachewink1/`
  - `http://www.ibm.com/developerworks/webservices/library/ws-restful/`
  - `http://www.ibm.com/developerworks/web/library/wa-datawebapp/`
- SAML resources and examples:
  - `http://www.ibm.com/developerworks/websphere/techjournal/1004_chao/1004_chao.html`
  - `https://www.ibm.com/developerworks/wikis/download/attachments/116424904/Introduction+to+SAML+and+support+in+7.0.0.7.pdf`
  - `http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/cwbs_samloverview.html`

Locating the web material

The web material that is associated with this paper is available in softcopy on the Internet from the IBM Redbooks web server. Enter the following URL in a web browser and then download the two ZIP files:

`ftp://www.redbooks.ibm.com/redbooks/REDP4884`

Alternatively, you can go to the IBM Redbooks website:

`http://www.ibm.com/redbooks`

Accessing the web material

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks publication form number, REDP-4884.

Additional information: For more information about the additional material, refer to *Rational Application Developer for WebSphere Software V8 Programming Guide*, SG24-7835.

The additional web material that accompanies this paper includes the following files:

File name	*Description*
`4884code.zip`	Compressed file that contains sample code
`4884codesolution.zip`	Compressed file that contains solution interchange files

System requirements for downloading the web material

We recommend the following system configuration:

Hard disk space:	20 GB minimum
Operating system:	Microsoft Windows or Linux
Processor:	2 GHz
Memory:	2 GB

The team who wrote this paper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Martin Keen is a Consulting IT Specialist at the ITSO, Raleigh Center. He writes extensively about WebSphere products and service-oriented architecture (SOA). He also teaches IBM classes worldwide about WebSphere, SOA, and enterprise service bus (ESB). Before joining the ITSO, Martin worked in the EMEA WebSphere Lab Services team in Hursley, U.K. Martin holds a Bachelors degree in Computer Studies from Southampton Institute of Higher Education.

Rafael Coutinho is an IBM Advisory Software Engineer working for Software Group in the Brazil Software Development Lab. His professional expertise covers many technology areas ranging from embedded to platform-based solutions. He is currently working on IBM Maximo® Spatial, which is the geographic information system (GIS) add-on of IBM Maximo Enterprise Asset Management (EAM). He is a certified Java enterprise architect and Accredited IT Specialist, specialized in high-performance distributed applications on corporate and financial projects.

Rafael is a computer engineer graduate from the State University of Campinas (Unicamp), Brazil, and has a degree in Information Technologies from the Centrale Lyon (ECL), France.

Sylvi Lippmann is a Software IT Specialist in the GBS Financial Solutions team in Germany. She has over seven years of experience as a Software Engineer, Technical Team Leader, Architect, and Customer Support representative. She is experienced in the draft, design, and realization of object-oriented software systems, in particular, the development of Java EE-based web applications, with a priority in the surrounding field of the WebSphere product family. She holds a degree in Business Informatic Engineering.

Salvatore Sollami is a Software IT Specialist in the Rational brand team in Italy. He has been working at IBM with particular interest in the change and configuration area and web application security. He also has experience in the Agile Development Process and Software Engineering. Before joining IBM, Salvatore worked as a researcher for Process Optimization Algorithmic, Mobile Agent Communication, and IT Economics impact. He developed the return on investment (ROI) SOA investment calculation tool. He holds the “Laurea” (M.S.) degree in Computer Engineering from the University of Palermo. In cooperation with IBM, he received an M.B.A. from the MIP - School of Management - polytechnic of Milan.

Sundaragopal Venkatraman is a Technical Consultant at the IBM India Software Lab. He has over 11 years of experience as an Architect and Lead working on web technologies, client server, distributed applications, and IBM System z®. He works on the WebSphere stack on process integration, messaging, and the SOA space. In addition to handling training on WebSphere, he also gives back to the technical community by lecturing at WebSphere technical conferences and other technical forums.

Steve Baber has been working in the Computer Industry since the late 1980s. He has over 15 years of experience within IBM, first as a consultant to IBM and then as an employee. Steve has supported several industries during his time at IBM, including health care, telephony, and banking and currently supports the IBM Global Finance account as a Team Lead for the Global Contract Management project.

Henry Cui works as an independent consultant through his own company, Kaka Software Solution. He provides consulting services to large financial institutions in Canada. Before this work, Henry worked with the IBM Rational services and support team for eight years, where he helped many clients resolve design, development, and migration issues with Java EE

development. His areas of expertise include developing Java EE applications with Rational Application Developer tools and administering WebSphere Application Server servers, security, SOA, and web services. Henry is a frequent contributor of IBM developerWorks articles. He also co-authored five IBM Redbooks publications. Henry holds a degree in Computer Science from York University.

Craig Fleming is a Solution Architect who works for IBM Global Business Services® in Auckland, New Zealand. He has worked for the last 15 years leading and delivering software projects for large enterprises as a solution developer and architect. His area of expertise is in designing and developing middleware solutions, mainly with WebSphere technologies. He has worked in several industries, including Airlines, Insurance, Retail, and Local Government. Craig holds a Bachelor of Science (Honors) in Computer Science from Otago University in New Zealand.

Now you can become a published author, too!

Stay connected to IBM Redbooks publications

Notices

This document REDP-4884-00 was created or updated on July 20, 2012.

Trademarks