

# Use the Eclipse SOA Tools Platform plug-in to build and deploy a Web service

## The Eclipse plug-in for easy Web services development

Skill Level: Intermediate

[Tyler Anderson](#)  
Freelance Writer  
Backstop Media

27 Mar 2007

Work on the Eclipse SOA Tools Platform (STP) plug-in is well under way, and we should expect lots of new features to aid Service-Oriented Architecture (SOA) deployment in the coming months. The Eclipse STP became the ninth top-level project for the Eclipse organization in 2005, and since then, development has come a long way. With the Eclipse STP plug-in, you can go from a Java™ interface, annotate it with Web service-related properties, create a Web Services Description Language (WSDL) for the Web service, generate and code Java stubs you can then compile into a WAR file, and deploy on your favorite Web server. This tutorial shows you how to accomplish all this using the Eclipse STP plug-in.

## Section 1. Before you start

What is SOA? SOA is defined as "an IT architectural style that supports the transformation of a business into a set of linked services, or repeatable business tasks, that can be accessed when needed over a network."

Why use Eclipse STP? Its aim is "to build frameworks and exemplary extensible tools that enable the design, configuration, assembly, deployment, monitoring, and management of software designed around a SOA."

## About this tutorial

This tutorial is for Service-Oriented Architecture (SOA) and Web services developers who want to learn to use the Eclipse SOA Tools Project (STP) plug-in and see for themselves how it can simplify their SOA development.

This tutorial shows the advantages and simplicity of using the Eclipse STP plug-in to build a Web service. You will learn how to do the following with the STP plug-in:

- Create a project
- Develop a Java interface
- Annotate the interface with Web services properties
- Generate WSDL
- Generate Java code from WSDL
- Add implementation code to your Web service
- Compile a WAR file of your Web service
- Deploy and test the Web service

## System requirements

This tutorial relies on several Eclipse and Apache technologies for development with the STP plug-in:

### Eclipse

Eclipse is the platform upon which the Eclipse SOA Tools plug-in runs. Download [Eclipse V3.2](#) from Eclipse Foundation.

### Eclipse STP prerequisite plug-ins

Before you install the STP plug-in, you need to install several prerequisite plug-ins. The version numbers and links to download them are listed at [Eclipse.org](#) for the STP version you specify. At the time this was written, the [latest stable build](#) was 1 Feb 2007. The plug-ins you need to install and download and install are shown under "Requirements." You should already have Eclipse V3.2 installed. Prerequisite plug-ins:

- Eclipse EMF-SDO-XSD SDK
- Eclipse GEF SDK
- Eclipse JEM SDK

- [Eclipse WTP SDK](#)
- [Eclipse GMF Runtime](#)

### **Eclipse STP Core SDK**

Now that you've got the prerequisite plug-ins, you can download the STP Core SDK plug-in, which you'll see under "SOA Tools Platform" on the STP download page.

### **Eclipse STP ServiceCreation**

In addition to the STP Core SDK, you need to download the STP ServiceCreation plug-in, which you'll see under the link from which you downloaded the STP Core SDK.

### **Eclipse STP SOAS**

In addition to the STP Core SDK, you need to download the STP SOAS plug-in, which you'll see under the link from which you downloaded the STP Core SDK and STP ServiceCreation.

### **Eclipse DTP SDK**

You need to use the Eclipse DTP to aid deployment on Web servers other than the stand-alone Web server bundled inside Eclipse.

### **Eclipse Apache CXF plug-in**

The STP plug-in uses Apache CXF its JAX-WS implementation, among other things.

### **Apache CXF Runtime**

Once you've got everything installed, you need to tell Eclipse where to find the Apache CXF Runtime.

### **Apache Tomcat**

You'll do a final test by deploying the Web service on Apache Tomcat. Download the latest Apache Tomcat V5.5.

You'll learn more about installation details before you begin coding later in the tutorial.

---

## Section 2. Introduction

SOA is becoming more popular as companies and applications become increasingly

Web-based. The Eclipse STP plug-in was designed to simplify the deployment of Web services and your SOA. This section introduces you to the STP plug-in and the application you'll create in this tutorial.

## Why the STP?

An SOA comprises several Web services that interact and work together to make a whole -- in contrast to a single monolithic application. SOA, on the other hand, is easier to maintain and deploy since it's constructed with one Web service, one building block at a time. That's what the STP can help you do: build and deploy each individual Web service.

If you've ever tried creating a Web service, you know how challenging it can be. You usually end up with many files, but you only need to change one or two of them. Not only that but the command-line tools can be difficult for beginners to master.

The STP plug-in conquers this by bringing the entire flow into a slick GUI-based environment: Eclipse. The STP plug-in makes SOA simpler because literally, as you'll see in this tutorial, anyone can build and deploy Web services with it. It's so simple you'll wonder why it hadn't been done sooner.

## Benefits of the STP plug-in

A major benefit of using the STP plug-in is derived from the Apache CXF, a Java API for XML Web services (JAX-WS) implementation, which the STP plug-in uses. Apache CXF simplifies Web service creation by allowing you to define the intricacies of your Web service using Java annotations, rather than a plethora of options.

As an additional advantage, you can recreate your Web service exactly the same way you did the first time without having to save any options you used. This is because those options are built into the Java interface using the Java annotations.

## Application overview

In this tutorial, you're going to create a Web service that acts as a scientific calculator, which performs the following functions:

- `float add(float, float)`
- `float square(float)`
- `float divide(float, float)`
- `float subtract(float, float)`

- `float multiply(float, float)`
- `float squareRoot(float, float)`
- `float inverse(float)`
- `float subtractUnary(float)`

First, we create a Java interface that defines each of the above methods, then annotate them with Java Web services properties. Then we generate WSDL from the interface, and generate client and service stubs from the WSDL. Last, we define the functionality of a Web service and create a client. In the end, we will have a fully functional Web service we can test with using the client.

---

## Section 3. Installing and creating an STP project

With so many extra plug-ins and runtimes, the STP plug-in isn't a trivial install. This section shows the worry-free way to install it, complete with all the plug-ins and the CXF runtime.

### Installing Eclipse and needed plug-ins

Once you've downloaded all the necessary plug-ins, you're ready to begin installing them. Place everything you've downloaded in the same directory and:

1. Unzip the Eclipse V3.2 package you downloaded. You should see a directory named *eclipse*. The Eclipse binary is located at *eclipse/eclipse.exe*.
2. Unzip the DTP SDK, EMF-SDO-XSD SDK, GEF SDK, GMF Runtime, JEM SDK, WTP SDK, STP Core SDK, STP ServiceCreation, and the STP SOAS plug-ins. You might be asked to overwrite any files, and it's OK to click **Yes to all**.
3. Unzip the Apache CXF plug-in you downloaded. A directory named *plug-ins* will appear. Move *plug-ins* into the *eclipse* directory. It will ask you to replace existing files within the plug-ins directory, and it's OK to click **Yes to all**.

That completes the installation on the Eclipse side. Next, we install the CXF Runtime.

## Installing the Apache CXF Runtime

You should have the CXF Runtime downloaded. Install this by unzipping it and make note of the directory you've installed it at (C:\apps\cxf-2.0-incubator-RC-SNAPSHOT for this tutorial). You'll need it when you create your first project.

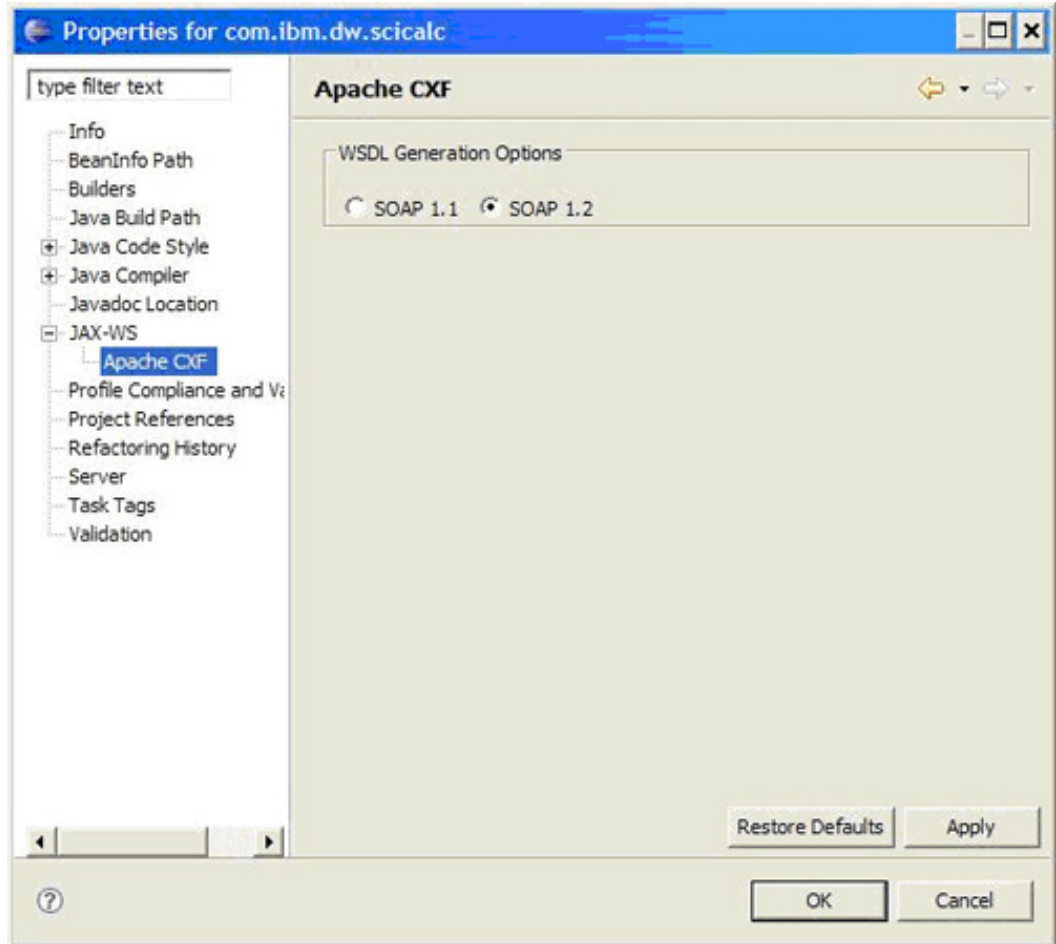
## Creating your first STP project

You're ready to open up Eclipse. When opening Eclipse, it'll ask you for a workbench, and you can put yours anyplace, and click **Enter**. When Eclipse finishes loading:

1. Click **File > New Project**
2. Select **SOA Tools > JAX-WS Java First Project**
3. Click **Next**
4. Enter a project name: `com.ibm.dw.scicalc`
5. Click **Finish**

Now that the project has been created, we can set project-specific properties:

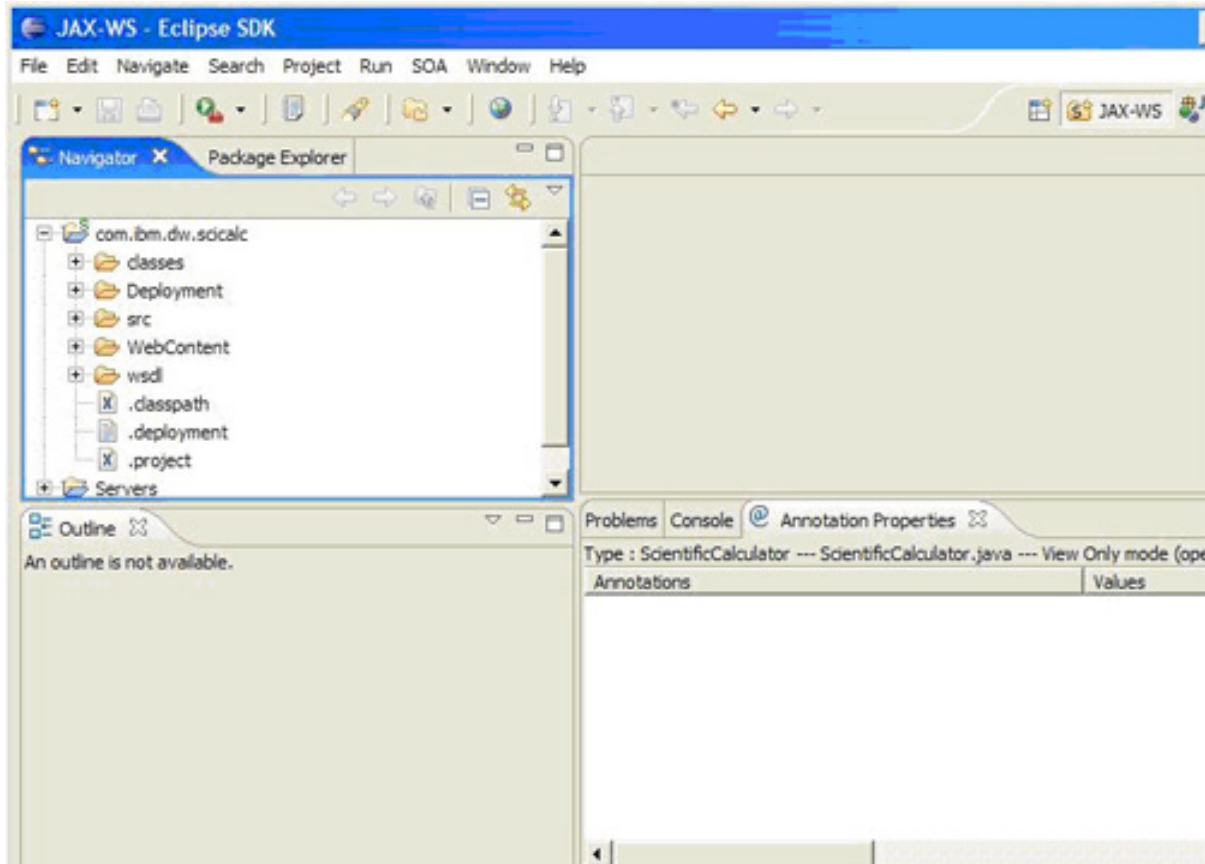
1. Go to **Project > Properties**
  2. Click on **JAX-WS**
  3. Enter the directory you installed CXF to in the Runtime Installation box (C:\apps\cxf-2.0-incubator-RC-SNAPSHOT for this tutorial)
  4. Click **Apply**
  5. Now click on the child of JAX-WS, **Apache CXF**
  6. In the WSDL Generation Options pane, select SOAP 1.2, as shown in Figure 1
- Figure 1. Selecting SOAP V1.2 as a WSDL-generation option**



7. Click **Apply**, then **OK**
8. Do the same thing for the preferences; click **Window > Preferences**
9. Click **SOA Tools > JAX-WS**
10. In the Installed runtimes pane, enter the directory you installed CXF to in the Apache CXF box (C:\apps\cxf-2.0-incubator-RC-SNAPSHOT for this tutorial)
11. Click **Apply**
12. Click on the child of JAX-WS, **Apache CXF**
13. In the WSDL Generation Options pane, select SOAP 1.2, similar to what you did in Figure 1
14. Click **Apply**, then click **OK**

The project's now ready for prime time. Before you begin coding, open the JAX-WS perspective by clicking **Window > Open Perspective > Other**. Select JAX-WS from the list and click **OK**. Your window should look much like Figure 2.

**Figure 2. Your project in the JAX-WS perspective**



Excellent! You're ready to move on and begin development of the Java interface.

---

## Section 4. Creating the Java interface

The Java interface is the starting point from which you'll make the full definition of your Web service using annotations and compile into WSDL later.

### The Java interface

Before coding a Java interface, we need to add a new one to our project. Right-click your project (the `com.ibm.dw.scicalc` folder in the Navigator window), then click **New**

> **Other**. This will take you to a list of things you can create. Select Interface (the purple I) and click **Next**. Fill in the package name using `com.ibm.dw.scicalc`, with the name being `ScientificCalculator`. Now click **Finish**.

You should now have an empty `ScientificCalculator` interface in `src/com/ibm/dw/scicalc/` directory. Open this file, and it should mostly look like the following.

```
package com.ibm.dw.scicalc;

public interface ScientificCalculator {
}
```

Next, we create methods in the Java interface that define the operations a Web service can perform.

## Defining Web service operations

As a scientific calculator-type Web service, the Web service we're creating will have the functions defined in the "Application overview" section. Code each of them in the interface, as shown below.

### Listing 1. Coding the Java interface

```
public interface ScientificCalculator {
    public float squareRoot( // value^pow
        float value,
        float pow
    );

    public float square( // value^2
        float value
    );

    public float inverse( // 1/value
        float value
    );

    public float divide( // numerator/denominator
        float numerator,
        float denominator
    );

    public float multiply( // value1*value2
        float value1,
        float value2
    );

    public float add( // value1 + value2
        float value1,
        float value2
    );

    public float subtract( // value1-value2
        float value1,
```

```
        float value2
    );

    public float subtractUnary( // -value
        float value
    );
}
```

Note that we're simply declaring methods -- the operations of the Web service we're ultimately creating. Most of them have two input parameters, and a few of them have just one, with each returning the result of the operation.

Next, we begin annotating the Java interface with Web service-specific properties.

## Annotating the class

Begin by annotating the class as a Web service. In the Outline view (lower left), click on the Java interface, `ScientificCalculator`. Click on **SOA > JAX-WS > Create Web Service**, then click **Finish**. You'll notice the following was added to your code.

### Listing 2. Annotating the Java interface as a Web service

```
...
import javax.jws.WebService;
...
@WebService(wsdlLocation = "file:/C:/Documents and
Settings/Tyler3/workspaces/STPwarCreate/com.ibm.dw.scicalc/wsdl/com/ibm
/dw/scicalc/ScientificCalculator
.wsdl", targetNamespace = "http://scicalc.dw.ibm.com/", name = "ScientificCalculator")
public interface ScientificCalculator {
...
}
```

Essentially, we have declared that our interface is now a Web service, complete with a target namespace and name. It also defines the location that our WSDL file will be created.

Manually add one last annotation to the class by adding the code below.

### Listing 3. Manually adding an annotation

```
...
import javax.jws.soap.SOAPBinding;
...
@WebService(wsdlLocation = "file:/C:/Documents and ...
...
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE,
    use = SOAPBinding.Use.LITERAL,
    style = SOAPBinding.Style.DOCUMENT)
public interface ScientificCalculator {
...
}
```

---

We declared the SOAP binding of our Web service, `DOCUMENT` style, using literals and bare parameter style.

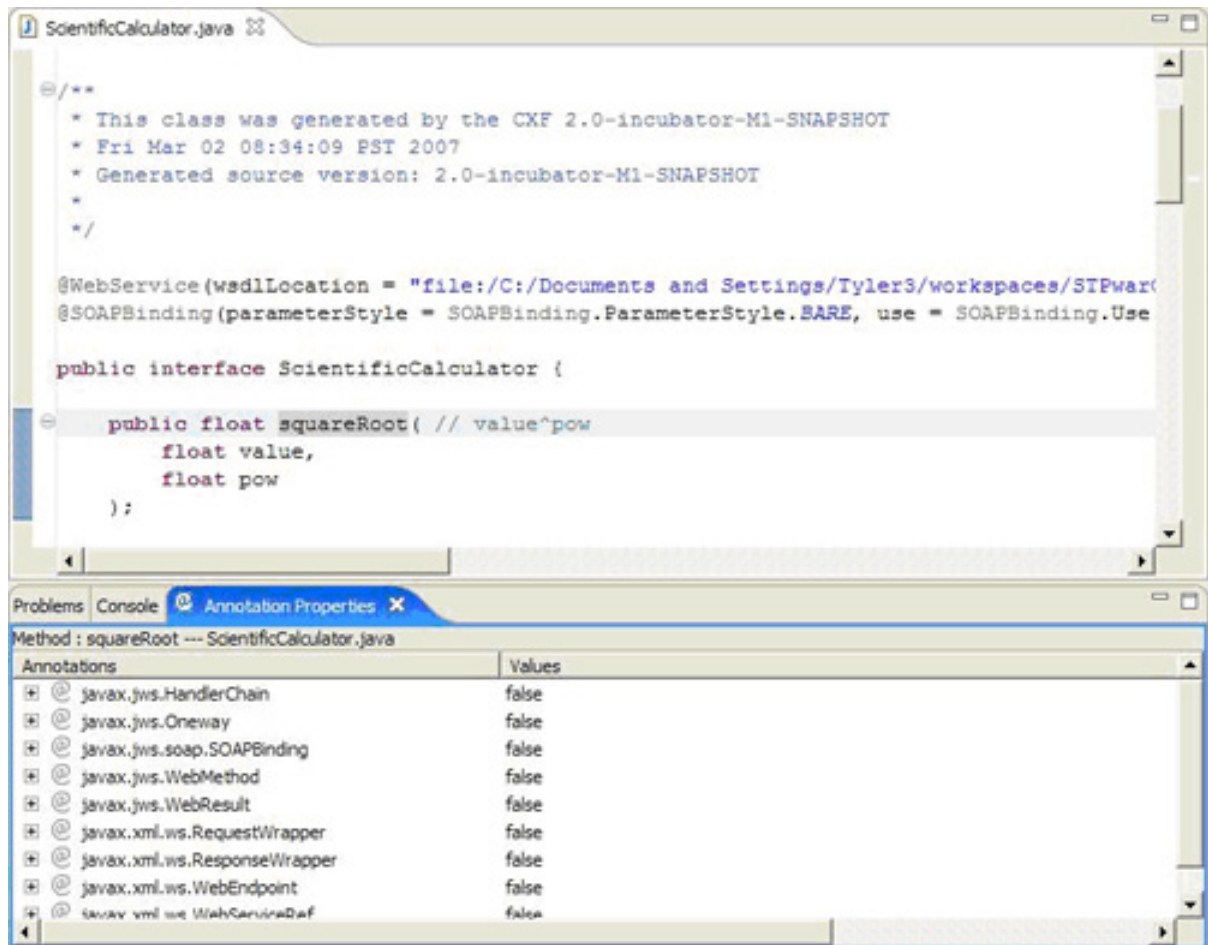
That completes the interface annotations. Next, we create annotations for the methods and the operations of your Web service.

## Annotating a method

Annotating a method is about the same as annotating a class. Begin by annotating the `squareRoot` method. Use the Annotation Properties view to do it. If you can't see it, select **Window > Show View > Other**. Now select **SOA Tools > Annotation Properties**. You should now see it in the bottom right section of your view.

To create annotations using the Annotation Properties view, put the cursor on the following line in your interface: `public float squareRoot( // value^pow.` Look in the Annotation Properties view, and you should see a list of available annotations.

### Figure 3. The Annotation Properties view



First, add the WebMethod annotation. You can see it above. Select it in the Annotation Properties view, click on the drop-down box beside it, and change false to true. The annotation should now appear.

#### Listing 4. The WebMethod annotation

```

...
import javax.jws.WebMethod;
...
@WebMethod(operationName = "squareRoot", exclude=false)
public float squareRoot( // value^pow
    float value,
    float pow
);
...

```

You have now declared the squareRoot method as an operation in your Web service. There are now just a few more annotations to go: declaring the return value and parameters of our Web service.

Select the WebResult annotation and change false to true. This adds the following to

your code.

### Listing 5. The WebResult annotation

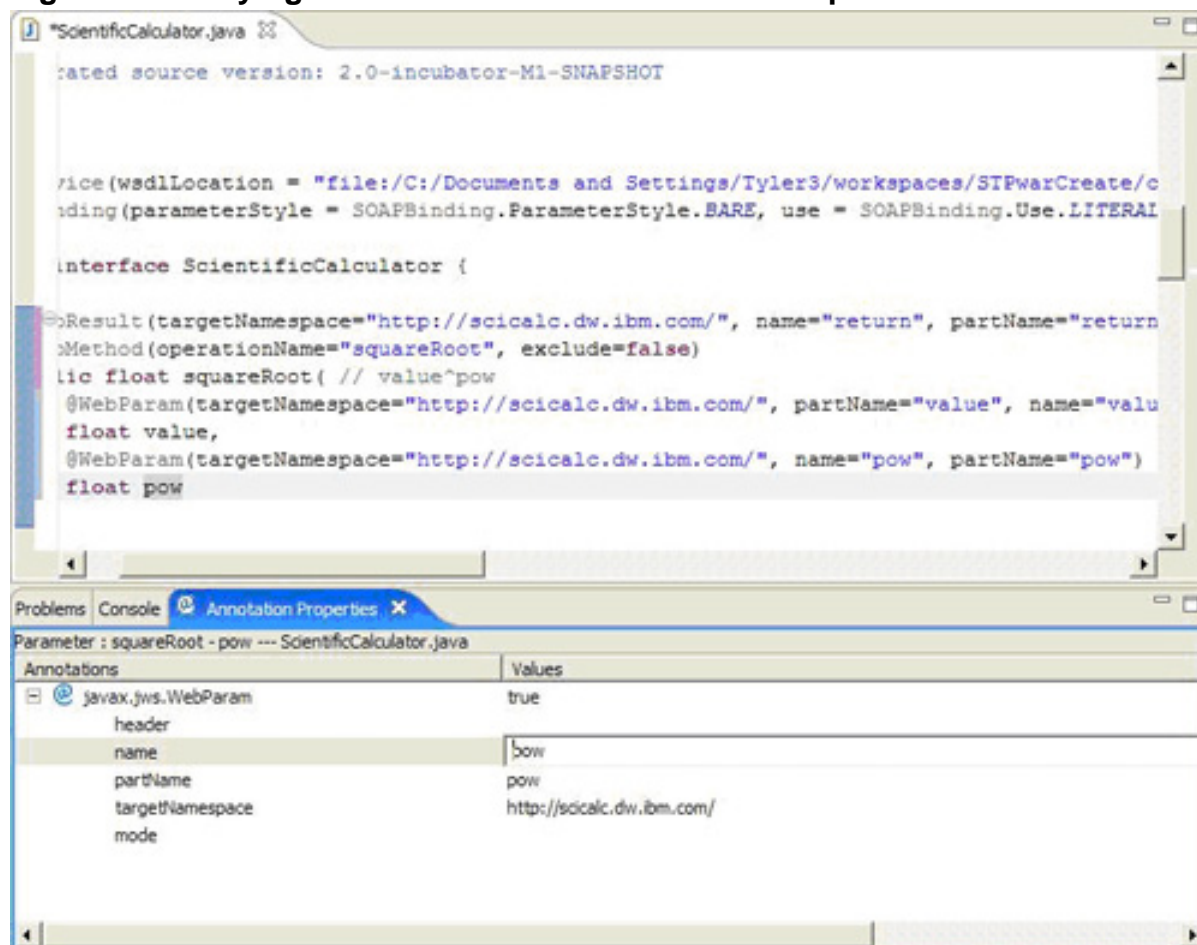
```

...
import javax.jws.WebResult;
...
    @WebResult(targetNamespace="http://scicalc.dw.ibm.com/",
               name="return", partName="return")
    @WebMethod(operationName = "squareRoot", exclude=false)
    public float squareRoot( // value^pow
        float value,
        float pow
    );
...

```

Add the `WebParam` annotations to both parameters. Select the line containing the `float value`, then select the `WebParam` annotation in the Annotation Properties box, and change `false` to `true`. Do the same for the `float pow` line. Return to the same line and modify the properties of the parameter.

Figure 4. Modifying annotations in the Annotation Properties view



For both parameters, we have defined the name and `partName` properties in the `WebParam` annotation. You can see the code that was added in Listing 7.

### Listing 6. Code added from the `WebParam` annotation properties

```
...
import javax.jws.WebParam;
...
    @WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
               partName = "return", name = "return")
    @WebMethod(operationName = "squareRoot")
    public float squareRoot( // value^pow
        @WebParam(targetNamespace="http://scicalc.dw.ibm.com",
                  partName = "value", name = "value")
        float value,
        @WebParam(targetNamespace="http://scicalc.dw.ibm.com",
                  partName = "pow", name = "pow")
        float pow
    );
...
```

You've successfully finished annotating the `squareRoot` method. Next, we annotate the rest of the methods.

## Annotating other methods

Before quickly annotating the rest of the methods, there's one more way to annotate a method you should know, which is complementary to how you annotated the Web service. Select the `square(float)` method in the Outline view (lower left) and:

- Select **SOA > Create Web Method**
- The first specifies `WebMethod`, which you can leave as is; click **Next**
- Uncheck **Add Annotation** because you don't need a `RequestWrapper`, then click **Next**
- Check **Add Annotation**, then click **Next**
- Uncheck **Add Annotation** because you don't need a `ResponseWrapper`, then click **Finish**

It should create the following annotations for you.

### Listing 7. Annotations created using create Web method

```
@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
           header=false,
           partName = "return", name = "return")
@WebMethod(operationName = "square", exclude=false)
public float square( // value^2
    float value
```

```
);
```

You can now annotate the value parameter as you did in the previous section, bringing the `square` method's code to the following.

```
public float square( // value^2
    @WebParam(partName = "value", name = "value")
    float value
);
```

That finishes the `square` method! You should now be able to annotate the rest of the methods on your own. When you're done, they should look like what's shown below.

### Listing 8. Completing the annotations

```
@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
    partName = "return", name = "return")
@WebMethod(operationName = "inverse")
public float inverse( // 1/value
    @WebParam(partName = "value", name = "value")
    float value
);

@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
    partName = "return", name = "return")
@WebMethod(operationName = "divide")
public float divide( // numerator/denominator
    @WebParam(partName = "numerator", name = "numerator")
    float numerator,
    @WebParam(partName = "denominator", name = "denominator")
    float denominator
);

@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
    partName = "return", name = "return")
@WebMethod(operationName = "multiply")
public float multiply( // value1*value2
    @WebParam(partName = "value1", name = "value1")
    float value1,
    @WebParam(partName = "value2", name = "value2")
    float value2
);

@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
    partName = "return", name = "return")
@WebMethod(operationName = "add")
public float add( // value1 + value2
    @WebParam(partName = "value1", name = "value1")
    float value1,
    @WebParam(partName = "value2", name = "value2")
    float value2
);

@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
    partName = "return", name = "return")
@WebMethod(operationName = "subtract")
public float subtract( // value1-value2
    @WebParam(partName = "value1", name = "value1")
    float value1,
```

```
    @WebParam(partName = "value2", name = "value2")
    float value2
);

@WebResult(targetNamespace = "http://scicalc.dw.ibm.com/",
           partName = "return", name = "return")
@WebMethod(operationName = "subtractUnary")
public float subtractUnary( // -value
    @WebParam(partName = "value", name = "value")
    float value
);
```

There you have it! You're ready to jump in and generate WSDL.

---

## Section 5. Generating WSDL

In this section, we go from the annotations in your Java interface to WSDL.

### Generating WSDL from the Java interface

To create the latest WSDL version of your Java interface, make sure you save the interface. This automatically creates WSDL at `wSDL/com/ibm/dw/scicalc/ScientificCalculator.wSDL` in your project.

### Taking a look at the WSDL

Open the WSDL you just learned about. Note that you don't need to change anything since it's auto-generated, except the address of the service, which you'll do in the next section. Let's examine what has been generated. Look at the WSDL for the `squareRoot` operation below (other details in the WSDL have been removed, as there isn't much difference from one to the other).

#### Listing 9. The auto-generated WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wSDL:definitions targetNamespace="http://scicalc.dw.ibm.com/"
    xmlns:tns="http://scicalc.dw.ibm.com/"
    xmlns:soap12="http://schemas.xmlsoap.org/wSDL/soap12/"
    xmlns:ns1="http://scicalc.dw.ibm.com/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
  <wSDL:types>
    <xsd:schema targetNamespace="http://scicalc.dw.ibm.com/"
        version="1.0"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```

        <xs:element name="squareRoot0" type="xs:float"/>
        <xs:element name="squareRoot1" type="xs:float"/>
    ...
</xsd:schema>
</wsdl:types>
<wsdl:message name="squareRoot">
    <wsdl:part name="value" element="tns:squareRoot0">
</wsdl:part>
    <wsdl:part name="pow" element="tns:squareRoot1">
</wsdl:part>
</wsdl:message>
<wsdl:message name="squareRootResponse">
    <wsdl:part name="return" element="tns:return">
</wsdl:part>
</wsdl:message>
...
<wsdl:portType name="ScientificCalculator">
    <wsdl:operation name="squareRoot">
        <wsdl:input name="squareRoot" message="tns:squareRoot">
</wsdl:input>
        <wsdl:output name="squareRootResponse"
            message="tns:squareRootResponse">
</wsdl:output>
    </wsdl:operation>
...
</wsdl:portType>
<wsdl:binding name="ScientificCalculatorBinding"
    type="tns:ScientificCalculator">
    <soap12:binding style="document"
        transport="http://www.w3.org/2003/05/soap/bindings/HTTP/" />
    <wsdl:operation name="squareRoot">
        <soap12:operation style="document" />
        <wsdl:input name="squareRoot">
            <soap12:body use="literal" />
        </wsdl:input>
        <wsdl:output name="squareRootResponse">
            <soap12:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
...
</wsdl:binding>
<wsdl:service name="ScientificCalculatorService">
    <wsdl:port name="ScientificCalculatorPort"
        binding="tns:ScientificCalculatorBinding">
        <soap12:address location="http://localhost/changedme" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Note the schema that shows the elements, the parameters used in the `squareRoot` operation (`squareRoot0` and `squareRoot1` are both of type `float`). Next, see the two messages -- one for the request and the other for the response -- and notice how they reference the two elements in the schema. Next is the `portType`, which defines the `squareRoot` operation, its input and output messages. The binding specifies lower-level details like the SOAP version, transport style (`document`), and whether to use literal or encoded format for the message body. Last, the service specifies the name of the Web service and the location at which you can find it. You'll modify this line in the next section.

## Setting the address of the Web service

Now that the WSDL is ready to go, you just have to make a slight change to its location: the URL where you ultimately intend to post the Web service. This tutorial places it at `http://localhost:8080/ScientificCalculator/services/ScientificCalculatorService`. Make the following change:

### Listing 10. Specifying the location of the Web service

```
...
  <wsdl:service name="ScientificCalculatorService">
    <wsdl:port name="ScientificCalculatorPort"
      binding="tns:ScientificCalculatorBinding">
      <soap12:address location="http://localhost:8080/ScientificCalculator/services/
ScientificCalculatorService"/>
    </wsdl:port>
  </wsdl:service>
...
```

When you've deployed the Web service later, the above URL is where you'll go to view the Web service.

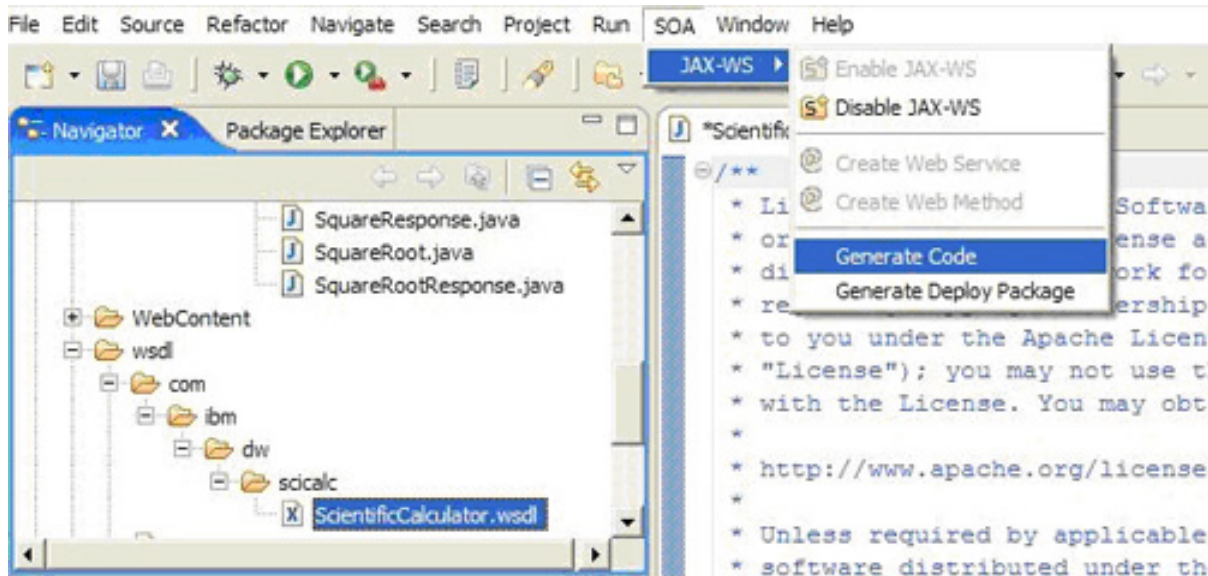
---

## Section 6. Code generation from WSDL

### Creating client and service stubs from WSDL

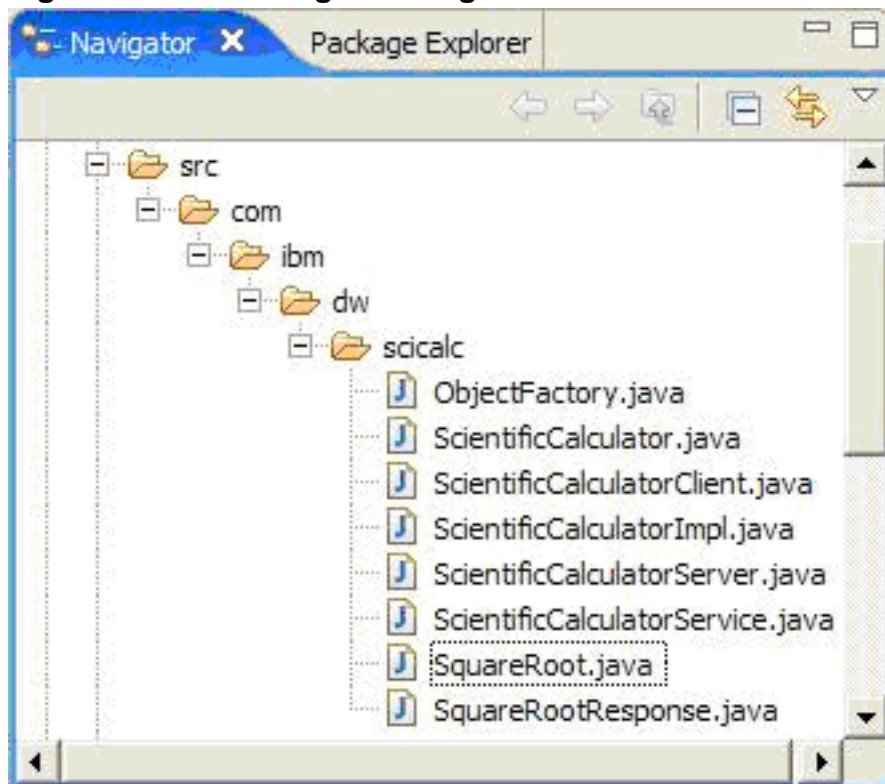
With the WSDL complete, select the WSDL file in the Navigator view, then select **SOA > Generate code**.

#### Figure 5. Generating code



On the next screen, make sure that Server and Client are selected, and click **Finish**. You should now see several new files created in the Navigator view, as shown in Figure 6.

**Figure 6. Results of generating code**



OK, that'll do it. Next, you'll define the service code.

---

## Section 7. Defining the service implementation

The service implementation is the code that gets run when something -- usually a client -- invokes an operation on the Web service. Right now, it does nothing, and so in this section, we define it and make it do something useful.

### The implementation class

The implementation class was auto-generated, and below are the methods of interest to you -- the methods that define the operations of the Web service. Write code for these methods in the next section.

#### Listing 11. Auto-generated methods of the service implementation class

```
...
public class ScientificCalculatorImpl implements ScientificCalculator {

    private static final Logger LOG =
        Logger.getLogger(ScientificCalculatorImpl.class.getPackage().getName());

    public float add(
        float value1,
        float value2
    )
    {
        LOG.info("Executing operation add");
        return 0.0f;
    }

    public float square(
        float value
    )
    {
        LOG.info("Executing operation square");
        return 0.0f;
    }

    public float divide(
        float numerator,
        float denominator
    )
    {
        LOG.info("Executing operation divide");
        return 0.0f;
    }

    public float subtract(
        float value1,
        float value2
    )
    {
        LOG.info("Executing operation subtract");
        return 0.0f;
    }
}
```

```
public float multiply(  
    float value1,  
    float value2  
)  
{  
    LOG.info("Executing operation multiply");  
    return 0.0f;  
}  
  
public float squareRoot(  
    float value,  
    float pow  
)  
{  
    LOG.info("Executing operation squareRoot");  
    return 0.0f;  
}  
  
public float inverse(  
    float value  
)  
{  
    LOG.info("Executing operation inverse");  
    return 0.0f;  
}  
  
public float subtractUnary(  
    float value  
)  
{  
    LOG.info("Executing operation subtractUnary");  
    return 0.0f;  
}  
}
```

Note that the parameters are the exact same as the ones you named in the `WebParam` annotations in your Java interface. Next you'll do the proper functions with the passed in parameters and return the correct result.

## Writing code for the Web service operations

So now you just have to return correct values for the operations. It's simple floating-point arithmetic, which you probably already know. Either way, the code for each method is shown below.

### Listing 12. Coding the Web service operations

```
    LOG.info("Executing operation add");  
    return value1 + value2;  
...  
    LOG.info("Executing operation square");  
    return (float)Math.pow((double)value, 2);  
...  
    LOG.info("Executing operation divide");  
    return numerator/denominator;  
  
    LOG.info("Executing operation subtract");  
    return value1-value2;  
...
```

```
LOG.info("Executing operation multiply");
return value1*value2;

LOG.info("Executing operation squareRoot");
return (float)Math.pow((double)value, (double)pow);
...
LOG.info("Executing operation inverse");
return (float)1/value;
...
LOG.info("Executing operation subtractUnary");
return -value;
```

Simply return the valid calculations using the values passed into the function. The next section defines code that calls each of the above Web service operations.

---

## Section 8. Defining the client code

Client code allows you to write Java code that can interact with the Web service you're creating in this tutorial, as well as other Web services. This section defines that client code.

### The client class

The auto-generated client class has simply spit out code that uses each of the operations in your Web service. Take a look at what was generated.

#### Listing 13. The auto-generated client code

```
...
ScientificCalculatorService ss = new \
ScientificCalculatorService(wsdlURL, SERVICE_NAME);
ScientificCalculator port = ss.getScientificCalculatorPort();

System.out.println("Invoking add...");
float _add_return = port.add(0.0f,0.0f);

System.out.println("Invoking square...");
float _square_return = port.square(0.0f);

System.out.println("Invoking divide...");
float _divide_return = port.divide(0.0f,0.0f);

System.out.println("Invoking subtract...");
float _subtract_return = port.subtract(0.0f,0.0f);

System.out.println("Invoking multiply...");
float _multiply_return = port.multiply(0.0f,0.0f);

System.out.println("Invoking squareRoot...");
```

```
float _squareRoot_return = port.squareRoot(0.0f,0.0f);

System.out.println("Invoking inverse...");
float _inverse_return = port.inverse(0.0f);

System.out.println("Invoking subtractUnary...");
float _subtractUnary_return = port.subtractUnary(0.0f);

System.exit(0);
}
}
```

It isn't very useful yet, but in the next section, you'll modify it to pass in useful parameters and display the results to standard output.

## Writing client code for the Web service operations

Now that you have a client, you can use it to fully test your Web service and perform regression tests if you want. Take a look at the code you'll add.

### Listing 14. Defining client code

```
...
System.out.println("Invoking add...");
float _add_return = port.add(1.2f,3.6f);
System.out.println("Result: "+_add_return);

System.out.println("Invoking square...");
float _square_return = port.square(3);
System.out.println("Result: "+_square_return);

System.out.println("Invoking divide...");
float _divide_return = port.divide(4.8f,1.2f);
System.out.println("Result: "+_divide_return);

System.out.println("Invoking subtract...");
float _subtract_return = port.subtract(2.5f,0.5f);
System.out.println("Result: "+_subtract_return);

System.out.println("Invoking multiply...");
float _multiply_return = port.multiply(1.5f,10.0f);
System.out.println("Result: "+_multiply_return);

System.out.println("Invoking squareRoot...");
float _squareRoot_return = port.squareRoot(4.0f,0.5f);
System.out.println("Result: "+_squareRoot_return);

System.out.println("Invoking inverse...");
float _inverse_return = port.inverse(2.0f);
System.out.println("Result: "+_inverse_return);

System.out.println("Invoking subtractUnary...");
float _subtractUnary_return = port.subtractUnary(6.999f);
System.out.println("Result: "+_subtractUnary_return);
...
```

Here, you pass in various parameters and display them to standard output. Thus,

when you run the client, you can see the output and know that it's working. Next, we deploy and test the Web service.

---

## Section 9. Deployment and testing

Seems like we're ready to deploy and test the Web service. In this section, we package our Web service for deployment, deploy it, and test it using Representational State Transfer (REST) and the client you've built.

### Creating a deployment package of your Web service

Before you can run the stand-alone server and test the Web service, you need to package your code in a WAR file for deployment:

- Select the WSDL file in the Navigator view
- Click **SOA > Generate Deploy Package**

That creates a deployable WAR file at WebContent/ScientificCalculator.war in your project.

Now you're good to move on and run the stand-alone Web server.

### Running the stand-alone Web server

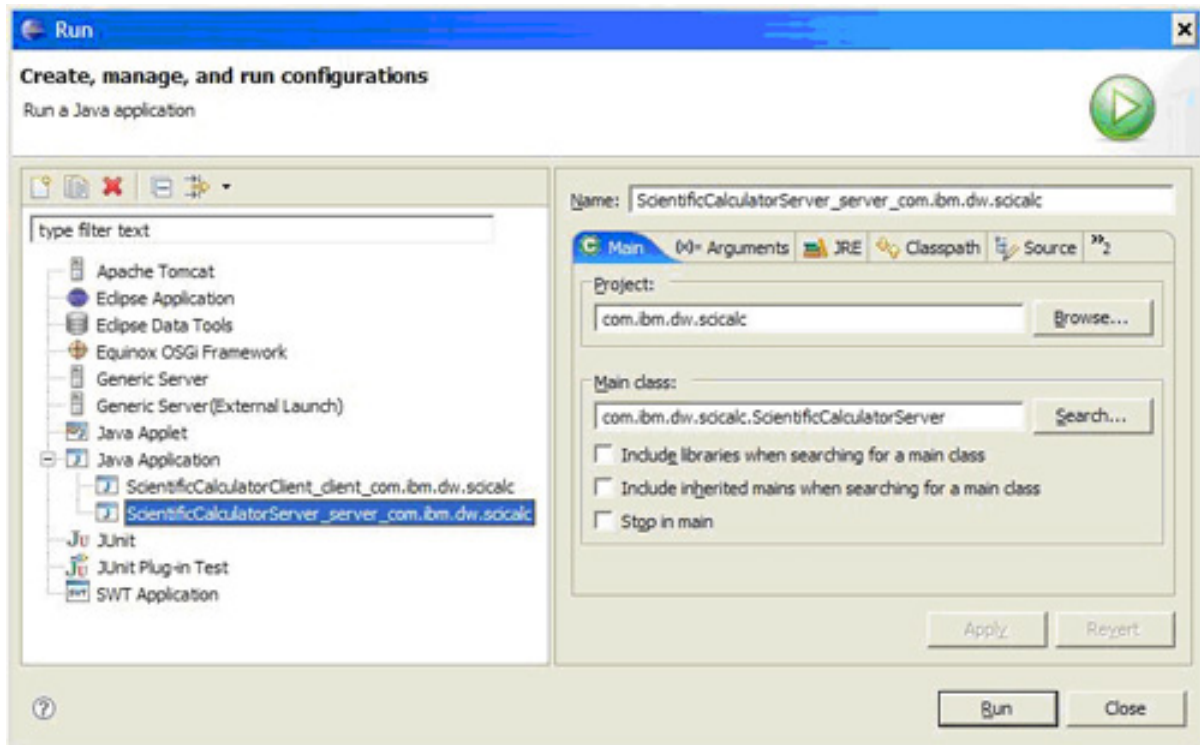
The development framework comes with a stand-alone Web server, which allows you to perform testing of your Web service within the Eclipse framework.

Note that the STP plug-in has created a couple preconfigured run configurations. To run the server:

1. Double-click the **Java** interface in the Navigator view
2. Click **Run > Run**
3. Select **Java Application > ScientificCalculatorServer\_server\_com.ibm.dw.scicalc**, as shown in Figure 7

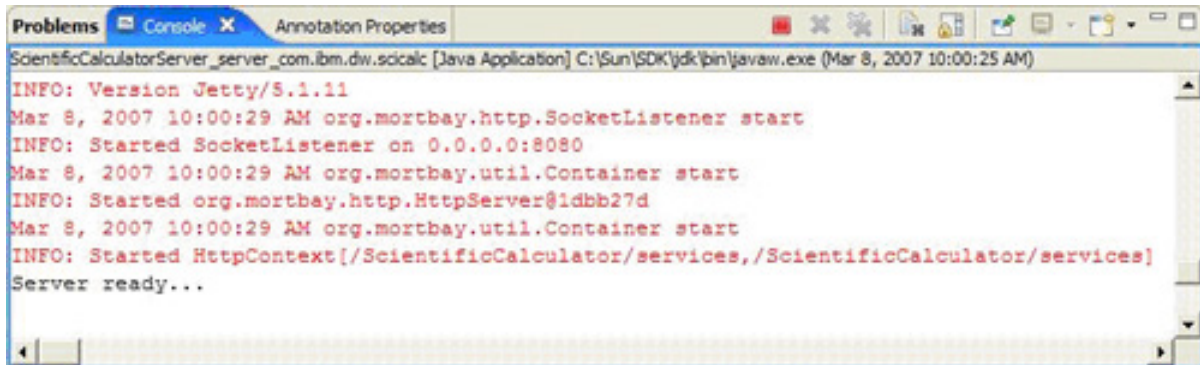
#### 4. Click Run

**Figure 7. Running the server**



The server should then run and let you know it's ready.

**Figure 8. Server ready**



Now point your browser to <http://localhost:8080/ScientificCalculator/services/ScientificCalculatorService?wsdl> and you should see the WSDL. Next, we test the Web service in a browser using REST.

## Testing using REST

With a browser pointed to the WSDL of the Web service, you can now test operations on it. Point your browser to <http://localhost:8080/ScientificCalculator/services/ScientificCalculatorService/square?square0=4.5>.

Notice that we're calling the square operation with 4.5 as the parameter. You'll get a file of type soap+xml, that will show 20.25 as the result, as shown in the SOAP message that gets returned below.

### Listing 15. SOAP message response

```
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <ns2:return xmlns:ns2="http://scicalc.dw.ibm.com/"
      xmlns="http://www.w3.org/2005/08/addressing/wsdl">
      20.25
    </ns2:return>
  </soap:Body>
</soap:Envelope>
```

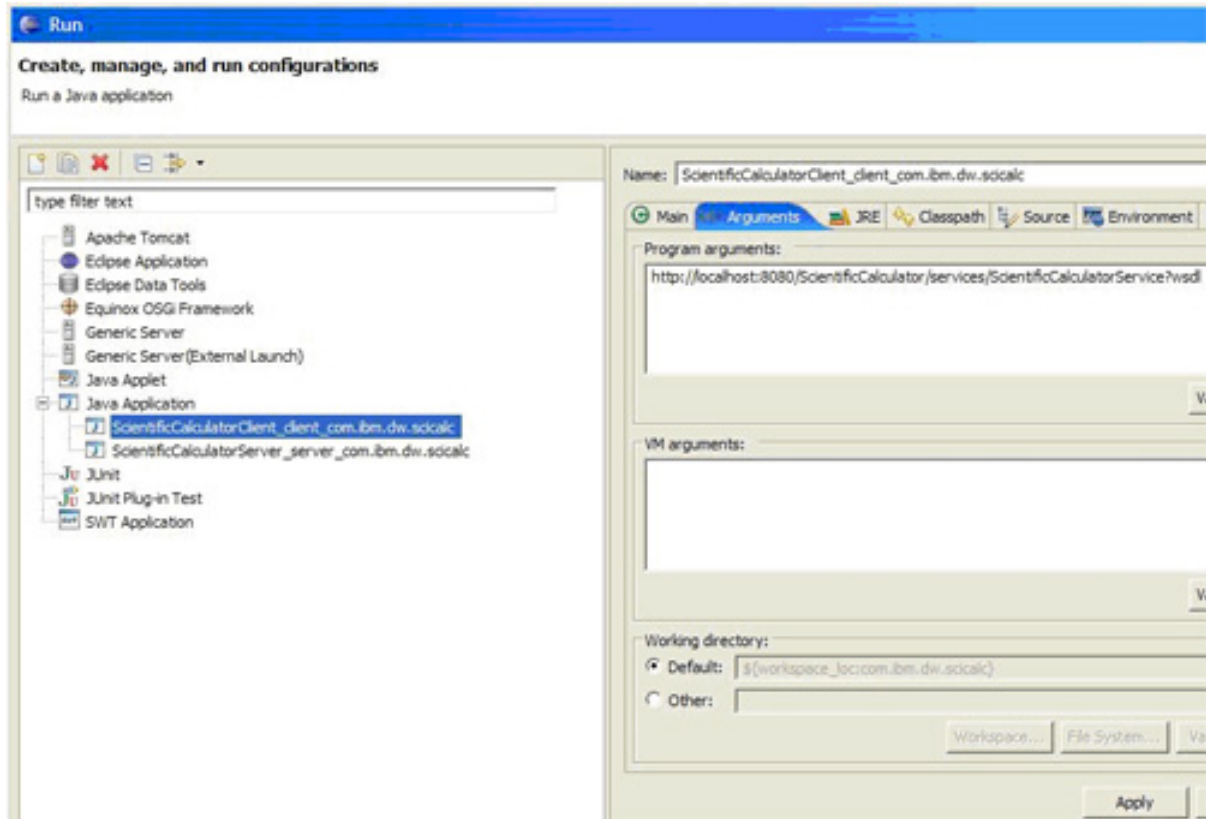
You can see the answer is 20.25. Feel free to test out the other operations using REST. Next, we do a more regressive test using your client.

## Testing by running the client code

Now that you know your service is up and functional with a REST test, you can perform a more rigorous and all-encompassing test using your client. To run the client:

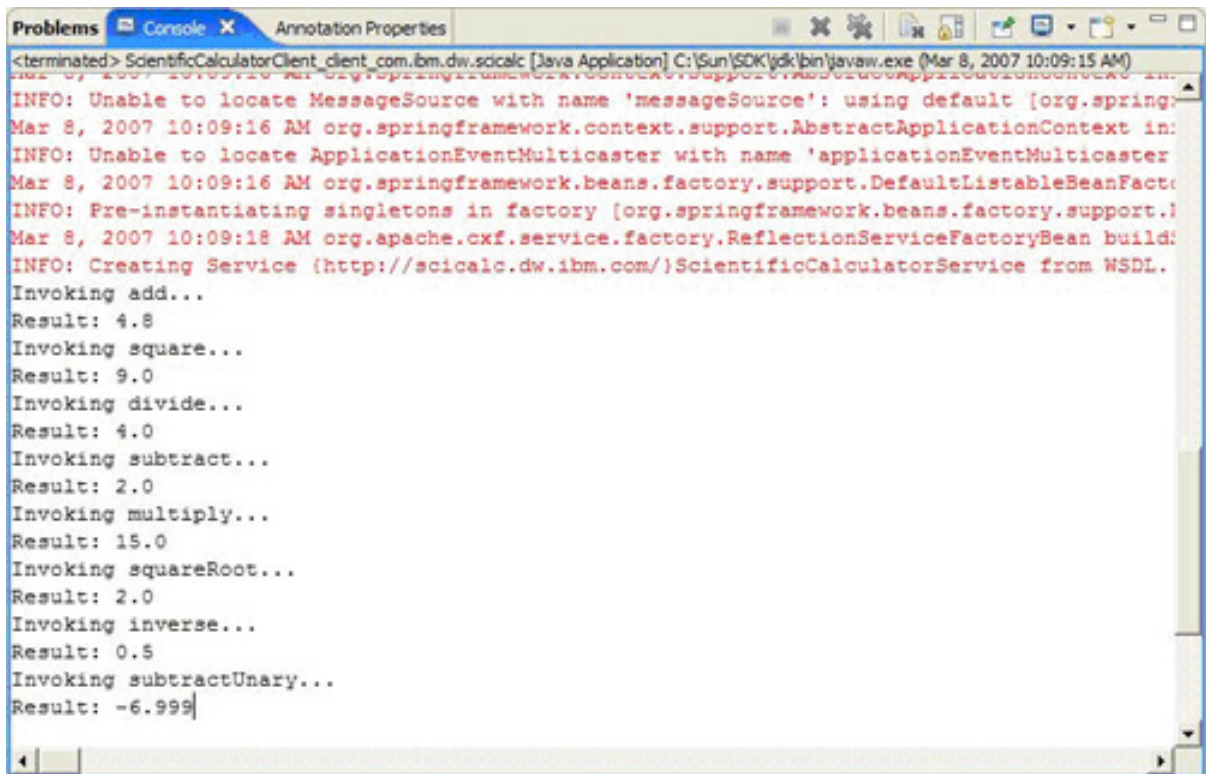
1. Double click the **Java** interface in the Navigator view
2. Click **Run > Run**
3. Select **Java Application > ScientificCalculatorClient\_client\_com.ibm.dw.scicalc**
4. Select the **Arguments** tab
5. Replace the "Program arguments" pane with <http://localhost:8080/ScientificCalculator/services/ScientificCalculatorService?wsdl>, as shown in Figure 9
6. Click **Run**

**Figure 9. Configuring the client**



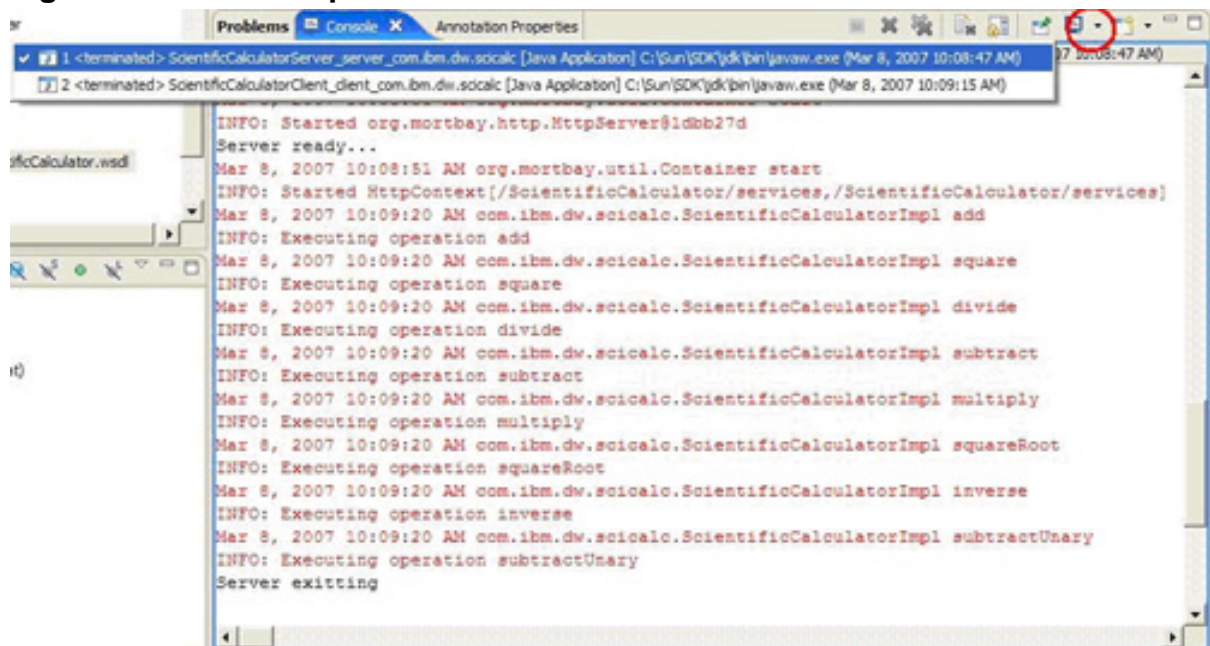
After clicking **Run**, the client will run and execute each operation in the console, as shown below.

**Figure 10. Results of executing the client**



You can see the server output of the client's execution by switching to its console, as shown below.

**Figure 11. Server output**



There you are! You have a completely functional Web service. Next, we do a final test by deploying the Web service on Apache Tomcat.

## Deploying and testing on Apache Tomcat

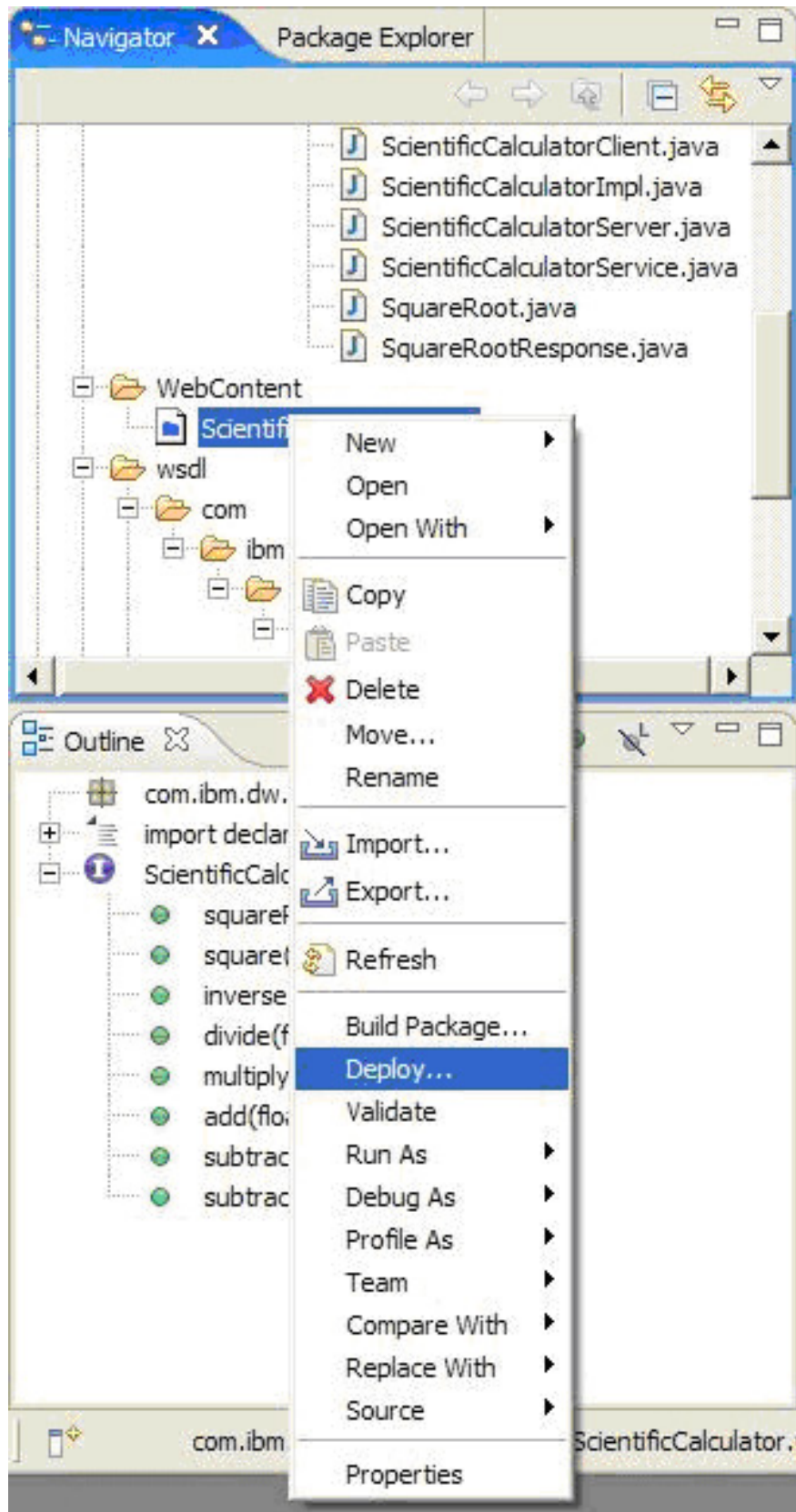
Before you begin setting up Tomcat inside of Eclipse, you first need to copy some jars over to Tomcat's shared/lib directory by doing the following: Copy all jars except cxf-integration-jbi-\*.jar (note the asterisk being a wildcard), or cxf-integration-jbi-2.0-incubator-RC-SNAPSHOT.jar for this tutorial, from CXF-Runtime-install-directory/lib to Tomcat-install-directory/shared/lib.

OK -- you're ready to move on and deploy the package on Tomcat using the Eclipse DTP. You'll begin by creating a new connector (the first four steps are also for if you want to optional start the Tomcat server within Eclipse):

1. Click **File > New > Other**
2. Open the Server folder and select Server
3. Select, under the Apache folder, Tomcat v5.5 Server
4. Click **Finish**
5. Now click **File > New > Other**
6. Open the Connection Profiles folder and select Connection Profile
7. Select Tomcat Connection Profile
8. Click **Next**
9. Enter a name, **Tomcat 5.5**, and click **Next**
10. Browse for the directory you installed Tomcat to (C:\apps\tomcat-5.5.20 for this tutorial)
11. Click **Finish**

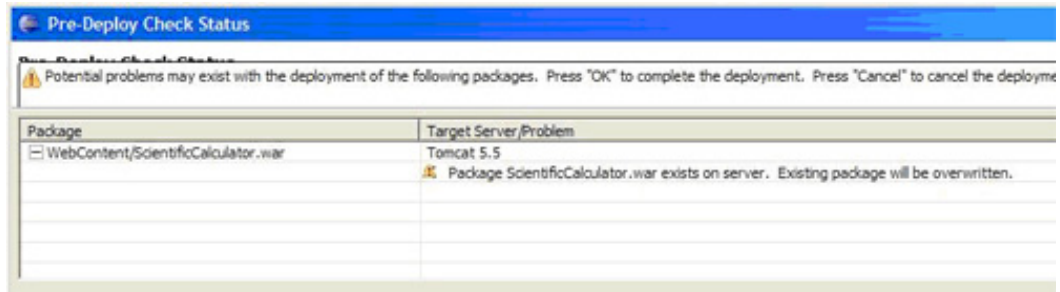
To deploy on Tomcat, start the Tomcat server by running startup.bat (Windows®) or startup.sh (Linux®). Once Tomcat is running, deploy your Web service by:

1. Selecting the WAR file in the Navigator view
2. Right-clicking it and selecting **Deploy**, as shown below  
**Figure 12. Server output**



3. Click the connection profile you made, Tomcat 5.5, and click **OK**
4. If you've deployed before, you'll get a warning message, as shown in Figure 13. Click **OK**

**Figure 13. Possible ignorable warning message**



Now run the client as before, and you should get exactly the same output shown in Figure 10. Tomcat's standard output should also display what's shown in Figure 11.

We have built and tested a Web service successfully using the Eclipse STP plug-in.

---

## Section 10. Summary

Wasn't that painless? Web service development using some tools can sometimes be quite difficult, and thankfully, this was a cakewalk in comparison. We annotated a Java interface, generated a WSDL from it that you generated client and service stubs from. We then defined code for the stubs, successfully deployed the service on both Eclipse's stand-alone server and Tomcat, and tested them both using your client.

But wait -- there's more. The Eclipse SOA Tools Platform (STP) plug-in is just in its infancy, and more functionality is being added all the time. To learn more about it, check out [Resources](#).

## Downloads

| Description | Name                        | Size | Download method      |
|-------------|-----------------------------|------|----------------------|
| Source code | os-eclipse-soatp.source.zip | 39KB | <a href="#">HTTP</a> |

[Information about download methods](#)

# Resources

## Learn

- Check out the [Eclipse SOA Tools Platform \(STP\) project](#) at Eclipse.org.
- Learn about the [Eclipse Service Creation \(SC\) subproject](#).
- Learn more at the [Eclipse STP wiki](#).
- To learn more about Service-Oriented Architecture (SOA), check out the developerWorks [SOA and Web services zone](#).
- Check out the developerWorks tutorial "[Grid and SOA](#)."
- Answers to your Eclipse questions are available in "[What is Eclipse and how do I use it?](#)"
- Visit the [Eclipse Foundation](#) to learn more about it and its many projects.
- Check out the extensive documentation, tutorials, presentations, and screencasts that illuminate the [Eclipse Test & Performance Tools Platform \(TPTP\)](#).
- For an excellent introduction to the Eclipse platform, see "[Getting started with the Eclipse Platform](#)."
- Check out the "[Recommended Eclipse reading list](#)."
- Browse all the [Eclipse content](#) on developerWorks.
- Expand your Eclipse skills by checking out IBM developerWorks' [Eclipse project resources](#).
- To listen to interesting interviews and discussions for software developers, check out [developerWorks podcasts](#).
- Stay current with developerWorks' [Technical events and webcasts](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

## Get products and technologies

- Get the latest [Eclipse SOA Tools Platform downloads](#).
- Visit Eclipse.org for [Eclipse data tools platform project downloads](#).
- Download the [CXF Runtime](#) at Apache.org.

- Get the latest [Apache Tomcat downloads](#) at Apache.org.
- Check out the latest [Eclipse technology downloads](#) at IBM [alphaWorks](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

### Discuss

- Connect with Eclipse developers and other users in the [Eclipse mailing lists and newsgroups](#). (Free registration required.)
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

## About the author

### Tyler Anderson

Tyler Anderson received his bachelor's degree in computer science in 2004 and his master's degree in electrical and computer engineering in 2005 from Brigham Young University. He worked with Stexar Corp. as a design engineer from May 2005 to August 2006. He was discovered by Backstop Media LLC in early 2005, and has written and coded numerous articles and tutorials for IBM developerWorks.