

- [2 Active Modules](#)
-
- [5 Badges](#)
-
- [8050 Points](#)

[Apex Integration Services](#)

Apex SOAP Callouts

Learning Objectives

After completing this module, you'll be able to:

- Generate Apex classes using WSDL2Apex.
- Perform a callout to send data to an external service using SOAP.
- Test callouts by using mock callouts.

Use WSDL2Apex to Generate Apex Code

In addition to REST callouts, Apex can also make callouts to SOAP web services using XML. Working with SOAP can be a painful (but necessary) experience. Fortunately, we have tools to make the process easier.

WSDL2Apex automatically generates Apex classes from a WSDL document. You download the web service's WSDL file, and then you upload the WSDL and WSDL2Apex generates the Apex classes for you. The Apex classes construct the SOAP XML, transmit the data, and parse the response XML into Apex objects. Instead of developing the logic to construct and parse the XML of the web service messages, let the Apex classes generated by WSDL2Apex internally handle all that overhead. If you are familiar with WSDL2Java or with importing a WSDL as a Web Reference in .NET, this functionality is similar to WSDL2Apex. You're welcome.



Note

Use outbound messaging to handle integration solutions when possible. Use callouts to third-party web services only when necessary.

For this example, we're using a simple calculator web service to add two numbers. It's a groundbreaking service that is all the rage! The first thing we need to do is download the WSDL file to generate the Apex classes. [Click this link](#) and download the calculator.xml file to your computer. Remember where you save this file, because you need it in the next step.

Generate an Apex Class from the WSDL

1. From Setup, enter `Apex classes` in the Quick Find box, then click **Apex Classes**.
2. Click **Generate from WSDL**.
3. Click **Choose File** and select the downloaded calculator.xml file.
4. Click **Parse WSDL**. The application generates a default class name for each namespace in the WSDL document and reports any errors.

For this example, use the default class name. However, in real life it is highly recommended that you change the default names to make them easier to work with and make your code more intuitive.

It's time to talk honestly about the WSDL parser. WSDL2Apex parsing is a notoriously fickle beast. The parsing process can fail for several reasons, such as an unsupported type, multiple bindings, or unknown elements. Unfortunately, you could be forced to manually code the Apex classes that call the web service or use HTTP.

5. Click **Generate Apex code**. The final page of the wizard shows the generated classes, along with any errors. The page also provides a link to view successfully generated code.


The generated Apex classes include stub and type classes for calling the third-party web service represented by the WSDL document. These classes allow you to call the external web service from Apex. For each generated class, a second class is created with the same name and the prefix `Async`. The `calculatorServices` class is for synchronous callouts. The `AsyncCalculatorServices` class is for asynchronous callouts.

Execute the Callout

Prerequisites

Before you run this example, authorize the endpoint URL of the web service callout, <https://th-apex-soap-service.herokuapp.com>, using the steps from the [Authorize Endpoint Addresses](#) section.

Now you can execute the callout and see if it correctly adds two numbers. Have a calculator handy to check the results.

1. Open the Developer Console under Your Name or the quick access menu (.
2. In the Developer Console, select **Debug | Open Execute Anonymous Window**.
3. Delete all existing code and insert the following snippet.

```
1 | calculatorServices.CalculatorImplPort calculator = new calculatorServices.CalculatorImplPort();
2 | Double x = 1.0;
3 | Double y = 2.0;
4 | Double result = calculator.doAdd(x, y);
5 | System.debug(result);
```

4. Select **Open Log**, and then click **Execute**.
5. After the debug log opens, click **Debug Only** to view the output of the `System.debug` statements. The log should display `3.0`.

Test Web Service Callouts

All experienced Apex developers know that to deploy or package Apex code, at least 75% of that code must have test coverage. This coverage includes our classes generated by WSDL2Apex. You might have heard this before, but test methods don't support web service callouts, and tests that perform web service callouts fail. So, we have a little work to do. To prevent tests from failing and to increase code coverage, Apex provides a built-in `WebServiceMock` interface and the `Test.setMock` method. You can use this interface to receive fake responses in a test method, thereby providing the necessary test coverage.

Specify a Mock Response for Callouts

When you create an Apex class from a WSDL, the methods in the autogenerated class call `WebServiceCallout.invoke`, which performs the callout to the external service. When testing these methods, you can instruct the Apex runtime to generate a fake response whenever `WebServiceCallout.invoke` is called. To do so, implement the `WebServiceMock` interface and specify a fake response for the testing runtime to send.

Instruct the Apex runtime to send this fake response by calling `Test.setMock` in your test method. For the first argument, pass `WebServiceMock.class`. For the second argument, pass a new instance of your `WebServiceMock` interface implementation.

```
1 | Test.setMock(WebServiceMock.class, new MyWebServiceMockImpl());
```

That's a lot to grok, so let's look at some code for a complete example. In this example, you create the class that makes the callout, a mock implementation for testing, and the test class itself.

1. In the Developer Console, select **File | New | Apex Class**.
2. For the class name, enter `AwesomeCalculator` and then click **OK**.
3. Replace autogenerated code with the following class definition.

```
1 | public class AwesomeCalculator {
2 |     public static Double add(Double x, Double y) {
3 |         calculatorServices.CalculatorImplPort calculator =
4 |             new calculatorServices.CalculatorImplPort();
5 |         return calculator.doAdd(x, y);
6 |     }
7 | }
```

4. Press **CTRL+S** to save.

Create your mock implementation to fake the callout during testing. Your implementation of `WebServiceMock` calls the `doInvoke` method, which returns the response you specify for testing. Most of this code is boilerplate. The hardest part of this exercise is figuring out how the web service returns a response so that you can fake a value.

5. In the Developer Console, select **File | New | Apex Class**.
6. For the class name, enter `CalculatorCalloutMock` and then click **OK**.
7. Replace the autogenerated code with the following class definition.

```

01 | @isTest
02 | global class CalculatorCalloutMock implements WebserviceMock {
03 |     global void doInvoke(
04 |         Object stub,
05 |         Object request,
06 |         Map<String, Object> response,
07 |         String endpoint,
08 |         String soapAction,
09 |         String requestName,
10 |         String responseNS,
11 |         String responseName,
12 |         String responseType) {
13 |         // start - specify the response you want to send
14 |         calculatorServices.doAddResponse response_x =
15 |             new calculatorServices.doAddResponse();
16 |         response_x.return_x = 3.0;
17 |         // end
18 |         response.put('response_x', response_x);
19 |     }
20 | }

```

8. Press **CTRL+S** to save.

Lastly, your test method needs to instruct the Apex runtime to send the fake response by calling `Test.setMock` before making the callout in the `AwesomeCalculator` class. Like any other test method, we assert that the correct result from our mock response was received.

9. In the Developer Console, select **File | New | Apex Class**.

10. For the class name, enter `AwsomeCalculatorTest` and then click **OK**.

11. Replace the autogenerated code with the following class definition.

```

01 | @isTest
02 | private class AwsomeCalculatorTest {
03 |     @isTest static void testCallout() {
04 |         // This causes a fake response to be generated
05 |         Test.setMock(WebserviceMock.class, new CalculatorCalloutMock());
06 |         // Call the method that invokes a callout
07 |         Double x = 1.0;
08 |         Double y = 2.0;
09 |         Double result = AwsomeCalculator.add(x, y);
10 |         // Verify that a fake result is returned
11 |         System.assertEquals(3.0, result);
12 |     }
13 | }

```

12. Press **CTRL+S** to save.

13. To run the test, select **Test | Run All**.

The `AwsomeCalculator` class should now display 100% code coverage!

Resources

- [Apex Developer Guide: SOAP Services: Defining a Class from a WSDL Document](#)
- [Apex Developer Guide: Test Web Service Callouts](#)
- [Announcing the Open-Source WSDL2Apex Generator](#)



Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, [switch to Salesforce Classic](#) to complete this challenge.

Hands-on Challenge +500 points

You'll be completing this challenge in your own personal Salesforce environment. Ready to get hands-on?

Default TP

- Choose your hands-on org
- [Default TP](#)
-
- [Log into a Developer Edition](#)
- [Create a Trailhead Playground](#)
- [Manage my hands-on orgs](#)

[LaunchHands-On Org Help](#)

Generate an Apex class using WSDL2Apex and write a test class.

Generate an Apex class using WSDL2Apex for a SOAP web service, write unit tests that achieve 100% code coverage for the class using a mock response, and run your Apex tests.

- Use WSDL2Apex to generate a class called 'ParkService' in public scope using [this WSDL file](#). After you click the 'Parse WSDL' button don't forget to change the name of the Apex Class Name from 'parksServices' to 'ParkService'.
- Create a class called 'ParkLocator' that has a 'country' method that uses the 'ParkService' class and returns an array of available park names for a particular country passed to the web service. Possible country names that can be passed to the web service include Germany, India, Japan and United States.
- Create a test class named ParkLocatorTest that uses a mock class called ParkServiceMock to mock the callout response.
- The unit tests must cover all lines of code included in the ParkLocator class, resulting in 100% code coverage.
- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Check challenge

Share



15



?



13

Time Estimate

About **20** mins

Topics

- [Learning Objectives](#)
- [Use WSDL2Apex to Generate Apex Code](#)
- [Test Web Service Callouts](#)
- [Resources](#)
-
- [Challenge +500 points](#)
-

Have questions about Trailhead or having problems using it?

[Trailhead Forum](#)

Want to send us Feedback or Ideas?

Submit Feedback

☐