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.

**Follow Along with Trail Together**

Want to follow along with an expert as you work through this step? Take a look at this video, part of the Trail Together series on Trailhead Live.

(This clip starts at the 45:49 minute mark, in case you want to rewind and watch the beginning of the step again.)

**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

**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](https://th-apex-soap-service.herokuapp.com/assets/calculator.xml) 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.

1. 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](https://trailhead.salesforce.com/apex_integration_services/apex_integration_callouts.htm#apex_integration_callouts_authorizing) 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 from the Setup gear (Setup gear icon).
2. In the Developer Console, select **Debug** | **Open Execute Anonymous Window**.
3. Delete all existing code and insert the following snippet.
4. calculatorServices.CalculatorImplPort calculator = new calculatorServices.CalculatorImplPort();
5. Double x = 1.0;
6. Double y = 2.0;
7. Double result = calculator.**doAdd**(x,y);

System.**debug**(result);

Copy

1. Select **Open Log**, and then click **Execute**.
2. 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.

Test.**setMock**(WebServiceMock.class, new MyWebServiceMockImpl());

Copy

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.
4. public class AwesomeCalculator {
5. public static Double **add**(Double x, Double y) {
6. calculatorServices.CalculatorImplPort calculator =
7. new calculatorServices.CalculatorImplPort();
8. return calculator.**doAdd**(x,y);
9. }

}

Copy

1. 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.

1. In the Developer Console, select **File** | **New** | **Apex Class**.
2. For the class name, enter CalculatorCalloutMock and then click **OK**.
3. Replace the autogenerated code with the following class definition.
4. @isTest
5. global class CalculatorCalloutMock implements WebServiceMock {
6. global void **doInvoke**(
7. Object stub,
8. Object request,
9. Map<String, Object> response,
10. String endpoint,
11. String soapAction,
12. String requestName,
13. String responseNS,
14. String responseName,
15. String responseType) {
16. // start - specify the response you want to send
17. calculatorServices.doAddResponse response\_x =
18. new calculatorServices.**doAddResponse**();
19. response\_x.return\_x = 3.0;
20. // end
21. response.**put**('response\_x', response\_x);
22. }

}

Copy

1. 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.

1. In the Developer Console, select **File** | **New** | **Apex Class**.
2. For the class name, enter AwesomeCalculatorTest and then click **OK**.
3. Replace the autogenerated code with the following class definition.
4. @isTest
5. private class AwesomeCalculatorTest {
6. @isTest static void **testCallout**() {
7. // This causes a fake response to be generated
8. Test.**setMock**(WebServiceMock.class, new CalculatorCalloutMock());
9. // Call the method that invokes a callout
10. Double x = 1.0;
11. Double y = 2.0;
12. Double result = AwesomeCalculator.**add**(x, y);
13. // Verify that a fake result is returned
14. System.**assertEquals**(3.0, result);
15. }

}

Copy

1. Press **CTRL+S** to save.
2. To run the test, select **Test** | **Run All**.

The AwesomeCalculator class should now display 100% code coverage!

**Resources**

* [*Apex Developer Guide*: SOAP Services: Defining a Class from a WSDL Document](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_callouts_wsdl2apex.htm)
* [*Apex Developer Guide*: Test Web Service Callouts](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_callouts_wsdl2apex_testing.htm)
* [*Salesforce Developers Blog*: Announcing the Open-Source WSDL2Apex Generator](https://developer.salesforce.com/blogs/engineering/2014/09/announcing-open-source-wsdl2apex-generator.html)

Top of Form

**Hands-on Challenge**

**+500 points**

**GET READY**

You’ll be completing this unit in your own hands-on org. Click **Launch** to get started, or click the name of your org to choose a different one.

If you use Trailhead in a language other than English, make sure that your hands-on org is set to the same language as the challenge instructions. Otherwise you may run into issues passing this unit. Want to find out more about using hands-on orgs on Trailhead? Check out [Trailhead Playground Management](https://trailhead.salesforce.com/en/content/learn/modules/trailhead_playground_management).

**YOUR CHALLENGE**

**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.  
  
**Prework**: Be sure the Remote Sites from the first unit are set up.

* Generate a class using this using [this WSDL file](https://th-apex-soap-service.herokuapp.com/assets/parks.xml):
  + Name: ParkService (Tip: After you click the **Parse WSDL** button, change the Apex class name from **parksServices** to ParkService)
  + Class must be in public scope
* Create a class:
  + Name: ParkLocator
  + Class must have a**country** method that uses the **ParkService** class
  + Method must return an array of available park names for a particular country passed to the web service (such as Germany, India, Japan, and United States)
* Create a test class:
  + Name: ParkLocatorTest
  + Test class uses a mock class called ParkServiceMock to mock the callout response
* Create unit tests:
  + 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.

**swamy2**

Last used on 7/6/2022

Bottom of Form