1. Apex Integration Services
2. Apex REST Callouts

Apex REST Callouts

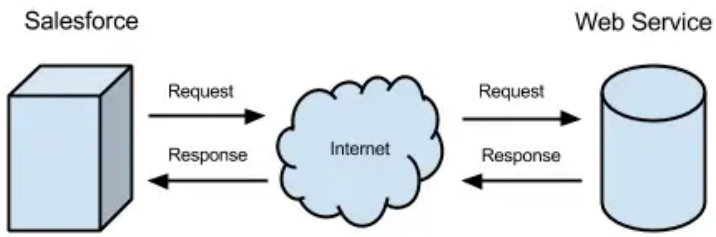
Learning Objectives

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

- Perform a callout to receive data from an external service.
- Perform a callout to send data to an external service.
- Test callouts by using mock callouts.

HTTP and Callout Basics

REST callouts are based on HTTP. To understand how callouts work, it's helpful to understand a few things about HTTP. Each callout request is associated with an HTTP method and an endpoint. The HTTP method indicates what type of action is desired.



The simplest request is a GET request (GET is an HTTP method). A GET request means that the sender wants to obtain information about a resource from the server. When the server receives and processes this request, it returns the request information to the recipient. A GET request is similar to navigating to an address in the browser. When you visit a web page, the browser performs a GET request behind the scenes. In the browser, the result of the navigation is a new HTML page that's displayed. With a callout, the result is the response object.

To illustrate how a GET request works, open your browser and navigate to the following URI: https://th-apex-http-callout.herokuapp.com/animals. Your browser displays a list of animals in a weird format, because the service returns the response in a format called JSON. Sometimes a GET response is also formatted in XML.

The following are descriptions of common HTTP methods.

Table 1. Some Common HTTP Methods

HTTP Method	Description

HTTP Method	Description
GET	Retrieve data identified by a URL.
POST	Create a resource or post data to the server.
DELETE	Delete a resource identified by a URL.
PUT	Create or replace the resource sent in the request body.

If you have some free time, browse the exhaustive list of all HTTP methods in the Resources section.

In addition to the HTTP method, each request sets a URI, which is the endpoint address at which the service is located. For example, an endpoint can be http://www.example.com/api/resource. In the example in the "HTTP and Callout Basics" unit, the endpoint is https://th-apex-http-callout.herokuapp.com/animals.

When the server processes the request, it sends a status code in the response. The status code indicates whether the request was processed successfully or whether errors were encountered. If the request is successful, the server sends a status code of `200`. You've probably seen some other status codes, such as 404 for file not found or 500 for an internal server error.

If you still have free time after browsing the list of HTTP methods, check out the list of all response status codes in the Resources section. If you're having a hard time sleeping at night, these two resources can help.

Note

In addition to the endpoint and the HTTP method, you can set other properties for a request. For example, a request can contain headers that provide more information about the request, such as the content type. A request can also contain data to be sent to the service, such as for POST requests. You'll see an example of a POST request in a later step. A POST request is similar to clicking a button on a web page to submit form data. With a callout, you send data as part of the body of the request instead of manually entering the data on a web page.

Get Data from a Service

It's time to put your new HTTP knowledge to use with some Apex callouts. This example sends a GET request to a web service to get a list of woodland creatures. The service sends the response in JSON format. JSON is essentially a string, so the built-in `JSONParser` class converts it to an object. We can then use that object to write the name of each animal to the debug log.

Before you run the examples in this unit, you'll need to authorize the endpoint URL of the callout, https://th-apex-http-callout.herokuapp.com, using the steps from the Authorize Endpoint Addresses section.

1. Open the Developer Console from the Setup gear ( ).
2. In the Developer Console, select **Debug** | **Open Execute Anonymous Window**.
3. Delete the existing code and insert the following snippet.

```
Http http = new Http();
HttpRequest request = new HttpRequest();
request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
request.setMethod('GET');
HttpResponse response = http.send(request);
// If the request is successful, parse the JSON response.
if (response.getStatusCode() == 200) {
    // Deserialize the JSON string into collections of primitive data types.
```

```
    Map<String, Object> results = (Map<String, Object>) JSON.deserializeUntyped(response.getBody());
    // Cast the values in the 'animals' key as a list
    List<Object> animals = (List<Object>) results.get('animals');
    System.debug('Received the following animals:');
    for (Object animal: animals) {
        System.debug(animal);
    }
}
```

Copy

4. Select **Open Log**, and then click **Execute**.
5. After the debug log opens, select **Debug Only** to view the output of the System.debug statements. The names of the animals are displayed.

The JSON for our example is fairly simple and easy to parse. For more complex JSON structures, you can use JSON2Apex. This tool generates strongly typed Apex code for parsing a JSON structure. You simply paste in the JSON, and the tool generates the necessary Apex code for you. Brilliant!

Send Data to a Service

Another common use case for HTTP callouts is sending data to a service. For instance, when you buy the latest Justin Bieber album or comment on your favorite “Cat in a Shark Costume Chases a Duck While Riding a Roomba” video, your browser is making a POST request to submit data. Let’s see how we send data in Apex.

This example sends a POST request to the web service to add an animal name. The new name is added to the request body as a JSON string. The request `Content-Type` header is set to let the service know that the sent data is in JSON format so that it can process the data appropriately. The service responds by sending a status code and a list of all animals, including the one you added. If the request was processed successfully, the status code returns 201, because a resource has been created. The response is sent to the debug log when anything other than 201 is returned.

1. Open the Developer Console from the Setup gear ( ).
2. In the Developer Console, select **Debug** | **Open Execute Anonymous Window**.
3. Delete any existing code and insert the following snippet.

```
Http http = new Http();
HttpRequest request = new HttpRequest();
request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
request.setMethod('POST');
request.setHeader('Content-Type', 'application/json;charset=UTF-8');
// Set the body as a JSON object
request.setBody('{"name":"mighty moose"}');
HttpResponse response = http.send(request);
// Parse the JSON response
if (response.getStatusCode() != 201) {
    System.debug('The status code returned was not expected: ' +
        response.getStatusCode() + ' ' + response.getStatus());
} else {
    System.debug(response.getBody());
}
```

Copy

4. Select **Open Log**, and then click **Execute**.
5. When the debug log opens, select **Debug Only** to view the output of the `System.debug` statement. The last item in the list of animals is `"mighty moose"`.

Test Callouts

There's good news and bad news regarding callout testing. The bad news is that Apex test methods don't support callouts, and tests that perform callouts fail. The good news is that the testing runtime allows you to "mock" the callout. Mock callouts allow you to specify the response to return in the test instead of actually calling the web service. You are essentially telling the runtime, "I know what this web service will return, so instead of calling it during testing, just return this data." Using mock callouts in your tests helps ensure that you attain adequate code coverage and that no lines of code are skipped due to callouts.

Prerequisites

Before you write your tests, let's create a class that contains the GET and POST request examples we executed anonymously in the "Send Data to a Service" unit. These examples are slightly modified so that the requests are in methods and return values, but they're essentially the same as the previous examples.

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

```
public class AnimalsCallouts {
    public static HttpResponse makeGetCallout() {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
        request.setMethod('GET');
        HttpResponse response = http.send(request);
        // If the request is successful, parse the JSON response.
        if (response.getStatusCode() == 200) {
            // Deserializes the JSON string into collections of primitive data types.
            Map<String, Object> results = (Map<String, Object>) JSON.deserializeUntyped(response.getBody());
            // Cast the values in the 'animals' key as a list
            List<Object> animals = (List<Object>) results.get('animals');
            System.debug('Received the following animals:');
            for (Object animal: animals) {
                System.debug(animal);
            }
        }
        return response;
    }
    public static HttpResponse makePostCallout() {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
        request.setMethod('POST');
        request.setHeader('Content-Type', 'application/json;charset=UTF-8');
        request.setBody('{"name":"mighty moose"}');
        HttpResponse response = http.send(request);
        // Parse the JSON response
        if (response.getStatusCode() != 201) {
            System.debug('The status code returned was not expected: ' +
                response.getStatusCode() + ' ' + response.getStatus());
        } else {
            System.debug(response.getBody());
        }
        return response;
    }
}
```

Copy

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

Test a Callout with StaticResourceCalloutMock

To test your callouts, use mock callouts by either implementing an interface or using static resources. In this example, we use static resources and a mock interface later on. The static resource contains the response body to return. Again, when using a mock callout, the request isn't sent to the endpoint. Instead, the Apex runtime knows to look up the response specified in the static resource and

return it instead. The `Test.setMock` method informs the runtime that mock callouts are used in the test method. Let's see mock callouts in action. First, we create a static resource containing a JSON-formatted string to use for the GET request.

1. In the Developer Console, select **File** | **New** | **Static Resource**.
2. For the name, enter `GetAnimalResource` .
3. For the MIME type, select **text/plain**, even though we are using JSON.
4. Click **Submit**.
5. In the tab that opens for the resource, insert the following content. Make sure that it is all on one line and doesn't break to the next line. This content is what the mock callout returns. It's an array of three woodland creatures.

```
{"animals": ["pesky porcupine", "hungry hippo", "squeaky squirrel"]}
```

Copy

6. Press `CTRL+S` to save.

You've successfully created your static resource! Now, let's add a test for our callout that uses this resource.

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

```
@isTest
private class AnimalsCalloutsTest {
    @isTest static  void testGetCallout() {
        // Create the mock response based on a static resource
        StaticResourceCalloutMock mock = new StaticResourceCalloutMock();
        mock.setStaticResource('GetAnimalResource');
        mock.setStatusCode(200);
        mock.setHeader('Content-Type', 'application/json;charset=UTF-8');
        // Associate the callout with a mock response
        Test.setMock(HttpCalloutMock.class, mock);
        // Call method to test
        HttpResponse result = AnimalsCallouts.makeGetCallout();
        // Verify mock response is not null
        System.assertNotEquals(null,result,
            'The callout returned a null response.');
        // Verify status code
        System.assertEquals(200,result.getStatusCode(),
          'The status code is not 200.');
        // Verify content type
        System.assertEquals('application/json;charset=UTF-8',
          result.getHeader('Content-Type'),
          'The content type value is not expected.');
        // Verify the array contains 3 items
        Map<String, Object> results = (Map<String, Object>)
            JSON.deserializeUntyped(result.getBody());
        List<Object> animals = (List<Object>) results.get('animals');
        System.assertEquals(3, animals.size(),
          'The array should only contain 3 items.');
    }
}
```

Copy

4. Press **CTRL+S** to save.
5. Select **Test** | **Always Run Asynchronously**. If you don't select Always Run Asynchronously, test runs that include only one class run synchronously. You can open logs from the Tests tab only for synchronous test runs.
6. To run the test, select **Test** | **New Run**.
7. From the Test Classes list, select **AnimalsCalloutsTest**.
8. Click **Add Selected** | **Run**.

The test result is displayed in the Tests tab under a test run ID. When the test execution finishes, expand the test run to view details. Now double-click `AnimalCallouts` in the Overall Code Coverage pane to see which lines are covered by your tests.

Test a Callout with HttpCalloutMock

To test your POST callout, we provide an implementation of the `HttpCalloutMock` interface. This interface enables you to specify the response that's sent in the `respond` method. Your test class instructs the Apex runtime to send this fake response by calling `Test.setMock` again. For the first argument, pass `HttpCalloutMock.class`. For the second argument, pass a new instance of `AnimalsHttpCalloutMock`, which is your interface implementation of `HttpCalloutMock`. (We'll write `AnimalsHttpCalloutMock` in the example after this one.)

```
Test.setMock(HttpCalloutMock.class, new AnimalsHttpCalloutMock());
```

Copy

Now add the class that implements the `HttpCalloutMock` interface to intercept the callout. If an HTTP callout is invoked in test context, the callout is not made. Instead, you receive the mock response that you specify in the `respond` method implementation in `AnimalsHttpCalloutMock`.

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

```
@isTest
global class AnimalsHttpCalloutMock implements HttpCalloutMock {
    // Implement this interface method
    global HTTPResponse respond(HTTPRequest request) {
        // Create a fake response
        HttpResponse response = new HttpResponse();
        response.setHeader('Content-Type', 'application/json');
        response.setBody('{"animals": ["majestic badger", "fluffy bunny", "scary bear", "chicken", "mighty moose"]}');
        response.setStatusCode(200);
        return response;
    }
}
```

Copy

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

In your test class, create the `testPostCallout` method to set the mock callout, as shown in the next example. The `testPostCallout` method calls `Test.setMock`, and then calls the `makePostCallout` method in the `AnimalsCallouts` class. It then verifies that the response that's returned is what you specified in the respond method of the mock implementation.

1. Modify the test class `AnimalsCalloutsTest` to add a second test method. Click the class tab, and then add the following method before the closing bracket.

```
@isTest static void testPostCallout() {
    // Set mock callout class
    Test.setMock(HttpCalloutMock.class, new AnimalsHttpCalloutMock());
    // This causes a fake response to be sent
    // from the class that implements HttpCalloutMock.
    HttpResponse response = AnimalsCallouts.makePostCallout();
    // Verify that the response received contains fake values
    String contentType = response.getHeader('Content-Type');
    System.assert(contentType == 'application/json');
    String actualValue = response.getBody();
    System.debug(response.getBody());
    String expectedValue = '{"animals": ["majestic badger", "fluffy bunny", "scary bear", "chicken", "mighty moose"]}';
    System.assertEquals(actualValue, expectedValue);
    System.assertEquals(200, response.getStatusCode());
}
```

Copy

2. Press **CTRL+S** to save.
3. Select **Test** | **New Run**.
4. From the Test Classes list, select **AnimalsCalloutsTest**.
5. Click **Add Selected** | **Run**.

   The test result displays in the Tests tab under a new test run ID. When the test execution finishes, expand the test run to view details about both tests.

Tell Me More...

Learn about using callouts in triggers and in asynchronous Apex, and about making asynchronous callouts.

When making a callout from a method, the method waits for the external service to send back the callout response before executing subsequent lines of code. Alternatively, you can place the callout code in an asynchronous method that's annotated with `@future(callout=true)` or use Queueable Apex. This way, the callout runs on a separate thread, and the execution of the calling method isn't blocked.

When making a callout from a trigger, the callout must not block the trigger process while waiting for the response. For the trigger to be able to make a callout, the method containing the callout code must be annotated with `@future(callout=true)` to run in a separate thread.

Resources

- *Apex Developer Guide*: Invoking Callouts Using Apex
- *Wikipedia*: Representational state transfer
- *W3C Note*: Simple Object Access Protocol (SOAP) 1.1

Note

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

Assessment Complete!

+500 points

Apex Integration Services
100%
Progress: 100%
Retake this Challenge
View more modules