- 2 Active Modules
- 5 Badges
- 8550 Points

Apex Integration Services

Apex Web Services

Learning Objectives

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

- Describe the two types of Apex web services and provide a high-level overview of these services.
- Create an Apex REST class that contains methods for each HTTP method.
- Invoke a custom Apex REST method with an endpoint.
- Pass data to a custom Apex REST method by sending a request body in JSON format.
- Write a test method for an Apex REST method and set properties in a test REST request.
- Write a test method for an Apex REST method by calling the method with parameter values.

Expose Your Apex Class as a Web Service

You can expose your Apex class methods as a REST or SOAP web service operation. By making your methods callable through the web, your external applications can integrate with Salesforce to perform all sorts of nifty operations.

For example, say your company's call center is using an internal application to manage on-premises resources. Customer support representatives are expected to use the same application to perform their daily work, including managing case records in Salesforce. By using one interface, representatives can view and update case records and access internal resources. The application calls an Apex web service class to manage Salesforce case records.

Expose a Class as a REST Service

Making your Apex class available as a REST web service is straightforward. Define your class as global, and define methods as global static. Add annotations to the class and methods. For example, this sample Apex REST class uses one method. The `getRecord` method is a custom REST API call. It's annotated with `@HttpGet` and is invoked for a GET request.

```
@RestResource(urlMapping='/Account/*')
global with sharing class MyRestResource {
    @HttpGet
    global static Account getRecord() {
        // Add your code
    }
}
```

As you can see, the class is annotated with `@RestResource(urlMapping='/Account/*)`. The base endpoint for Apex REST is https://*yourInstance*.salesforce.com/services/apexrest/. The URL mapping is appended to the base endpoint to form the endpoint for your REST service. For example, in the class example, the REST endpoint is https://*yourInstance*.salesforce.com/services/apexrest/Account/. For your org, it could look something like, https://*yourInstance*.salesforce.com/services/apexrest/Account/.

The URL mapping is case-sensitive and can contain a wildcard character (*).

Define each exposed method as `global static` and add an annotation to associate it with an HTTP method. The following annotations are available. You can use each annotation only once in each Apex class.

Annotation	Action	Details
`@HttpGet`	Read	Reads or retrieves records.
`@HttpPost`	Create	Creates records.
`@HttpDelete`	Delete	Deletes records.
`@HttpPut`	Upsert	Typically used to update existing records or create records.

Annotation	Action	Details
`@HttpPatch`	Update	Typically used to update fields in existing records.

Expose a Class as a SOAP Service

Making your Apex class available as a SOAP web service is as easy as with REST. Define your class as global. Add the `webservice` keyword and the `static` definition modifier to each method you want to expose. The `webservice` keyword provides global access to the method it is added to.

For example, here's a sample class with one method. The `getRecord` method is a custom SOAP API call that returns an Account record.

```
global with sharing class MySOAPWebService {
    webservice static Account getRecord(String id) {
        // Add your code
    }
}
```

The external application can call your custom Apex methods as web service operations by consuming the class WSDL file. Generate this WSDL for your class from the class detail page, accessed from the Apex Classes page in Setup. You typically send the WSDL file to third-party developers (or use it yourself) to write integrations for your web service.

Because platform security is a first-class Salesforce citizen, your web service requires authentication. In addition to the Apex class WSDL, external applications must use either the Enterprise WSDL or the Partner WSDL for login functionality.

Apex REST Walkthrough

Now the fun stuff. The next few steps walk you through the process of building an Apex REST service. First, you create the Apex class that is exposed as a REST service. Then you try calling a few methods from a client, and finally write unit tests. There's quite a bit of code, but it will be worth the effort!

Your Apex class manages case records. The class contains five methods, and each method corresponds to an HTTP method. For example, when the client application invokes a REST call for the GET HTTP method, the `getCaseById` method is invoked.

Because the class is defined with a URL mapping of /Cases/*, the endpoint used to call this REST service is any URI that starts with https://*yourInstance*.salesforce.com/services/apexrest/Cases/.

We suggest that you also think about versioning your API endpoints so that you can provide upgrades in functionality without breaking existing code. You could create two classes specifying URL mappings of /Cases/v1/* and /Cases/v2/* to implement this functionality.

Let's get started by creating an Apex REST class.

1. Open the Developer Console under Your Name or the quick access menu ( ).
2. In the Developer Console, select **File | New | Apex Class**.
3. For the class name, enter `CaseManager` and then click **OK**.
4. Replace the autogenerated code with the following class definition.

```
@RestResource(urlMapping='/Cases/*')
global with sharing class CaseManager {

    @HttpGet
    global static Case getCaseById() {
        RestRequest request = RestContext.request;
        // grab the caseId from the end of the URL
        String caseId = request.requestURI.substring(
          request.requestURI.lastIndexOf('/')+1);
        Case result =  [SELECT CaseNumber,Subject,Status,Origin,Priority
                        FROM Case
                        WHERE Id = :caseId];
        return result;
    }

```

```
    @HttpPost
    global static ID createCase(String subject, String status,
        String origin, String priority) {
        Case thisCase = new Case(
            Subject=subject,
            Status=status,
            Origin=origin,
            Priority=priority);
        insert thisCase;
        return thisCase.Id;
    }

    @HttpDelete
    global static void deleteCase() {
        RestRequest request = RestContext.request;
        String caseId = request.requestURI.substring(
            request.requestURI.lastIndexOf('/')+1);
        Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
        delete thisCase;
    }

    @HttpPut
    global static ID upsertCase(String subject, String status,
        String origin, String priority, String id) {
        Case thisCase = new Case(
                Id=id,
                Subject=subject,
                Status=status,
                Origin=origin,
                Priority=priority);
        // Match case by Id, if present.
        // Otherwise, create new case.
        upsert thisCase;
        // Return the case ID.
        return thisCase.Id;
    }

    @HttpPatch
    global static ID updateCaseFields() {
        RestRequest request = RestContext.request;
        String caseId = request.requestURI.substring(
            request.requestURI.lastIndexOf('/')+1);
        Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
        // Deserialize the JSON string into name-value pairs
        Map<String, Object> params = (Map<String, Object>)JSON.deserializeUntyped(request.requestbody.tostring());
        // Iterate through each parameter field and value
        for(String fieldName : params.keySet()) {
            // Set the field and value on the Case sObject
            thisCase.put(fieldName, params.get(fieldName));
        }
        update thisCase;
        return thisCase.Id;
    }

}
```

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

Create a Record with a POST Method

Let's use the Apex REST class that you've just created and have some fun. First, we'll call the POST method to create a case record.

To invoke your REST service, you need to use a… REST client! You can use almost any REST client, such as your own API client, the cURL command-line tool, or the curl library for PHP. We'll use the Workbench tool as our REST client application, but we'll take a peek at cURL later on.

Apex REST supports two formats for representations of resources: JSON and XML. JSON representations are passed by default in the body of a request or response, and the format is indicated by the `Content-Type` property in the HTTP header. Since JSON is easier to read and understand than XML, this unit uses JSON exclusively. In this step, you send a case record in JSON format.

Apex REST supports OAuth 2.0 and session authentication mechanisms. In simple terms, this means that we use industry standards to keep your application and data safe. Fortunately, you can use Workbench to make testing easier. Workbench is a powerful, web-based suite of tools for administrators and developers to interact with orgs via Force.com APIs. With Workbench, you use session authentication as you log in with your username and password to Salesforce. And you use the REST Explorer to call your REST service.

1. Navigate to https://workbench.developerforce.com/login.php.
2. For Environment, select **Production**.
3. Select the latest API version from the **API Version** drop-down.
4. Accept the terms of service, and click **Login with Salesforce**.
5. To allow Workbench to access your information, click **Allow**.
6. Enter your login credentials and then click **Log in to Salesforce**.
7. After logging in, select **utilities | REST Explorer**.
8. Select **POST**.
9. The URL path that REST Explorer accepts is relative to the instance URL of your org. Provide only the path that is appended to the instance URL. In the relative URI input field, replace the default URI with `/services/apexrest/Cases/`.
10. For the request body, insert the following JSON string representation of the object to insert.

```
{
  "subject" : "Bigfoot Sighting!",
  "status" : "New",
  "origin" : "Phone",
  "priority" : "Low"
}
```

11. Click **Execute**.
    This invocation calls the method that is associated with the POST HTTP method, namely the `createCase` method.
12. To view the response returned, click **Show Raw Response**.
    The returned response looks similar to this response. The response contains the ID of the new case record. Your ID value is likely different from `50061000000t7kYAAQ`. Save your ID value to use in the next steps.

```
HTTP/1.1 200 OK
Date: Wed, 07 Oct 2015 14:18:20 GMT
Set-Cookie: BrowserId=F1wxIhHPQHCXp6wrvqToXA;Path=/;Domain=.salesforce.com;Expires=Sun, 06-Dec-2015 14:18:20 GMT
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json;charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked

"50061000000t7kYAAQ"
```

Retrieve Data with a Custom GET Method

By following similar steps as before, use Workbench to invoke the GET HTTP method.

1. In Workbench, select **GET**.
2. Enter the URI /services/apexrest/Cases/<*Record ID*>, replacing <*Record ID*> with the ID of the record you created in the previous step.
3. Click **Execute**.

   This invocation calls the method associated with the GET HTTP method, namely the `getCaseById` method.
4. To view the response returned, click **Show Raw Response**.

The returned response looks similar to this response. The response contains the fields that the method queried for the new case record.

```
HTTP/1.1 200 OK
Date: Wed, 07 Oct 2015 14:28:20 GMT
Set-Cookie: BrowserId=j5qAnPDdRxSu8eHGqaRVLQ;Path=/;Domain=.salesforce.com;Expires=Sun, 06-Dec-2015 14:28:20 GMT
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json;charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked

{
  "attributes" : {
    "type" : "Case",
    "url" : "/services/data/v34.0/sobjects/Case/50061000000t7kYAAQ"
  },
  "CaseNumber" : "00001026",
  "Subject" : "Bigfoot Sighting!",
  "Status" : "New",
  "Origin" : "Phone",
  "Priority" : "Low",
  "Id" : "50061000000t7kYAAQ"
}
```

Retrieve Data Using cURL

Every good developer should know at least three things: 1) how to make an animated GIF of yourself eating ice cream; 2) the value of pi to 25 decimal places; and 3) how to use cURL. The first two are beyond the scope of this module, so we'll concentrate on the last one.

cURL is a command-line tool for getting or sending files using URL syntax. It comes in quite handy when working with REST endpoints. Instead of using Workbench for your Apex REST service, you use cURL to invoke the GET HTTP method. Each time you "cURL" your REST endpoint, you pass along the session ID for authorization. You were spoiled when working in Workbench because it passes the session ID for you, under the covers, after you log in.

To obtain a session ID, you first create a connected app in your Salesforce organization and enable OAuth. Your client application, cURL in this case, uses the connected app to connect to Salesforce. Follow these instructions to create a connected app that provides you with the consumer key and consumer secret that you need to get your session ID. When selecting the OAuth scopes for your connected app, choose the "Access and manage your data (api)" scope. It can take 5 to 10 minutes for the connected app to finish setting up. When ready, use the following cURL command with your credentials and the connected app.

```
curl -v https://login.salesforce.com/services/oauth2/token -d "grant_type=password" -d "client_id=<your_consumer_key>" -d "client_secret=
<your_consumer_secret>" -d "username=<your_username>" -d "password=<your_password_and_security_token>" -H 'X-PrettyPrint:1'
```

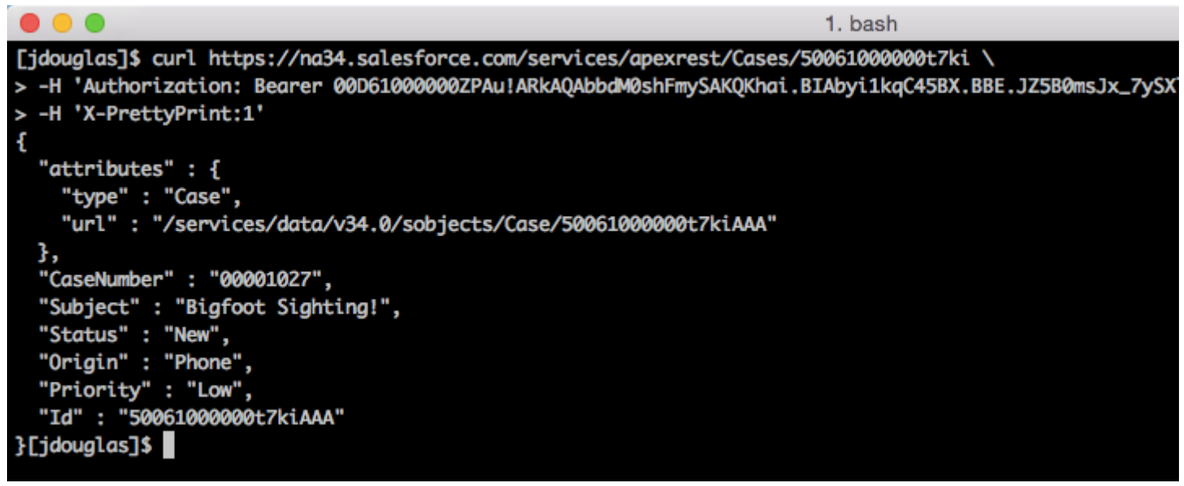
If all was successful, the result includes an access_token, which is your session ID and instance_url for your organization.

```
{
  "access_token" : "00D61000000ZPAu!ARkAQH1z_gd.ugz7ZqFJl4ocPX9kpQOcneLFucBPNXRNVNNDJ5e17TB3FaYD_RBB.azIkorg
  "instance_url" : "https://na34.salesforce.com",
  "id" : "https://login.salesforce.com/id/00D61000000ZPAuEAO/00561000000jSRwAAM",
  "token_type" : "Bearer",
  "issued_at" : "1494964213626",
  "signature" : "JVu8Rz15eZDaddduPS7epVOervb73+cqvAfaSpJDw/0="
* Connection #0 to host login.salesforce.com left intact
}~$
```

Now enter your cURL command, which will be similar to the following, to call your Apex REST service and return the case information.

```
curl https://yourInstance.salesforce.com/services/apexrest/Cases/<Record_ID> -H 'Authorization: Bearer <your_session_id>' -H 'X-PrettyPrint:1'
```

After pressing Enter, you see something similar to the following. Now that you are a command-line master, feel free to cURL, jq, sed, awk, and grep to your heart's content. For more info on cURL, see the Resources section.



Update Data with a Custom PUT or PATCH Method

You can update records with the PUT or PATCH HTTP methods. The PUT method either updates the entire resource, if it exists, or creates the resource if it doesn't exist. PUT is essentially an upsert method. The PATCH method updates only the specified portions of an existing resource. In Apex, update operations update only the specified fields and don't overwrite the entire record. We'll write some Apex code to determine whether our methods update or upsert.

Update Data with the PUT Method

The `upsertCase` method that you added to the `CaseManager` class implements the PUT action. This method is included here for your reference. The method uses the built-in `upsert` Apex DML method to either create or overwrite case record fields by matching the ID value. If an ID is sent in the body of the request, the case sObject is populated with it. Otherwise, the case sObject is created without an ID. The `upsert` method is invoked with the populated case sObject, and the DML statement does the rest. Voila!

```
@HttpPut
global static ID upsertCase(String subject, String status,
    String origin, String priority, String id) {
    Case thisCase = new Case(
        Id=id,
        Subject=subject,
        Status=status,
        Origin=origin,
        Priority=priority);
    // Match case by Id, if present.
    // Otherwise, create new case.
    upsert thisCase;
    // Return the case ID.
    return thisCase.Id;
}
```

To invoke the PUT method:

1. In Workbench REST Explorer, select **PUT**.
2. For the URI, enter `/services/apexrest/Cases/`.
3. The `upsertCase` method expects the field values to be passed in the request body. Add the following for the request body, and then replace <*Record ID*> with the ID of the case record you created earlier.

```
{
  "id": "<Record_ID>",
  "status" : "Working",
  "subject" : "Bigfoot Sighting!",
  "priority" : "Medium"
}
```

Note

The ID field is optional. To create a case record, omit this field. In our example, you're passing this field because you want to update the case record.

4. Click **Execute**.

   This request invokes the `upsertCase` method from your REST service. The Status, Subject, and Priority fields are updated. The subject is updated, even though its value matches the old subject. Also, because the request body didn't contain a value for the Case Origin field, the origin parameter in the `upsertCase` method is null. As a result, when the record is updated, the Origin field is cleared.

   To check these fields, view this record in Salesforce by navigating to https://*yourInstance*.salesforce.com/<*Record ID*>.

Update Data with the PATCH Method

As an alternative to the PUT method, use the PATCH method to update record fields. You can implement the PATCH method in different ways. One way is to specify parameters in the method for each field to update. For example, you can create a method to update the priority of a case with this signature: `updateCasePriority(String priority)`. To update multiple fields, you can list all the desired fields as parameters.

Another approach that provides more flexibility is to pass the fields as JSON name/value pairs in the request body. That way the method can accept an arbitrary number of parameters, and the parameters aren't fixed in the method's signature. Another advantage of this approach is that no field is accidentally cleared because of being null. The `updateCaseFields` method that you added to the `CaseManager` class uses this second approach. This method deserialize the JSON string from the request body into a map of name/value pairs and uses the sObject `PUT` method to set the fields.

```
@HttpPatch
global static ID updateCaseFields() {
    RestRequest request = RestContext.request;
    String caseId = request.requestURI.substring(
        request.requestURI.lastIndexOf('/')+1);
    Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
    // Deserialize the JSON string into name-value pairs
    Map<String, Object> params = (Map<String, Object>)JSON.deserializeUntyped(request.requestbody.tostring());
    // Iterate through each parameter field and value
    for(String fieldName : params.keySet()) {
        // Set the field and value on the Case sObject
        thisCase.put(fieldName, params.get(fieldName));
    }
    update thisCase;
    return thisCase.Id;
}
```

To invoke the PATCH method:

1. In Workbench REST Explorer, click **PATCH**.
2. For the URI, enter `/services/apexrest/Cases/<Record ID>`. Replace <*Record ID*> with the ID of the case record created earlier. Enter the following JSON in the Request Body.

```
{
  "status" : "Escalated",
  "priority" : "High"
}
```

   This JSON has two field values: status and priority. The `updateCaseFields` method retrieves these values from the submitted JSON and are used to specify the fields to update in the object.

3. Click **Execute**.

   This request invokes the `updateCaseFields` method in your REST service. The Status and Priority fields of the case record are updated to new values. To check these fields, view this record in Salesforce by navigating to https://*yourInstance*.salesforce.com/<*Record ID*>.

Test Your Apex REST Class

Testing your Apex REST class is similar to testing any other Apex class—just call the class methods by passing in parameter values and then verify the results. For methods that don't take parameters or that rely on information in the REST request, create a test REST request.

In general, here's how you test Apex REST services. To simulate a REST request, create a `RestRequest` in the test method, and then set properties on the request as follows. You can also add params that you "pass" in the request to simulate URI parameters.

```
// Set up a test request
RestRequest request = new RestRequest();

// Set request properties
request.requestUri =
    'https://yourInstance.salesforce.com/services/apexrest/Cases/'
    + recordId;
request.httpMethod = 'GET';

// Set other properties, such as parameters
request.params.put('status', 'Working');

// more awesome code here....
// Finally, assign the request to RestContext if used
RestContext.request = request;
```

If the method you're testing accesses request values through `RestContext`, assign the request to `RestContext` to populate it (`RestContext.request = request;`).

Now, let's save the entire class in the Developer Console and run the results.

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

```
@IsTest
private class CaseManagerTest {

    @isTest static void testGetCaseById() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'GET';
        RestContext.request = request;
        // Call the method to test
        Case thisCase = CaseManager.getCaseById();
        // Verify results
        System.assert(thisCase != null);
        System.assertEquals('Test record', thisCase.Subject);
    }

    @isTest static void testCreateCase() {
        // Call the method to test
        ID thisCaseId = CaseManager.createCase(
            'Ferocious chipmunk', 'New', 'Phone', 'Low');
        // Verify results
        System.assert(thisCaseId != null);
        Case thisCase = [SELECT Id,Subject FROM Case WHERE Id=:thisCaseId];
        System.assert(thisCase != null);
```

```
        System.assertEquals(thisCase.Subject, 'Ferocious chipmunk');
    }

    @isTest static void testDeleteCase() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'GET';
        RestContext.request = request;
        // Call the method to test
        CaseManager.deleteCase();
        // Verify record is deleted
        List<Case> cases = [SELECT Id FROM Case WHERE Id=:recordId];
        System.assert(cases.size() == 0);
    }

    @isTest static void testUpsertCase() {
        // 1. Insert new record
        ID case1Id = CaseManager.upsertCase(
                'Ferocious chipmunk', 'New', 'Phone', 'Low', null);
        // Verify new record was created
        System.assert(Case1Id != null);
        Case case1 = [SELECT Id,Subject FROM Case WHERE Id=:case1Id];
        System.assert(case1 != null);
        System.assertEquals(case1.Subject, 'Ferocious chipmunk');
        // 2. Update status of existing record to Working
        ID case2Id = CaseManager.upsertCase(
                'Ferocious chipmunk', 'Working', 'Phone', 'Low', case1Id);
        // Verify record was updated
        System.assertEquals(case1Id, case2Id);
        Case case2 = [SELECT Id,Status FROM Case WHERE Id=:case2Id];
        System.assert(case2 != null);
        System.assertEquals(case2.Status, 'Working');
    }

    @isTest static void testUpdateCaseFields() {
        Id recordId = createTestRecord();
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'PATCH';
        request.addHeader('Content-Type', 'application/json');
        request.requestBody = Blob.valueOf('{"status": "Working"}');
        RestContext.request = request;
        // Update status of existing record to Working
        ID thisCaseId = CaseManager.updateCaseFields();
        // Verify record was updated
        System.assert(thisCaseId != null);
        Case thisCase = [SELECT Id,Status FROM Case WHERE Id=:thisCaseId];
        System.assert(thisCase != null);
        System.assertEquals(thisCase.Status, 'Working');
    }

    // Helper method
    static Id createTestRecord() {
        // Create test record
        Case caseTest = new Case(
```

```
            Subject='Test record',
            Status='New',
            Origin='Phone',
            Priority='Medium');
        insert caseTest;
        return caseTest.Id;
    }

}
```

4. Press `CTRL+S` to save.
5. Run all the tests in your org by selecting **Test | Run All**.

The test results display in the Tests tab. After the test execution finishes, check the CaseManager row in the Overall Code Coverage pane. It's at 100% coverage.

Tell Me More ...

Learn about supported data types and namespaces in Apex REST, Salesforce APIs, and security considerations.

Supported Data Types for Apex REST

Apex REST supports these data types for parameters and return values.

- Apex primitives (excluding sObject and Blob).
- sObjects
- Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).
- User-defined types that contain member variables of the types listed above.

Namespaces in Apex REST Endpoints

Apex REST methods can be used in managed and unmanaged packages. When calling Apex REST methods that are contained in a managed package, you need to include the managed package namespace in the REST call URL. For example, if the class is contained in a managed package namespace called `packageNamespace` and the Apex REST methods use a URL mapping of /MyMethod/*, the URL used via REST to call these methods would be of the form https://*instance*.salesforce.com/services/apexrest/packageNamespace/MyMethod/.

Custom Apex Web Services and Salesforce APIs

Instead of using custom Apex code for REST and SOAP services, external applications can integrate with Salesforce by using Salesforce's REST and SOAP APIs. These APIs let you create, update, and delete records. However, the advantage of using Apex web services is that Apex methods can encapsulate complex logic. This logic is hidden from the consuming application. Also, the Apex class operations can be faster than making individual API calls, because fewer roundtrips are performed between the client and the Salesforce servers. With an Apex web service call, there is only one request sent, and all operations within the method are performed on the server.

Security Considerations for Apex Web Services

The security context under which Apex web service methods run differs from the security context of Salesforce APIs. Unlike Salesforce APIs, Apex web service methods run with system privileges and don't respect the user's object and field permissions. However, Apex web service methods enforce sharing rules when declared with the `with sharing` keyword.

Resources

- *Apex Developer Guide*: Introduction to Apex REST
- *Apex Developer Guide*: Exposing Apex Classes as REST Web Services
- *Apex Developer Guide*: Exposing Apex Methods as SOAP Web Services
- *Force.com REST API Developer Guide*: Using cURL in the REST Examples
- Creating REST APIs Using Apex REST
- RFC7231 - Information about HTTP 1.1 (including request methods and responses)
- *Wikipedia*: Representational state transfer
- *Salesforce Developers*: REST API
- cURL Documentation

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

Create an Apex REST service that returns an account and it's contacts.

To pass this challenge, create an Apex REST class that is accessible at '/Accounts/<Account_ID>/contacts'. The service will return the account's ID and Name plus the ID and Name of all contacts associated with the account. Write unit tests that achieve 100% code coverage for the class and run your Apex tests.

- The Apex class must be called 'AccountManager'.
- The Apex class must have a method called 'getAccount' that is annotated with @HttpGet
- The method must return the ID and Name for the requested record and all associated contacts with their ID and Name.
- The unit tests must be in a separate Apex class called 'AccountManagerTest'.
- The unit tests must cover all lines of code included in the AccountManager 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

Time Estimate

About **50** mins

Topics

- Learning Objectives
- Expose Your Apex Class as a Web Service
- Apex REST Walkthrough
- Create a Record with a POST Method
- Retrieve Data with a Custom GET Method
- Update Data with a Custom PUT or PATCH Method
- Test Your Apex REST Class
- Tell Me More ...
- Resources
-
- Challenge +500 points
-

Have questions about Trailhead or having problems using it?

Trailhead Forum