**Data Types Overview**

Apex supports various data types, including a data type specific to Salesforce—the sObject data type.

Apex supports the following data types.

A primitive, such as an Integer, Double, Long, Date, Datetime, String, ID, Boolean, among others.

An sObject, either as a generic sObject or as a specific sObject, such as an Account, Contact, or MyCustomObject\_\_c (you’ll learn more about sObjects in a later unit.)

A collection, including:

A list (or array) of primitives, sObjects, user defined objects, objects created from Apex classes, or collections

A set of primitives

A map from a primitive to a primitive, sObject, or collection

A typed list of values, also known as an *enum*

User-defined Apex classes

System-supplied Apex classes

Apex Collections: List

Lists hold an ordered collection of objects. Lists in Apex are synonymous with arrays and the two can be used interchangeably.

The following two declarations are equivalent. The colors variable is declared using the List syntax.

List<String> colors = new List<String>();

Alternatively, the colors variable can be declared as an array but assigned to a list rather than an array.

String[] colors = new List<String>();

Grow collections as needed by using the List.add() method to add new elements. Use the square bracket array notation to reference existing elements in the collection by index. You can't, however, use square bracket array notation to add more elements.

This example shows how to add elements to a list when you create it, and then use the add() method to add more elements.

// Create a list and add elements to it in one step

List<String> colors = new List<String> { 'red', 'green', 'blue' };

// Add elements to a list after it has been created

List<String> moreColors = new List<String>();

moreColors.add('orange');

moreColors.add('purple');

List elements can be read by specifying an index between square brackets, just like with array elements. Also, you can use the get() method to read a list element. This example is based on the lists created in the previous example and shows how to read list elements using either method. The example also shows how to iterate over array elements.

// Get elements from a list

String color1 = moreColors.get(0);

String color2 = moreColors[0];

System.assertEquals(color1, color2);

// Iterate over a list to read elements

for(Integer i=0;i<colors.size();i++) {

// Write value to the debug log

System.debug(colors[i]);

}

Apex Classes

One of the benefits of Apex classes is code reuse. Class methods can be called by triggers and other classes. The following tutorial walks you through saving an example class in your organization, using this class to send emails, and inspecting debug logs.

**Save an Apex Class**

Save the EmailManager class in your organization:

Open the Developer Console under Your Name or the quick access menu (Setup gear icon).

In the Developer Console, click **File** | **New** | **Apex Class**, and enter EmailManager for the class name, and then click **OK**.

Replace the default class body with the EmailManager class example.  
  
The EmailManager class has a public method (sendMail()) that sends email and uses built-in Messaging methods of the Apex class library. Also, this class has a private helper method (inspectResults()), which can’t be called externally because it is private but is used only within the class. This helper method inspects the results of the email send call and is called by sendMail().

public class EmailManager {

// Public method

public void sendMail(String address, String subject, String body) {

// Create an email message object

Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

String[] toAddresses = new String[] {address};

mail.setToAddresses(toAddresses);

mail.setSubject(subject);

mail.setPlainTextBody(body);

// Pass this email message to the built-in sendEmail method

// of the Messaging class

Messaging.SendEmailResult[] results = Messaging.sendEmail(

new Messaging.SingleEmailMessage[] { mail });

// Call a helper method to inspect the returned results

inspectResults(results);

}

// Helper method

private static Boolean inspectResults(Messaging.SendEmailResult[] results) {

Boolean sendResult = true;

// sendEmail returns an array of result objects.

// Iterate through the list to inspect results.

// In this class, the methods send only one email,

// so we should have only one result.

for (Messaging.SendEmailResult res : results) {

if (res.isSuccess()) {

System.debug('Email sent successfully');

}

else {

sendResult = false;

System.debug('The following errors occurred: ' + res.getErrors());

}

}

return sendResult;

}

}

<https://help.salesforce.com/HTViewHelpDoc?id=code_dev_console_view_system_log.htm&language=en_US>

<https://www.youtube.com/watch?v=WBeCWlbGX38>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_intro.htm>

<https://trailhead.salesforce.com/content/learn/modules/developer_console>

object references in Apex are actually sObject versions of the Object created in Salesforce. sObject is an abstraction of the original.

Standard objects are referenced as expected:

Account acct = new Account(Name='Acme', Phone='(415)555-1212', NumberOfEmployees=100);

Custom:

Merchandise corresponds to the Merchandise\_\_c sObject in Apex

or custom objects and custom fields, the API name always ends with the \_\_c suffix. For custom relationship fields, the API name ends with the \_\_r suffix. For example:

A custom object with a label of Merchandise has an API name of Merchandise\_\_c.

A custom field with a label of Description has an API name of Description\_\_c.

A custom relationship field with a label of Items has an API name of Items\_\_r.

In addition, spaces in labels are replaced with underscores in API names. For example, a custom field name of Employee Seniority has an API name of Employee\_Seniority\_\_c.

<https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/>

sObject sobj1 = new Account(Name='Trailhead');

sObject sobj2 = new Book\_\_c(Name='Workbook 1');

// Cast a generic sObject to an Account

Account acct = (Account)myGenericSObject;

// Now, you can use the dot notation to access fields on Account

String name = acct.Name;

String phone = acct.Phone;

<https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/langCon_apex_SObjects.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_methods_system_sobject.htm#apex_System_SObject_methods>

DML = Data Manipulation Language

insert

update

upsert

delete

undelete

merge

// Create the account sObject

Account acct = new Account(Name='Acme', Phone='(415)555-1212', NumberOfEmployees=100);

// Insert the account by using DML

insert acct;

// Get the new ID on the inserted sObject argument

ID acctID = acct.Id;

// Display this ID in the debug log

System.debug('ID = ' + acctID);

// Debug log result (the ID will be different in your case)

// DEBUG|ID = 001D000000JmKkeIAF

// Create a list of contacts

List<Contact> conList = new List<Contact> {

new Contact(FirstName='Joe',LastName='Smith',Department='Finance'),

new Contact(FirstName='Kathy',LastName='Smith',Department='Technology'),

new Contact(FirstName='Caroline',LastName='Roth',Department='Finance'),

new Contact(FirstName='Kim',LastName='Shain',Department='Education')};

// Bulk insert all contacts with one DML call

insert conList;

// List to hold the new contacts to update

List<Contact> listToUpdate = new List<Contact>();

// Iterate through the list and add a title only

// if the department is Finance

for(Contact con : conList) {

if (con.Department == 'Finance') {

con.Title = 'Financial analyst';

// Add updated contact sObject to the list.

listToUpdate.add(con);

}

}

// Bulk update all contacts with one DML call

update listToUpdate;

========================

upsert sObjectList Account.Fields.MyExternalId;

========================================

Contact jane = new Contact(FirstName='Jane',

LastName='Smith',

Email='jane.smith@example.com',

Description='Contact of the day');

insert jane;

// 1. Upsert using an idLookup field

// Create a second sObject variable.

// This variable doesn’t have any ID set.

Contact jane2 = new Contact(FirstName='Jane',

LastName='Smith',

Email='jane.smith@example.com',

Description='Prefers to be contacted by email.');

// Upsert the contact by using the idLookup field for matching.

upsert jane2 Contact.fields.Email;

// Verify that the contact has been updated

System.assertEquals('Prefers to be contacted by email.',

[SELECT Description FROM Contact WHERE Id=:jane.Id].Description);

========================================

Contact[] contactsDel = [SELECT Id FROM Contact WHERE LastName='Smith'];

delete contactsDel;

DML Exceptions:

try {

// This causes an exception because

// the required Name field is not provided.

Account acct = new Account();

// Insert the account

insert acct;

} catch (DmlException e) {

System.debug('A DML exception has occurred: ' +

e.getMessage());

}

Database class performs DML operations.

Static

Have optional allOrNone parm (should it partially succeed? Default is true)

Database.insert()

Database.update()

Database.upsert()

Database.delete()

Database.undelete()

Database.merge()

Database.insert(recordList, false);

Database.SaveResult[] results = Database.insert(recordList, false);

Should You Use DML Statements or Database Methods?

Use DML statements if you want any error that occurs during bulk DML processing to be thrown as an Apex exception that immediately interrupts control flow (by using try. . .catch blocks). This behavior is similar to the way exceptions are handled in most database procedural languages.

Use Database class methods if you want to allow partial success of a bulk DML operation—if a record fails, the remainder of the DML operation can still succeed. Your application can then inspect the rejected records and possibly retry the operation. When using this form, you can write code that never throws DML exception errors. Instead, your code can use the appropriate results array to judge success or failure. Note that Database methods also include a syntax that supports thrown exceptions, similar to DML statements.

Related Objects

Account acct = new Account(Name='SFDC Account');

insert acct;

// Once the account is inserted, the sObject will be

// populated with an ID.

// Get this ID.

ID acctID = acct.ID;

// Add a contact to this account.

Contact mario = new Contact(

FirstName='Mario',

LastName='Ruiz',

Phone='415.555.1212',

AccountId=acctID);

insert mario;

============================

// Query for the contact, which has been associated with an account.

Contact queriedContact = [SELECT Account.Name

FROM Contact

WHERE FirstName = 'Mario' AND LastName='Ruiz'

LIMIT 1];

// Update the contact's phone number

queriedContact.Phone = '(415)555-1213';

// Update the related account industry

queriedContact.Account.Industry = 'Technology';

// Make two separate calls

// 1. This call is to update the contact's phone.

update queriedContact;

// 2. This call is to update the related account's Industry field.

update queriedContact.Account;

Delete

Account[] queriedAccounts = [SELECT Id FROM Account WHERE Name='SFDC Account'];

delete queriedAccounts;

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_data_intro.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_dml_section.htm>

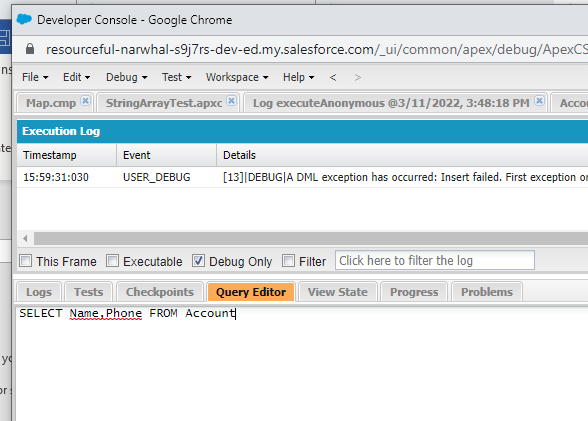
<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_methods_system_database.htm>

SOSQL

Account[] accts = [SELECT Name,Phone FROM Account];

========================

Developer Console > Query Editor



‘\*’ Does not work – need to list all desired fields

Account[] accts = [SELECT Name,Phone FROM Account

WHERE (Name='SFDC Computing' AND NumberOfEmployees>25)

ORDER BY Name

LIMIT 10];

System.debug(accts.size() + ' account(s) returned.');

// Write all account array info

System.debug(accts);

Variables:

String targetDepartment = 'Wingo';

Contact[] techContacts = [SELECT FirstName,LastName

FROM Contact WHERE Department=:targetDepartment];

Child records

SELECT Name, (SELECT LastName FROM Contacts) FROM Account WHERE Name = 'SFDC Computing'

Another Example:

Account[] acctsWithContacts = [SELECT Name, (SELECT FirstName,LastName FROM Contacts)

FROM Account

WHERE Name = 'SFDC Computing'];

// Get child records

Contact[] cts = acctsWithContacts[0].Contacts;

System.debug('Name of first associated contact: '

+ cts[0].FirstName + ', ' + cts[0].LastName);

Traverse relationships:

Contact[] cts = [SELECT Account.Name FROM Contact

WHERE FirstName = 'Carol' AND LastName='Ruiz'];

Contact carol = cts[0];

String acctName = carol.Account.Name;

System.debug('Carol\'s account name is ' + acctName);

Using for loop:

insert new Account[]{new Account(Name = 'for loop 1'),

new Account(Name = 'for loop 2'),

new Account(Name = 'for loop 3')};

// The sObject list format executes the for loop once per returned batch

// of records

Integer i=0;

Integer j=0;

for (Account[] tmp : [SELECT Id FROM Account WHERE Name LIKE 'for loop \_']) {

j = tmp.size();

i++;

}

System.assertEquals(3, j); // The list should have contained the three accounts

// named 'yyy'

System.assertEquals(1, i); // Since a single batch can hold up to 200 records and,

// only three records should have been returned, the

// loop should have executed only once

<https://developer.salesforce.com/docs/atlas.en-us.224.0.soql_sosl.meta/soql_sosl/>

SOSL = Salesforce Object Search Language

SOQL can only query 1 object at a time

SOSL can search all objects

SOQL-exact match(unless you use a wildcard), SOSL-search sting is contained in the field (%searchval%)

List<List<SObject>> searchList = [FIND 'SFDC' IN ALL FIELDS

RETURNING Account(Name), Contact(FirstName,LastName)];

You can execute it in the Query Editor

FIND {Wingo} IN ALL FIELDS RETURNING Account(Name), Contact(FirstName,LastName,Department)

Above would return any object with a field containing ‘Wingo’

Basic SOSL Syntax

SOSL allows you to specify the following search criteria:

Text expression (single word or a phrase) to search for

Scope of fields to search

List of objects and fields to retrieve

Conditions for selecting rows in the source objects

This is the syntax of a basic SOSL query:

FIND 'SearchQuery' [IN SearchGroup] [RETURNING ObjectsAndFields]

*SearchQuery* is the text to search for (a single word or a phrase). Search terms can be grouped with logical operators (AND, OR) and parentheses. Also, search terms can include wildcard characters (\*, ?). The \* wildcard matches zero or more characters at the middle or end of the search term. The ? wildcard matches only one character at the middle or end of the search term.

Text searches are case-insensitive. For example, searching for Customer, customer, or CUSTOMER all return the same results.

*SearchGroup* is optional. It is the scope of the fields to search. If not specified, the default search scope is all fields. *SearchGroup* can take one of the following values.

ALL FIELDS

NAME FIELDS

EMAIL FIELDS

PHONE FIELDS

SIDEBAR FIELDS

*ObjectsAndFields* is optional. It is the information to return in the search result—a list of one or more sObjects and, within each sObject, list of one or more fields, with optional values to filter against. If not specified, the search results contain the IDs of all objects found.

Single Words and Phrases

A *SearchQuery* contains two types of text:

Single Word— single word, such as test or hello. Words in the SearchQuery are delimited by spaces, punctuation, and changes from letters to digits (and vice-versa). Words are always case insensitive.

Phrase— collection of words and spaces surrounded by double quotes such as "john smith". Multiple words can be combined together with logic and grouping operators to form a more complex query.

Search Examples

To learn about how SOSL search works, let’s play with different search strings and see what the output is based on our sample data. This table lists various example search strings and the SOSL search results.

| Search in all fields for: | Search Description | Matched Records and Fields |
| --- | --- | --- |
| The Query | This search returns all records whose fields contain both words: The and Query, in any location of the text. The order of words in the search term doesn’t matter. | Account: The SFDC Query Man (Name field matched) |
| Wingo OR Man | This search uses the OR logical operator. It returns records with fields containing the word Wingo or records with fields containing the word Man. | Contact: Carol Ruiz, Department: 'Wingo'  Account: The SFDC Query Man (Name field matched) |
| 1212 | This search returns all records whose fields contain the word 1212. Phone fields that end with -1212 are matched because 1212 is considered a word when delimited by the dash. | Account: The SFDC Query Man, Phone: '(415)555-1212'  Contact: Carol Ruiz, Phone: '(415)555-1212' |
| wing\* | This is a wildcard search. This search returns all records that have a field value starting with wing. | Contact: Maria Ruiz, Department: 'Wingo'  Account: The SFDC Query Man, Description: 'Expert in wing technologies.' |

Example:

String soslFindClause = 'Wingo OR SFDC';

List<List<sObject>> searchList = [FIND :soslFindClause IN ALL FIELDS

RETURNING Account(Name),Contact(FirstName,LastName,Department)];

Account[] searchAccounts = (Account[])searchList[0];

Contact[] searchContacts = (Contact[])searchList[1];

System.debug('Found the following accounts.');

for (Account a : searchAccounts) {

System.debug(a.Name);

}

System.debug('Found the following contacts.');

for (Contact c : searchContacts) {

System.debug(c.LastName + ', ' + c.FirstName);

}

<https://developer.salesforce.com/docs/atlas.en-us.224.0.soql_sosl.meta/soql_sosl/>

Apex Triggers:

Perform custom actions before or after database record events.

trigger HelloWorldTrigger on Account (before insert) {

System.debug('Hello World!');

}

before insert

before update

before delete

after insert

after update

after delete

after undelete

To access the records that caused the trigger to fire, use context variables. For example, Trigger.New contains all the records that were inserted in insert or update triggers. Trigger.Old provides the old version of sObjects before they were updated in update triggers, or a list of deleted sObjects in delete triggers. Triggers can fire when one record is inserted, or when many records are inserted in bulk via the API or Apex. Therefore, context variables, such as Trigger.New, can contain only one record or multiple records. You can iterate over Trigger.New to get each individual sObject.

trigger HelloWorldTrigger on Account (before insert) {

for(Account a : Trigger.New) {

a.Description = 'New description';

}

}

Another Example:

trigger ContextExampleTrigger on Account (before insert, after insert, after delete) {

if (Trigger.isInsert) {

if (Trigger.isBefore) {

// Process before insert

} else if (Trigger.isAfter) {

// Process after insert

}

}

else if (Trigger.isDelete) {

// Process after delete

}

}

| Variable | Usage |
| --- | --- |
| isExecuting | Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an executeanonymous() API call. |
| isInsert | Returns true if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API. |
| isUpdate | Returns true if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API. |
| isDelete | Returns true if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API. |
| isBefore | Returns true if this trigger was fired before any record was saved. |
| isAfter | Returns true if this trigger was fired after all records were saved. |
| isUndelete | Returns true if this trigger was fired after a record is recovered from the Recycle Bin. This recovery can occur after an undelete operation from the Salesforce user interface, Apex, or the API. |
| new | Returns a list of the new versions of the sObject records.  This sObject list is only available in insert, update, and undelete triggers, and the records can only be modified in before triggers. |
| newMap | A map of IDs to the new versions of the sObject records.  This map is only available in before update, after insert, after update, and after undelete triggers. |
| old | Returns a list of the old versions of the sObject records.  This sObject list is only available in update and delete triggers. |
| oldMap | A map of IDs to the old versions of the sObject records.  This map is only available in update and delete triggers. |
| operationType | Returns an enum of type System.TriggerOperation corresponding to the current operation.  Possible values of the System.TriggerOperation enum are: BEFORE\_INSERT, BEFORE\_UPDATE, BEFORE\_DELETE,AFTER\_INSERT, AFTER\_UPDATE, AFTER\_DELETE, and AFTER\_UNDELETE. If you vary your programming logic based on different trigger types, consider using the switch statement with different permutations of unique trigger execution enum states. |
| size | The total number of records in a trigger invocation, both old and new. |

Example to add a related record:

trigger AddRelatedRecord on Account(after insert, after update) {

List<Opportunity> oppList = new List<Opportunity>();

// Get the related opportunities for the accounts in this trigger

Map<Id,Account> acctsWithOpps = new Map<Id,Account>(

[SELECT Id,(SELECT Id FROM Opportunities) FROM Account WHERE Id IN :Trigger.New]);

// Add an opportunity for each account if it doesn't already have one.

// Iterate through each account.

for(Account a : Trigger.New) {

System.debug('acctsWithOpps.get(a.Id).Opportunities.size()=' + acctsWithOpps.get(a.Id).Opportunities.size());

// Check if the account already has a related opportunity.

if (acctsWithOpps.get(a.Id).Opportunities.size() == 0) {

// If it doesn't, add a default opportunity

oppList.add(new Opportunity(Name=a.Name + ' Opportunity',

StageName='Prospecting',

CloseDate=System.today().addMonths(1),

AccountId=a.Id));

}

}

if (oppList.size() > 0) {

insert oppList;

}

}

Error handling (.addError)

trigger AccountDeletion on Account (before delete) {

// Prevent the deletion of accounts if they have related opportunities.

for (Account a : [SELECT Id FROM Account

WHERE Id IN (SELECT AccountId FROM Opportunity) AND

Id IN :Trigger.old]) {

Trigger.oldMap.get(a.Id).addError(

'Cannot delete account with related opportunities.');

}

}

Example Http calls:

public class CalloutClass {

@future(callout=true)

public static void makeCallout() {

HttpRequest request = new HttpRequest();

// Set the endpoint URL.

String endpoint = 'http://yourHost/yourService';

request.setEndPoint(endpoint);

// Set the HTTP verb to GET.

request.setMethod('GET');

// Send the HTTP request and get the response.

HttpResponse response = new HTTP().send(request);

}

}

trigger CalloutTrigger on Account (before insert, before update) {

CalloutClass.makeCallout();

}

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_triggers.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_callouts.htm>

<https://trailhead.salesforce.com/modules/apex_integration_services>

Example SOQL:

trigger AccountAddressTrigger on Account (before insert, before update) {

for(Account a : Trigger.New) {

if(a.Match\_Billing\_Address\_\_c){

a.ShippingPostalCode = a.BillingPostalCode;

}

}

}

Bulk Triggers

trigger SoqlTriggerBulk on Account(after update) {

// Perform SOQL query once.

// Get the accounts and their related opportunities.

List<Account> acctsWithOpps =

[SELECT Id,(SELECT Id,Name,CloseDate FROM Opportunities)

FROM Account WHERE Id IN :Trigger.New];

// Iterate over the returned accounts

for(Account a : acctsWithOpps) {

Opportunity[] relatedOpps = a.Opportunities;

// Do some other processing

}

}

Example DML

trigger DmlTriggerBulk on Account(after update) {

// Get the related opportunities for the accounts in this trigger.

List<Opportunity> relatedOpps = [SELECT Id,Name,Probability FROM Opportunity

WHERE AccountId IN :Trigger.New];

List<Opportunity> oppsToUpdate = new List<Opportunity>();

// Iterate over the related opportunities

for(Opportunity opp : relatedOpps) {

// Update the description when probability is greater

// than 50% but less than 100%

if ((opp.Probability >= 50) && (opp.Probability < 100)) {

opp.Description = 'New description for opportunity.';

oppsToUpdate.add(opp);

}

}

// Perform DML on a collection

update oppsToUpdate;

}

To limit the triggers to records that meet a condition:

[SELECT Id,Name FROM Account WHERE Id IN :Trigger.New AND

Id NOT IN (SELECT AccountId FROM Opportunity)]

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_triggers.htm>

Example – create a task on Opportunity after creating or updating:

trigger ClosedOpportunityTrigger on Opportunity (after insert, after update) {

List<Task> taskList = new List<task>();

for (Opportunity opp : Trigger.new){

if(opp.StageName == 'Closed Won'){

taskList.add(new Task(Subject= 'Follow Up Test Task', WhatId = opp.Id));

}

}

if(taskList.size()>0) {

insert taskList;

}

}

APEX Unit Tests

Example:

@isTest

private class TemperatureConverterTest {

@isTest static void testWarmTemp() {

Decimal celsius = TemperatureConverter.FahrenheitToCelsius(70);

System.assertEquals(21.11,celsius);

}

@isTest static void testFreezingPoint() {

Decimal celsius = TemperatureConverter.FahrenheitToCelsius(32);

System.assertEquals(0,celsius);

}

@isTest static void testBoilingPoint() {

Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);

System.assertEquals(100,celsius,'Boiling point temperature is not expected.');

}

@isTest static void testNegativeTemp() {

Decimal celsius = TemperatureConverter.FahrenheitToCelsius(-10);

System.assertEquals(-23.33,celsius);

}

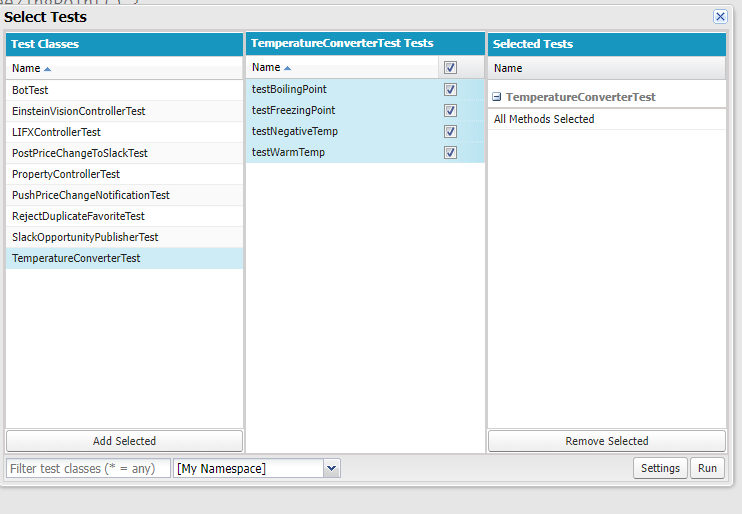
}

To execute a unit test:

Test > New Run

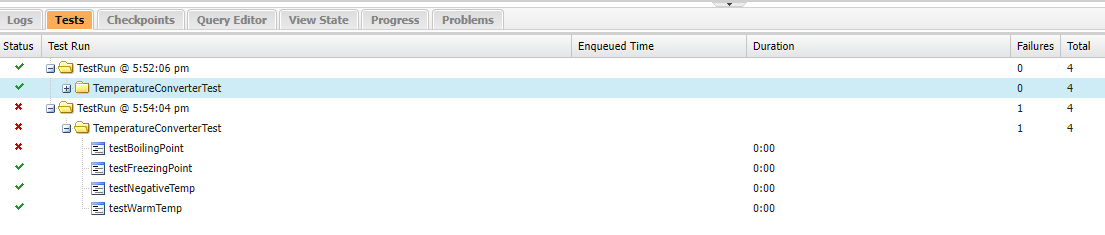
Select the test class

Select the methods



Run

Results



<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_testing_best_practices.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_testing_unit_tests_running.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_testing_data_access.htm>

<https://help.salesforce.com/apex/HTViewHelpDoc?id=code_dev_console_tests_coverage.htm&language=en_US>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_testing_tools_start_stop_test.htm>

@isTest

public class TestRestrictContactByName {

@isTest

public static void TestInvalidName() {

Contact c = new Contact(LastName='INVALIDNAME');

Database.SaveResult result = Database.insert(c, false);

System.assert(!result.isSuccess());

System.assert(result.getErrors().size() > 0);

System.assertEquals('The Last Name "'+c.LastName+'" is not allowed for DML',

result.getErrors()[0].getMessage());

}

}

The TestDataFactory class is a special type of class—it is a public class that is annotated with isTest and can be accessed only from a running test. Test utility classes contain methods that can be called by test methods to perform useful tasks, such as setting up test data.

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_testing_utility_classes.htm>

**External Service Integrations**

* Web service callouts to SOAP web services use XML, and typically require a WSDL document for code generation.
* HTTP callouts to services typically use REST with JSON.

Security:

You need to authorize the call in the Salesforce Remote Site Setings page

Setup > Remote Site Settings

<https://stormpath.com/blog/rest-vs-soap/>

<https://trailhead.salesforce.com/live/broadcasts/a2r3k000001vDCw/trail-together---apex-integration-services>

<https://trailhead.salesforce.com/content/learn/modules/apex_integration_services/apex_integration_rest_callouts?trail_id=force_com_dev_intermediate#resources>

**Sample code to call a service:**

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);

}

}

[JSON2Apex](https://json2apex.herokuapp.com/) – useful to parse complex json

**Sample POST:**

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());

}

**NOTE: Apex Test Methods do not support callouts**

Use a mock

@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.');

}

}

1. 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.

**To test a POST:**

@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;

}

}

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.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_callouts.htm>

Convert json into Apex code:

<https://json2apex.herokuapp.com/?_ga=2.76418645.1724540088.1647203886-1766590648.1646405561>

**Example: Parse without casting into an Object:**

JSON response looks like this:

{

    "animal": {

        "id": 1,

        "name": "chicken",

        "eats": "chicken food",

        "says": "cluck cluck"

    }

}

public class AnimalLocator {

public String getAnimalNameById(Integer id){

String animalName = '';

Http http = new Http();

HttpRequest request = new HttpRequest();

request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals/' + String.valueOf(id));

request.setMethod('GET');

request.setHeader('Content-Type', 'application/json;charset=UTF-8');

HttpResponse response = http.send(request);

// If the request is successful, parse the JSON response.

if(response.getStatusCode() == 200) {

Map<String, Object> results = (Map<String, Object>) JSON.deserializeUntyped(response.getBody());

System.debug('UAC: results ' + results );

Map<String, Object> item = (Map<String, Object>)results.get('animal') ;

System.debug('UAC: ' + (String) item.get('name'));

animalName = (String) item.get('name') ;

}

return animalName;

}

}

**Testing SUCKS!!!!!**

Create a Mock Response:

@isTest

global class AnimalLocatorMock 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(' {"animal":{"id":1,"name":"chicken","eats":"chicken food","says":"cluck cluck"}}');

response.setStatusCode(200);

return response;

}

}

Test that overrides the http call using the mock response looks like this:

@isTest

private class AnimalLocatorTest

{

static testMethod void testMethod1()

{

// Set mock callout class

Test.setMock(HttpCalloutMock.class, new AnimalLocatorMock());

// This causes a fake response to be sent

// from the class that implements HttpCalloutMock.

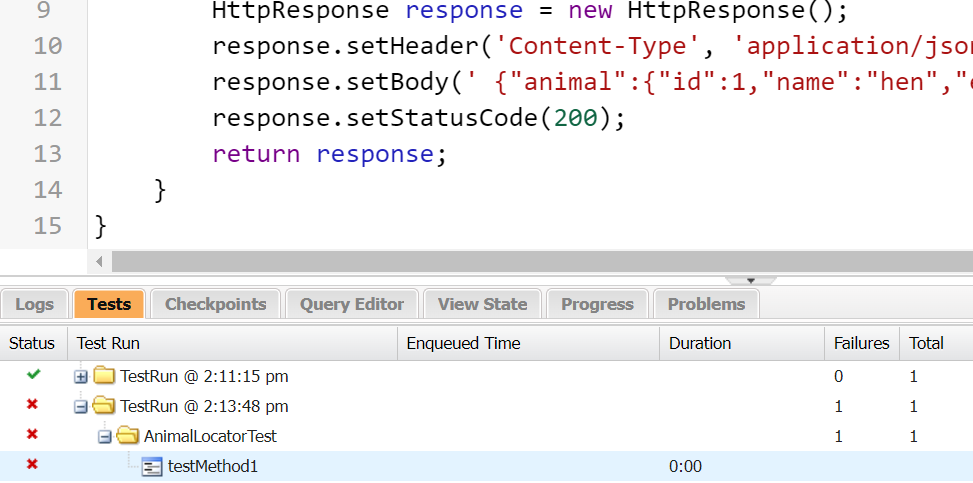
String str = AnimalLocator.getAnimalNameById(1);

System.assertEquals('chicken', str, 'The returned value should be chicken.');

}

}

Confirmed: when I changed the mock response to a different value (hen, actual service returns chicken) the test failed:



**SOAP**

Generate APEX from a WSDL

Setup > Apex Classes > Generate from WSDL > Choose File

Browse to file

Parse WSDL

Generate Apex Code

\*\*\*\*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.