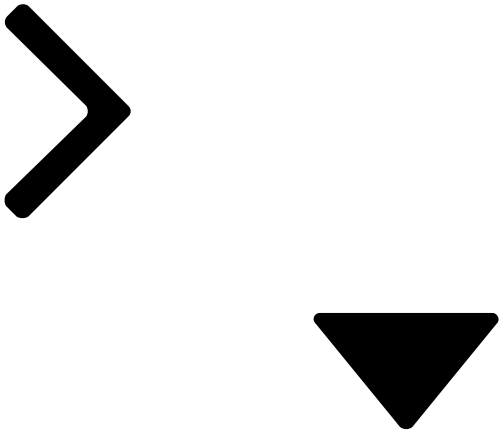
1. Apex Triggers



2. Get Started with Apex Triggers

Get Started with Apex Triggers

Learning Objectives

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

- Write a trigger for a Salesforce object.
- Use trigger context variables.
- Call a class method from a trigger.
- Use the sObject `addError()` method in a trigger to restrict save operations.

Writing Apex Triggers

Apex triggers enable you to perform custom actions before or after events to records in Salesforce, such as insertions, updates, or deletions. Just like database systems support triggers, Apex provides trigger support for managing records.

Typically, you use triggers to perform operations based on specific conditions, to modify related records or restrict certain operations from happening. You can use triggers to do anything you can do in Apex, including executing SOQL and DML or calling custom Apex methods.

Use triggers to perform tasks that can't be done by using the point-and-click tools in the Salesforce user interface. For example, if validating a field value or updating a field on a record, use validation rules and workflow rules instead.

Triggers can be defined for top-level standard objects, such as Account or Contact, custom objects, and some standard child objects. Triggers are active by default when created. Salesforce automatically fires active triggers when the specified database events occur.

Trigger Syntax

The syntax of a trigger definition is different from a class definition's syntax. A trigger definition starts with the `trigger` keyword. It is then followed by the name of the trigger, the Salesforce object that the trigger is associated with, and the conditions under which it fires. A trigger has the following syntax:

```
trigger TriggerName on ObjectName (trigger_events) {
   code_block
}
```

Copy

To execute a trigger before or after insert, update, delete, and undelete operations, specify multiple trigger events in a comma-separated list. The events you can specify are:

- `before insert`

- `before update`
- `before delete`
- `after insert`
- `after update`
- `after delete`
- `after undelete`

Example

This simple trigger fires before you insert an account and writes a message to the debug log.

1. In the Developer Console, click **File** | **New** | **Apex Trigger**.
2. Enter `HelloWorldTrigger` for the trigger name, and then select Account for the sObject. Click **Submit**.
3. Replace the default code with the following.

```
trigger HelloWorldTrigger on Account (before insert) {
       System.debug('Hello World!');
}
```

Copy

4. To save, press **Ctrl+S**.
5. To test the trigger, create an account.
    a. Click **Debug** | **Open Execute Anonymous Window**.
    b. In the new window, add the following and then click **Execute**.

```
Account a = new Account(Name='Test Trigger');
insert a;
```

Copy

6. In the debug log, find the `Hello World!` statement. The log also shows that the trigger has been executed.

Types of Triggers

There are two types of triggers.

- *Before triggers* are used to update or validate record values before they're saved to the database.
- *After triggers* are used to access field values that are set by the system (such as a record's `Id` or `LastModifiedDate` field), and to affect changes in other records. The records that fire the *after trigger* are read-only.

Using Context Variables

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.

This example is a modified version of the `HelloWorldTrigger` example trigger. It iterates over each account in a for loop and updates the Description field for each.

```
trigger HelloWorldTrigger on Account (before insert) {
    for(Account a : Trigger.New) {
        a.Description = 'New description';
    }
}
```

Copy

Note

The system saves the records that fired the before trigger after the trigger finishes execution. You can modify the records in the trigger without explicitly calling a DML insert or update operation. If you perform DML statements on those records, you get an error.

Some other context variables return a Boolean value to indicate whether the trigger was fired due to an update or some other event. These variables are useful when a trigger combines multiple events. For 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
    }
}
```

Copy

Trigger Context Variables

The following table is a comprehensive list of all context variables available for triggers.

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.

Variable	Usage
`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.

Calling a Class Method from a Trigger

You can call public utility methods from a trigger. Calling methods of other classes enables code reuse, reduces the size of your triggers, and improves maintenance of your Apex code. It also allows you to use object-oriented programming.

The following example trigger shows how to call a static method from a trigger. If the trigger was fired because of an insert event, the example calls the static `sendMail()` method on the `EmailManager` class. This utility method sends an email to the specified recipient and contains the number of contact records inserted.

Note

The `EmailManager` class is included in the class example of the Get Started with Apex unit. You must have saved the `EmailManager` class in your org and changed the `sendMail()` method to static before saving this trigger.

1. In the Developer Console, click **File** | **New** | **Apex Trigger**.
2. Enter `ExampleTrigger` for the trigger name, and then select **Contact** for the sObject. Click **Submit**.
3. Replace the default code with the following, and then modify the email address placeholder text in `sendMail()` to your email address.

```
trigger ExampleTrigger on Contact (after insert, after delete) {
    if (Trigger.isInsert) {
        Integer recordCount = Trigger.New.size();
        // Call a utility method from another class
        EmailManager.sendMail('Your email address', 'Trailhead Trigger Tutorial',
                    recordCount + ' contact(s) were inserted.');
    }
    else if (Trigger.isDelete) {
        // Process after delete
    }
}
```

Copy

4. To save, press **Ctrl+S**.
5. To test the trigger, create a contact.
    a. Click **Debug** | **Open Execute Anonymous Window**.
    b. In the new window, add the following and then click **Execute**.

```
Contact c = new Contact(LastName='Test Contact');
insert c;
```

Copy

6. In the debug log, check that the trigger was fired. Toward the end of the log, find the debug message that was written by the utility method: `DEBUG|Email sent successfully`
7. Now check that you received an email with the body text `1 contact(s) were inserted.`

With your new trigger in place, you get an email every time you add one or more contacts!

Adding Related Records

Triggers are often used to access and manage records related to the records in the trigger context—the records that caused this trigger to fire.

This trigger adds a related opportunity for each new or updated account if no opportunity is already associated with the account. The trigger first performs a SOQL query to get all child opportunities for the accounts that the trigger fired on. Next, the trigger iterates over the list of sObjects in `Trigger.New` to get each account sObject. If the account doesn't have any related opportunity sObjects, the for loop creates one. If the trigger created any new opportunities, the final statement inserts them.

1. Add the following trigger using the Developer Console (follow the steps of the `HelloWorldTrigger` example but use `AddRelatedRecord` for the trigger name).

```
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;
    }
}
```

Copy

2. To test the trigger, create an account in the Salesforce user interface and name it `Apples & Oranges`.
3. In the Opportunities related list on the account's page, find the new opportunity. The trigger added this opportunity automatically!

Beyond the Basics

The trigger you've added iterates over all records that are part of the trigger context—the for loop iterates over `Trigger.New`. However, the loop in this trigger could be more efficient. We don't really need to access every account in this trigger context, but only a subset—the accounts without opportunities. The next unit shows how to make this trigger more efficient. In the Bulk Trigger Design Patterns unit, learn how to modify the SOQL query to get only the accounts with no opportunities. Then, learn to iterate only over those records.

Using Trigger Exceptions

You sometimes need to add restrictions on certain database operations, such as preventing records from being saved when certain conditions are met. To prevent saving records in a trigger, call the `addError()` method on the sObject in question. The `addError()` method throws a fatal error inside a trigger. The error message is displayed in the user interface and is logged.

The following trigger prevents the deletion of an account if it has related opportunities. By default, deleting an account causes a cascade delete of all its related records. This trigger prevents the cascade delete of opportunities. Try this trigger for yourself! If you've executed the previous example, your org has an account called Apples & Oranges with a related opportunity. This example uses that sample account.

1. Using the Developer Console, add the following trigger.

```
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.');
    }

}
```

Copy

2. In the Salesforce user interface, navigate to the `Apples & Oranges` account's page and click **Delete**.
3. In the confirmation popup, click **OK**.

   Find the validation error with the custom error message `Cannot delete account with related opportunities.`

4. Disable the `AccountDeletion` trigger. If you leave this trigger active, you can't check your challenges.
   a. From Setup, search for `Apex Triggers` .
   b. On the Apex Triggers page, click **Edit** next to the AccountDeletion trigger.
   c. Deselect **Is Active**.
   d. Click **Save**.

Beyond the Basics

Calling `addError()` in a trigger causes the entire set of operations to roll back, except when bulk DML is called with partial success.

- If a DML statement in Apex spawned the trigger, any error rolls back the entire operation. However, the runtime engine still processes every record in the operation to compile a comprehensive list of errors.
- If a bulk DML call in the Lightning Platform API spawned the trigger, the runtime engine sets the bad records aside. The runtime engine then attempts a partial save of the records that did not generate errors.

Triggers and Callouts

Apex allows you to make calls to and integrate your Apex code with external Web services. Apex calls to external Web services are referred to as callouts. For example, you can make a callout to a stock quote service to get the latest quotes. When making a callout from a trigger, the callout must be done asynchronously so that the trigger process doesn't block you from working while waiting for the external service's response.The asynchronous callout is made in a background process, and the response is received when the external service returns it.

To make a callout from a trigger, call a class method that executes asynchronously. Such a method is called a future method and is annotated with `@future(callout=true)`. This example class contains the future method that makes the callout.

Note

The example uses a hypothetical endpoint URL for illustration purposes only. You can't run this example unless you change the endpoint to a valid URL and add a remote site in Salesforce for your endpoint.

```
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);
    }
}
```

Copy

This example shows the trigger that calls the method in the class to make a callout asynchronously.

```
trigger CalloutTrigger on Account (before insert, before update) {
    CalloutClass.makeCallout();
}
```

Copy

This section offers only an overview of callouts and is not intended to cover callouts in detail. For more information, see Invoking Callouts Using Apex in the *Apex Developer Guide*.

Resources

- *Apex Developer Guide*
    - Triggers
    - Invoking Callouts Using Apex
- Apex Callouts

Get Ready

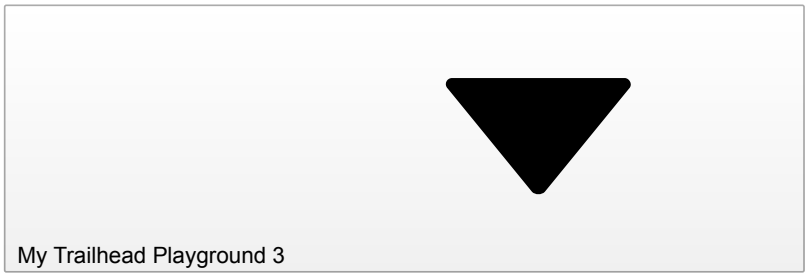
You'll be completing this challenge in your own personal Salesforce environment. Choose from the dropdown menu, then click **Launch** to get started. If you use Trailhead in a language other than English, set the language of your Trailhead Playground to English before you attempt this challenge. Want to find out more about using hands-on orgs for Trailhead learning? Check out the Trailhead Playground Management module.

Your Challenge

Create an Apex trigger for Account that matches Shipping Address Postal Code with Billing Address Postal Code based on a custom field.
For this challenge, you need to create a trigger that, before insert or update, checks for a checkbox, and if the checkbox field is true, sets the Shipping Postal Code (whose API name is ShippingPostalCode) to be the same as the Billing Postal Code

(BillingPostalCode).

- The Apex trigger must be called 'AccountAddressTrigger'.
- The Account object will need a new custom checkbox that should have the Field Label 'Match Billing Address' and Field Name of 'Match_Billing_Address'. The resulting API Name should be 'Match_Billing_Address__c'.
- With 'AccountAddressTrigger' active, if an Account has a Billing Postal Code and 'Match_Billing_Address__c' is true, the record should have the Shipping Postal Code set to match on insert or update.



Launch

Check challenge to earn 500 points