**Create a Custom Object**

In this step, you create a custom object called Book with one custom field called Price.

Prerequisites:

A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.

For more information about creating a sandbox org, see “Sandbox Types and Templates” in the Salesforce Help. To sign up for a free Developer org, see the [Developer Edition Environment Sign Up Page](http://developer.force.com/join).

1. Log in to your sandbox or Developer org.
2. From your management settings for custom objects, if you’re using Salesforce Classic, click **New Custom Object**, or if you’re using Lightning Experience, select **Create** | **Custom Object**.
3. Enter Book for the label.
4. Enter Books for the plural label.
5. Click **Save**.

Ta dah! You’ve now created your first custom object. Now let’s create a custom field.

1. In the **Custom Fields & Relationships** section of the Book detail page, click **New**.
2. Select Number for the data type and click **Next**.
3. Enter Price for the field label.
4. Enter 16 in the length text box.
5. Enter 2 in the decimal places text box, and click **Next**.
6. Click **Next** to accept the default values for field-level security.
7. Click **Save**.

**Adding an Apex Class**

In this step, you add an Apex class that contains a method for updating the book price. This method is called by the trigger that you will be adding in the next step.

Prerequisites:

* A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
* [The Book custom object](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_qs_customobject.htm).

1. From Setup, enter “Apex Classes” in the Quick Find box, then select **Apex Classes** and click **New**.
2. In the class editor, enter this class definition:

|  |  |  |
| --- | --- | --- |
| 1 | public class MyHelloWorld { | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | } |

1. The previous code is the class definition to which you will be adding one method in the next step. Apex code is generally contained in *classes.* This class is defined as public, which means the class is available to other Apex classes and triggers. For more information, see [Classes, Objects, and Interfaces](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes.htm).
2. Add this method definition between the class opening and closing brackets.

|  |  |  |
| --- | --- | --- |
| 1 | public static void applyDiscount(Book\_\_c[] books) { | |
| 2 | for (Book\_\_c b :books){ |

|  |  |  |
| --- | --- | --- |
| 3 | b.Price\_\_c \*= 0.9; | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |

1. This method is called applyDiscount, and it is both public and static. Because it is a static method, you don't need to create an instance of the class to access the method—you can just use the name of the class followed by a dot (.) and the name of the method. For more information, see [Static and Instance Methods, Variables, and Initialization Code](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_static.htm).
2. This method takes one parameter, a list of Book records, which is assigned to the variable books. Notice the \_\_c in the object name Book\_\_c. This indicates that it is a *custom object* that you created. Standard objects that are provided in the Salesforce application, such as Account, don't end with this postfix.
3. The next section of code contains the rest of the method definition:

|  |  |  |
| --- | --- | --- |
| 1 | for (Book\_\_c b :books){ | |
| 2 | b.Price\_\_c \*= 0.9; |

|  |  |
| --- | --- |
| 3 | } |

1. Notice the \_\_c after the field name Price\_\_c. This indicates it is a *custom field* that you created. Standard fields that are provided by default in Salesforce are accessed using the same type of dot notation but without the \_\_c, for example, Name doesn't end with \_\_c in Book\_\_c.Name. The statement b.Price\_\_c \*= 0.9; takes the old value of b.Price\_\_c, multiplies it by 0.9, which means its value will be discounted by 10%, and then stores the new value into the b.Price\_\_c field. The \*= operator is a shortcut. Another way to write this statement is b.Price\_\_c = b.Price\_\_c \* 0.9;. See [Expression Operators](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/langCon_apex_expressions_operators_understanding.htm).
2. Click **Save** to save the new class. You should now have this full class definition.

|  |  |
| --- | --- |
| 1 | public class MyHelloWorld { |
| 2 | public static void applyDiscount(Book\_\_c[] books) { | |

|  |  |  |
| --- | --- | --- |
| 3 | for (Book\_\_c b :books){ | |
| 4 | b.Price\_\_c \*= 0.9; |

|  |  |  |
| --- | --- | --- |
| 5 | } | |
| 6 | } |

|  |  |
| --- | --- |
| 7 | } |

You now have a class that contains some code that iterates over a list of books and updates the Price field for each book. This code is part of the applyDiscount static method called by the trigger that you will create in the next step.

**Add an Apex Trigger**

In this step, you create a trigger for the Book\_\_c custom object that calls the applyDiscount method of the MyHelloWorldclass that you created in the previous step.

Prerequisites:

* A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
* [The MyHelloWorld Apex class.](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_qs_class.htm)

A *trigger* is a piece of code that executes before or after records of a particular type are inserted, updated, or deleted from the Lightning platform database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire. All triggers run in bulk; that is, they process several records at once.

1. From the object management settings for books, go to Triggers, and then click **New**.
2. In the trigger editor, delete the default template code and enter this trigger definition:

|  |  |  |
| --- | --- | --- |
| 1 | trigger HelloWorldTrigger on Book\_\_c (before insert) { | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | Book\_\_c[] books = Trigger.new; | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | MyHelloWorld.applyDiscount(books); | |
| 6 | } |

1. The first line of code defines the trigger:

|  |  |
| --- | --- |
| 1 | trigger HelloWorldTrigger on Book\_\_c (before insert) { |

1. It gives the trigger a name, specifies the object on which it operates, and defines the events that cause it to fire. For example, this trigger is called HelloWorldTrigger, it operates on the Book\_\_c object, and runs before new books are inserted into the database.
2. The next line in the trigger creates a list of book records named books and assigns it the contents of a trigger context variable called Trigger.new. Trigger context variables such as Trigger.new are implicitly defined in all triggers and provide access to the records that caused the trigger to fire. In this case, Trigger.new contains all the new books that are about to be inserted.

|  |  |
| --- | --- |
| 1 | Book\_\_c[] books = Trigger.new; |

1. The next line in the code calls the method applyDiscount in the MyHelloWorld class. It passes in the array of new books.

|  |  |
| --- | --- |
| 1 | MyHelloWorld.applyDiscount(books); |

You now have all the code that is needed to update the price of all books that get inserted. However, there is still one piece of the puzzle missing. Unit tests are an important part of writing code and are required. In the next step, you will see why this is so and you will be able to add a test class.

**Add a Test Class**

In this step, you add a test class with one test method. You also run the test and verify code coverage. The test method exercises and validates the code in the trigger and class. Also, it enables you to reach 100% code coverage for the trigger and class.

Prerequisites:

* A Salesforce account in a sandbox Professional, Enterprise, Performance, or Unlimited Edition org, or an account in a Developer org.
* [The HelloWorldTrigger Apex trigger.](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_qs_trigger.htm)

Note

Testing is an important part of the development process. Before you can deploy Apex or package it for the Salesforce AppExchange, the following must be true.

* At least 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.

Note the following.

* + When deploying Apex to a production organization, each unit test in your organization namespace is executed by default.
  + Calls to System.debug are not counted as part of Apex code coverage.
  + Test methods and test classes are not counted as part of Apex code coverage.
  + While only 75% of your Apex code must be covered by tests, your focus shouldn't be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single records. This should lead to 75% or more of your code being covered by unit tests.
* Every trigger must have some test coverage.
* All classes and triggers must compile successfully.

1. From Setup, enter Apex Classes in the Quick Find box, then select **Apex Classes** and click **New**.
2. In the class editor, add this test class definition, and then click **Save**.

|  |  |
| --- | --- |
| 01 | @isTest |
| 02 | private class HelloWorldTestClass { | |

|  |  |
| --- | --- |
| 03 | static testMethod void validateHelloWorld() { |
| 04 | Book\_\_c b = new Book\_\_c(Name='Behind the Cloud', Price\_\_c=100); | |

|  |  |  |
| --- | --- | --- |
| 05 | System.debug('Price before inserting new book: ' + b.Price\_\_c); | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | // Insert book | |
| 08 | insert b; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | // Retrieve the new book | |

|  |  |  |
| --- | --- | --- |
| 11 | b = [SELECT Price\_\_c FROM Book\_\_c WHERE Id =:b.Id]; | |
| 12 | System.debug('Price after trigger fired: ' + b.Price\_\_c); |

|  |  |
| --- | --- |
| 13 |  |
| 14 | // Test that the trigger correctly updated the price | |

|  |  |  |
| --- | --- | --- |
| 15 | System.assertEquals(90, b.Price\_\_c); | |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |

1. This class is defined using the @isTest annotation. Classes defined this way should only contain test methods and any methods required to support those test methods. One advantage to creating a separate class for testing is that classes defined with isTest don’t count against your org’s limit of 3 MB of Apex code. You can also add the @isTest annotation to individual methods. For more information, see [IsTest Annotation](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_isTest.htm) and [Execution Governors and Limits](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm).
2. The method validateHelloWorld is defined as a testMethod. This annotation means that if changes are made to the database, they are rolled back when execution completes. You don’t have to delete any test data created in the test method.
3. Note
4. The testMethod keyword is now deprecated. Use the @isTest annotation on classes and methods instead.
5. First, the test method creates a book and inserts it into the database temporarily. The System.debug statement writes the value of the price in the debug log.

|  |  |  |
| --- | --- | --- |
| 1 | Book\_\_c b = new Book\_\_c(Name='Behind the Cloud', Price\_\_c=100); | |
| 2 | System.debug('Price before inserting new book: ' + b.Price\_\_c); |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Insert book | |

|  |  |
| --- | --- |
| 5 | insert b; |

1. After the book is inserted, the code retrieves the newly inserted book, using the ID that was initially assigned to the book when it was inserted. The System.debug statement then logs the new price that the trigger modified.

|  |  |
| --- | --- |
| 1 | // Retrieve the new book |
| 2 | b = [SELECT Price\_\_c FROM Book\_\_c WHERE Id =:b.Id]; | |

|  |  |
| --- | --- |
| 3 | System.debug('Price after trigger fired: ' + b.Price\_\_c); |

1. When the MyHelloWorld class runs, it updates the Price\_\_c field and reduces its value by 10%. The following test verifies that the method applyDiscount ran and produced the expected result.

|  |  |  |
| --- | --- | --- |
| 1 | // Test that the trigger correctly updated the price | |
| 2 | System.assertEquals(90, b.Price\_\_c); |

1. To run this test and view code coverage information, switch to the Developer Console.
2. In the Developer Console, click **Test** | **New Run**.
3. To select your test class, click **HelloWorldTestClass**.
4. To add all methods in the HelloWorldTestClass class to the test run, click **Add Selected**.
5. Click **Run**.

The test result displays in the Tests tab. Optionally, you can expand the test class in the Tests tab to view which methods were run. In this case, the class contains only one test method.

1. The Overall Code Coverage pane shows the code coverage of this test class. To view the percentage of lines of code in the trigger covered by this test, which is 100%, double-click the code coverage line for **HelloWorldTrigger**. Because the trigger calls a method from the MyHelloWorld class, this class also has coverage (100%). To view the class coverage, double-click **MyHelloWorld**.
2. To open the log file, in the Logs tab, double-click the most recent log line in the list of logs. The execution log displays, including logging information about the trigger event, the call to the applyDiscount method, and the price before and after the trigger.

By now, you have completed all the steps necessary for writing some Apex code with a test that runs in your development environment. In the real world, after you’ve tested your code and are satisfied with it, you want to deploy the code and any prerequisite components to a production org. The next step shows you how to do this deployment for the code and custom object you’ve created.

**Deploying Components to Production**

In this step, you deploy the Apex code and the custom object you created previously to your production organization using change sets.

Prerequisites:

* A Salesforce account in a sandbox **Performance**, **Unlimited**, or **Enterprise** Edition organization.
* [The HelloWorldTestClass Apex test class.](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_qs_test.htm)
* A deployment connection between the sandbox and production organizations that allows inbound change sets to be received by the production organization. See “Change Sets” in the Salesforce online help.
* “Create and Upload Change Sets” user permission to create, edit, or upload outbound change sets.

This procedure doesn't apply to Developer organizations since change sets are available only in **Performance**, **Unlimited**, **Enterprise**, or Database.com Edition organizations. If you have a Developer Edition account, you can use other deployment methods. For more information, see [Deploying Apex](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_deploying.htm).

1. From Setup, enter Outbound Changesets in the Quick Find box, then select **Outbound Changesets**.
2. If a splash page appears, click **Continue**.
3. In the Change Sets list, click **New**.
4. Enter a name for your change set, for example, HelloWorldChangeSet, and optionally a description. Click **Save**.
5. In the Change Set Components section, click **Add**.
6. Select Apex Class from the component type drop-down list, then select the MyHelloWorld and the HelloWorldTestClass classes from the list and click **Add to Change Set**.
7. Click **View/Add Dependencies** to add the dependent components.
8. Select the top checkbox to select all components. Click **Add To Change Set**.
9. In the Change Set Detail section of the change set page, click **Upload**.
10. Select the target organization, in this case production, and click **Upload**.
11. After the change set upload completes, deploy it in your production organization.
    1. Log into your production organization.
    2. From Setup, enter Inbound Change Sets in the Quick Find box, then select **Inbound Change Sets**.
    3. If a splash page appears, click **Continue**.
    4. In the change sets awaiting deployment list, click your change set's name.
    5. Click **Deploy**.