Use Case

When Renee LaFleur was in culinary school, she interned with one of the world's most esteemed chefs, who's renowned for his gourmet food truck with outdoor pop-up seating. Upon graduation, this experience inspired Renee to start her own food truck business: New Millennium Food Truck—a small food operation serving up big flavors.

After a few years of hard work and long hours, Renee grew her business from one to five food trucks. This success has allowed her to raise capital to pursue a passion project: a business in the "pay-it-forward" model. Remembering how hard it was to get started, she wants to use her new business to provide new chefs an easier way to establish their customer base.

Renee created New Millennium Delivery to allow new and emerging chefs to make a name for themselves by giving them a low-cost platform to peddle their food to the public. In return, their customers can order meals over the phone. A New Millennium Delivery representative then delivers the order and takes the happy customer's photo, which is then provided to the chef for social media posts.

Salesforce's platform, including Apex and Visualforce, is the technology that drives New Millennium Delivery's business. Previously, Renee had hired a developer to help her build her Salesforce application. Now, you're the lucky developer who's picking up where the previous developer left off.

Renee requested that the previous developer create the initial version of the application in just a few short weeks in order to so it could debut at a local food and wine festival. Renee knew she was forfeiting "good" for "fast" when she put her developer on such a tight timeline, and corners were cut and best practices sometimes ignored. Ignoring those best practices affected the rest of the development process—Renee wants you to fix this.

She's asked you to review and understand the legacy code in the current application, and then implement code fixes and new functionality using best practices for programming and application design.

Standard Objects

New Millennium Delivery stores its data in standard Salesforce objects, including:

- **Accounts**— New Millennium Delivery's customers who order food
- **Products**—The different meal items available for purchase
- **Pricebook Entries**— The prices of meal items
- **Orders**— Orders for meals to be delivered
- **Order Items**— Products included in a customer's order

If you are not familiar with the data model for standard objects related to Product, review it here.

Note that the New Millennium Delivery's business logic doesn't necessitate any new custom fields, relationships, or custom objects.

Business Requirements

Preliminary Test Data

After you complete Challenge 1, review the following New Millenium Delivery Products that will be created for you in your Trailhead playground.

Product Name	Product Family	Initial Inventory	Quantity Ordered	Active	Standard Price
Pizza	Entree	25	0	true	$20.00

Garlic Bread	Side	20	0	true	$6.00
Chocolate Cake	Dessert	15	0	true	$5.00
Coconut Water	Beverage	10	0	true	$3.00
Hamburger	Entree	25	0	true	$20.00
French Fries	Side	20	0	true	$6.00
Carrot Cake	Dessert	15	0	true	$5.00
Lemonade	Beverage	10	0	true	$3.00
Hot Dog	Entree	25	0	true	$10.00
Onion Rings	Side	20	0	true	$6.00
Jello	Dessert	15	0	true	$2.50
Iced Tea	Beverage	10	0	true	$3.00

Use Constants to Store Data

At Dreamforce, you learned that an application can be made more efficient with the best practice of using constants to store data whose value can change over time, but whose purpose remains constant. You want to implement this best practice by creating constants. A few of these constants will take advantage of custom labels.

As a reminder you should have already modified the Product Family picklist to only have the following active values:

Entree, Side, Dessert, Beverage

Create two custom labels with the following attributes:

Short Description	Name	Categories	Value	Protected Component
Select One	Select_One	constants	Select one	Unchecked
Inventory Level Low	Inventory_Level_Low	constants	Has a low inventory	Unchecked

Note: if you make a mistake when first creating a label, we recommend deleting it and recreating it from scratch.

Create an Apex class named Constants, defining the following constants.

Name	Value	Purpose
DEFAULT_ROWS	5	An Integer used to control the number of rows displayed by a Visualforce page.
SELECT_ONE	Value of the corresponding custom label	A String used to populate picklist values in Visualforce Pages.
INVENTORY_LEVEL_LOW	Value of the corresponding custom label	A String used to determine the threshold that causes low inventory alerts.
PRODUCT_FAMILY	List of Schema.PicklistEntry for the Family field on the Product2 object. This list is dynamic and must always reflect the currently configured values.	A list used to populate picklist values in Visualforce pages.
DRAFT_ORDER_STATUS	Draft	A String used to indicate that an order is a “draft”—an order that is in flight. You can’t activate a draft order unless you have a line item, and you can’t have a line item unless you have an order saved in the system.
ACTIVATED_ORDER_STATUS	Activated	A String used to evaluate if an Order is Activated or

		not.
INVENTORY_ANNOUNCEMENTS	Inventory Announcements	A String used to query a Chatter Group by Name.
ERROR_MESSAGE	An error has occurred, please take a screenshot with the URL and send it to IT.	A String used to display user friendly error messages on Visualforce pages.
STANDARD_PRICEBOOK_ID	Hardcoded value of the Standard Pricebook Id	An Id used to create Orders and PricebookEntries in business and test code. ** *FYI* ** Normally, we would recommend doing a query and using `Test.getStandardPricebookId()` to get the standard pricebook Id, however, since this code is being used for both business and test logic, that's not possible. In fact, just calling `Test.getStandardPricebookId()` from outside of a test method will throw a system exception. That would make it very hard on us to ensure that you completed this step properly.

Use Custom Metadata Types

Inventory managers told Renee they want to know when inventory is starting to run low. For instance, if there are only 15 of a type of dessert left in inventory, they want to be alerted. That is, once the inventory level of a particular product has dwindled down to the threshold of the product's associated Product Family, they want a notification.

Based on an analysis of the last few months of order data, Renee has determined the inventory threshold values for each of the Product Families. She wants to store the inventory threshold value for each Product Family, but she wants to easily modify these thresholds as sales continue to grow. After careful consideration, you determine that a custom metadata type fits the bill for meeting these requirements.

Create a custom metadata type with the following attributes.

Field	**Value**
Singular Label	Inventory Setting
Plural Label	Inventory Settings
Object Name	Inventory_Setting

Next, create a new field on the new metadata type.

Field	**Value**
Type	Number
Field Label	Low Quantity Alert
Length	18
Decimal Places	0
Field Name	Low_Quantity_Alert

Manage Inventory Settings to create the following custom metadata records.

Master Label	**Low Quantity Alert**
Entree	20
Side	10
Dessert	15
Beverage	5

Correctly Calculate Inventory Quantities

From speaking to Sam, a sales rep, Renee learned that there are issues entering new customer orders. Sam mentioned the following issues:

1) The value of the Quantity Ordered field isn't accurate

2) Saving a new order is often impossible because of system errors

3) Draft orders—orders that are in-flight but not yet activated—decrement available inventory prematurely.

After investigating the issues that Sam raised, you've made notes on resolving these issues:

- The business logic to derive the value of the Quantity Ordered field is not only faulty, it's also inefficient. The system should make the calculation efficiently using lean code, and correctly aggregate the Quantity Ordered.
- Sam correctly identified that draft orders prematurely decrement inventory levels. You can see that the previous developer tried to fix this, but then never finished. Now, you need to fix the business logic such that only successfully activated orders impact inventory levels.
- The logic to determine the value of the Quantity Ordered field should be updated to take into account all activated orders that are in the system, not just the orders visible to a single sales representative.
- The OrderTrigger should be updated to follow Apex Trigger best practices. This ensures that the business logic is modular and reusable. Additionally, the trigger definition should list only the necessary trigger events.

** *FYI* ** Normally this logic would apply on Update, Delete, and Undelete, and handle status changes back to Draft but in the interest of time and not making you perform repetitive tasks, we only focus on Update.

Correct the Visualforce Page That Allows the Addition of Products and Pricebook Entries

Renee's previous programmer created the Product2New Visualforce page. The intention of this page is to allow an inventory manager to rapidly enter, at once, multiple new Products and related PricebookEntry records. However, inventory managers have provided feedback that the page slows down data entry and doesn't give them all the information they need or want.

To resolve this, you need to implement an override of the standard **Add** and **New** buttons on the Product object with an updated version of Product2New.

The updated page should allow an inventory manager to create multiple products at once and enter an associated Unit Price for each product. Then, when the inventory manager clicks the **Save** button, the result in the system is that for each product entered, a related PriceBookEntry for the Standard Pricebook is created. Only entries on the page that have all fields populated should be saved; other entries shouldn't be saved. If an error occurs during save, a savepoint should be rolled back and a friendly error message should be displayed on the screen. Note that inventory managers should be able to enter as many products as they need to, and should be able to add multiple rows to the table with each click of the page's Add button.

Renee doesn't want to modify the Product2New page or the Product2Extension class when business requirements change in the future. To meet this need:

- Update the Visualforce page so that each column header displays the current field label.
- Update the Apex class such that the AddRows method uses the DEFAULT_ROWS constant.
- Create a new method named GetFamilyOptions for use by the **Family** picklist on the page.
  The GetFamilyOptions method should use both the SELECT_ONE and PRODUCT_FAMILY constants to generate the picklist options.

Because you want the Product and PriceBookEntry records to be associated with each other, implement and use an inner class named ProductWrapper, with the following attributes:

- **productRecord** of type **Product2**
- **pricebookEntryRecord** of type **PriceBookEntry**

Ensure current references to the list of Products are replaced with references to a list of ProductWrapper.

Inventory managers try to balance supply and demand for New Millennium Delivery's products. They requested that a horizontal bar chart be added to the Visualforce page. They want the chart to show them the Quantity Remaining of each Product Family. When a manager clicks **Add**, the chart should re-render performantly.

Because this chart data is useful for other purposes, it is generated from its own Apex Class named ChartHelper. Update ChartHelper to ensure the code inside always runs as the system. Complete the GetInventory method in ChartHelper to correctly calculate the aggregate of active products that have a positive Quantity Remaining, ensuring that the result is correct for all users. Renee wants this method to also be available for use by Lightning Components.

Generate a Test Data Factory

After some investigation, you determine that the previous programmer had decided to modularly and efficiently generate test data for unit tests, using the TestDataFactory class. However, you also notice that the class isn't finished—it has method names but no actual logic. Complete the TestDataFactoryclass so that it provides an efficient way to model sample business data that can be applied to the application's unit tests. Ensure that each method can be used as a utility method from your test classes and also that they never depend on the value of any instance member variables.

** *FYI* ** Normally, we'd use the @isTest annotation on the TestDataFactory class, but then you couldn't call the methods from your business logic and we couldn't call the methods to ensure they work properly.

Create Unit Tests for orderTrigger and product2Extension

The more recent Apex and Visualforce code implemented by the previous developer doesn't have the minimal required test coverage, and therefore can't be deployed. You're a seasoned developer, and you know that all code must be tested thoroughly. Renee has asked you to ensure that each method has the minimum 75% code coverage required to be deployed, and that new and existing unit tests do not use actual customer data. After reviewing the legacy unit tests, you notice you can use the existing Product2Tests Apex class to test Product2Extension, and you can use the existing OrderTests Apex class to test OrderTrigger and its helper.

Validate the Logic of orderTrigger

First, create a method that can be used by test methods to verify that the **Quantity Ordered** field is correctly updated on Products when Orders are activated. Go back to the TestDataFactory Apex class and create a new method with the following signature:

```
VerifyQuantityOrdered(Product2 originalProduct, Product2 updatedProduct, Integer qtyOrdered)
```

Copy

This method should perform an assertion that verifies that updatedProduct's **Quantity_Ordered__c**field is equal to the sum of the originalProduct's **Quantity_Ordered__c** field and the value of qtyOrdered.

Next, go back to the orderTests class and ensure its test methods don't (and can't) use live data. Create a new method named SetupTestData that is will be used to generate test data for all the unit tests in OrderTests. This method should just call the InsertTestData method in the TestDataFactory class

Finally, create a new test method named OrderUpdate_UnitTest in OrderTests. This method must activate the Orders created in your SetupTestData method using the ACTIVATED_ORDER_STATUS constant and then use the VerifyQuantityOrdered method to verify that the Quantity Ordered field on Products was increased by the trigger.

** *FYI* ** A method like **VerifyQuantityOrdered** can be useful to ensure logic is tested consistently across all your unit tests. It also makes it a lot easier for us to assess that you completed this challenge properly! As your codebase grows, you may create more of these types of methods. If so, you would want to move them into a different class than the TestDataFactory.

Validate the Logic of the product2New Page

Complete the Product2Extension_UnitTest in Product2Tests. This unit test should simulate a user's visit to, and interaction with, the Product2New page and test that it behaves as expected. When a user first visits the page, there should be multiple rows displayed on the screen. Assert that the size of the **productsToInsert** list is equal to the DEFAULT_ROWS constant. When the Add button is clicked, an additional set of rows should be added, so assert that the size of the **productsToInsert ** list is double **DEFAULT_ROWS** after the button is clicked once. Don't forget to ensure that the test methods don't (and can't) use live data.

Next, test the Save button. Verify that populated rows are saved and unpopulated rows are not saved. Loop through the rows in the **productsToInsert** list and populate the values of the first 5 records, and then simulate clicking the Save button. Verify that the button worked by asserting that only 5 products were saved.

Be sure to run your tests when you're done to ensure they pass without failures.

Automate Internal Announcements When Inventory Is Low

The staff at New Millennium Delivery needs to be aware of any inventory that is running low. The previous developer had unsuccessfully attempted to do this. Renee wants you to fix the issues in the previous developer's logic and ensure that all appropriate employees can choose to be notified, and not just those following a given product.

You've determined that a Chatter group is an easy way for employees to opt-into receiving these notifications. Start by creating a Chatter group named **Inventory Announcements** and give the group this description: This group is for New Millennium Delivery employees to receive inventory announcements. Be sure to create the group so that it wont be automatically archived and that it will be accessible by all users.

After reading about the **ConnectAPI**, you realize that a Chatter Announcement is a better fit than a FeedItem post because an announcement acts more like an alert, in that it is timely and it expires. You know that a low inventory value should prompt an announcement to be posted to the Inventory Announcements Chatter group, so an Apex trigger must be used. After reviewing the legacy code, you realize the previous programmer attempted to write this business logic in the Product2Trigger, but that logic is faulty.

Posts count as DML operations, so you will need to implement the Queueable interface to ensure that a bulk operation will result in all announcements being posted, including in the event that a large volume of announcements need to be posted. Your predecessor found some code on the Salesforce Developer Forum and created the legacy AnnouncementQueueable Apex class. Use this class to get started.
The AnnouncementQueueable and Product2Helper classes should use the **ConnectAPI** namespace to automatically post a Chatter Announcement when a product level falls below a given threshold.

Modify AnnouncementQueueable to implement the Queueable interface and call its postAnnouncements method. Ensure that it requeues itself when it has more Announcements to post than limits allow.

After reviewing the existing code in the product2Helper class, you realize that it too can benefit from your constants class. Modify Product2Helper to use the INVENTORY_ANNOUNCEMENTS constant instead of "group name" to ensure consistency in the app.

Complete the PostAlerts method in Product2Helper to construct new AnnouncementInputs for the Chatter Group and for use with the AnnouncementQueuable Apex class.

Next, complete the AfterUpdate method so that it uses the PostAlerts method when you determine that a Product's Quantity_Remaining__c field has dropped below the threshold value captured in the custom metadata records you created previously.

Best practices dictate that business logic code should be stored in Apex classes to make it reusable, easier to test, and easier to debug. Modify Product2Trigger to execute only on the After Update event and call the logic in Product2Helper.

** *FYI* ** Many developers (including us) prefer to use Test Driven Development and would start with Tests first. While it works well in the real world, it's not as easy to assess programatically, so stick to doing these challenges in the order provided.

Complete a Controller Extension for a Visualforce Page to Enable Rapid Order Entry

New Millennium Delivery's customers love some foods so much that products frequently sell out quickly. It's your job to ensure there's an easy way for inventory managers to determine the food most ordered by each customer so that they can offer the benefit of personalized customer menus.

The employees who take orders at New Millennium Delivery need a page that provides a streamlined process for saving orders and order items. The previous developer created a Visualforce page, OrderEdit, to facilitate this, but was not able to complete its controller extension.

Order Edit

New Order

Order Details

Save Cancel

Account Name: Jane Doe

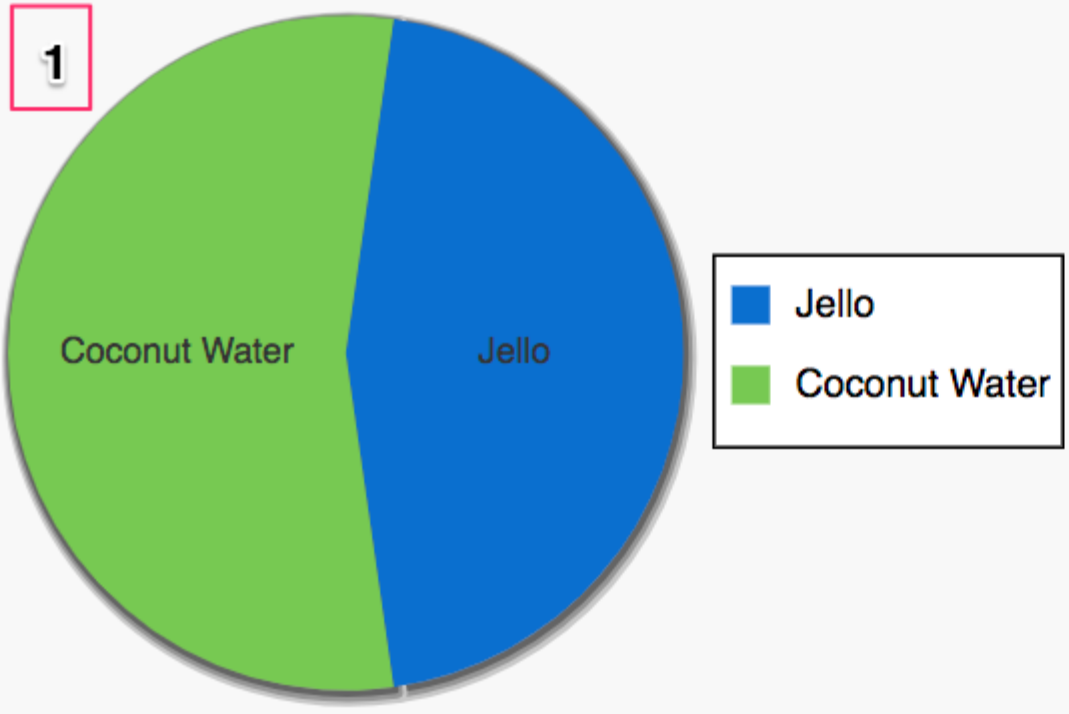
Order Start Date: 11/16/2017 [11/16/2017]

Save Cancel

Order Pricing Summary

Order

1



Product

Figure 1.1: The OrderEdit Visualforce Page

You need to ensure that this page is used when inventory managers create or edit an Order. Complete the OrderExtension class to allow the page to show order details, a pie chart summarizing the order items and Order Amount, and a list of active products.

1. The pie chart should display a wedge for each item in the order, with the value of the wedge equal to the quantity multiplied by the unit price. Complete the OnFieldChange method to keep track of changes to values in the Quantity or Unit Price fields. This method should also ensure that the data in the pie chart and Order Total reflects these changes and are updated on the screen.
2. The Products available for use on the page will be the entire set of all Active Products that Millennium Delivery sells. Use the **DEFAULT_ROWS** constant to limit the number of rows displayed at one time in the product list. To help inventory managers find products faster, complete the SelectFamily method so that it limits the Products displayed to only those with the selected Product Family. Ensure that quantity and unit price values entered are preserved when a user filters Products by Product Family.
3. Use the **StandardSetController** methods to complete the pagination methods provided, enabling the user to move through multiple pages of available products. Ensure that quantity and unit price values entered are preserved when a user paginates.

Just like she requested with the product2New page, Renee doesn't want to modify the OrderEdit page or OrderExtension class when business requirements change in the future. Copy the GetFamilyOptions method used in the Product2Extension class to complete the GetFamilyOptionsmethod.

If, at any time, the user clicks the **Cancel** button, none of their changes should be saved.

When the user clicks Save, the Order and Order Items should be saved. Only Order Items with a Quantity greater than zero should be saved; others should not. When editing an existing Order, any Order Items that have been modified to have a Quantity of zero should be deleted. If an error occurs during save, a savepoint should be rolled back and a friendly error message should be displayed on the screen.

Create Unit Tests

You may recall from the Apex Testing module that testing is the key to successful long-term development and is a critical component of the development process. In addition to being critical for quality assurance, Apex unit tests are also requirements for deploying and distributing Apex. Renee has asked you to create new unit tests.

- Create a new method titled OrderExtension_UnitTest in OrderTests that tests all the methods in the OrderExtension class. Use the code in Product2Tests as a template.
- Create a new method titled Product2Trigger_UnitTest in Product2Tests that tests the logic when a Product's Quantity Ordered value is updated.
- In the Developer Console, clear all test data and then run all tests.
- Ensure you have 75% or higher test coverage on Product2Trigger and OrderTrigger triggers.
- Ensure you have 75% or higher test coverage on Product2Extension and OrderExtension classes.
- Ensure you have 90% or higher coverage on Constants, ChartHelper, Product2Helper, OrderHelper, and TestDataFactory classes.

** *FYI* ** Test Coverage is a requirement for deployment and a great way to ensure that your code is not failing. As discussed in prior badges, the purpose of your Unit Tests is not to acheive a certain percent, but rather to ensure that your business logic behaves as expected.