Force.com Workbook

Last updated: March 8, 2013

Table of Contents

About the Force.com Workbook

The Force.com Workbook shows you how to create an application (or "app") in a series of tutorials. While you can use the Force.com platform to build virtually any kind of app, most apps share certain characteristics such as:

- A database to model the information in the app
- Business logic and workflow to carry out particular tasks under certain conditions
- A user interface to expose data and functionality to those logged in to your app
- A public website to show data and functionality on the Web

The tutorials are centered around building a very simple warehouse management system. You'll start developing the app from the bottom up; that is, you'll first build a database model for keeping track of merchandise. You'll continue by adding business logic: validation rules to ensure that there is enough stock, workflow to update inventory when something is sold, approvals to send email notifications for large invoice values, and trigger logic to update the prices in open invoices. Once the database and business logic are complete, you'll create a user interface to display a product inventory to staff, a public website to display a product catalog, and then the start of a simple store front.

Each of the tutorials builds on the previous tutorial to advance the app's development and simultaneously showcase a particular feature of the platform. It may sound like a lot, but it's all quite easy—as you'll soon see.

Intended Audience

This workbook is intended for developers new to the Force.com platform, and Salesforce admins who want to delve more deeply into app development using coding. If you're an admin just getting started with Force.com, see the Force.com Platform Fundamentals for point-and-click app development.

Supported Browsers

Browser	Comments
Microsoft® Internet Explorer® versions 7, 8, 9, and 10	If you use Internet Explorer, we recommend using the latest version. Apply all Microsoft software updates. Note these restrictions. • The Compatibility View feature in Internet Explorer is not supported. • The Metro version of Internet Explorer 10 is not supported. • Internet Explorer 10 is not supported for the Service Cloud console. For configuration recommendations, see "Configuring Internet Explorer" in the online help.
Mozilla® Firefox®, most recent stable version	Salesforce.com makes every effort to test and support the most recent version of Firefox. For configuration recommendations, see "Configuring Firefox" in the online help.

Browser	Comments
Google Chrome™, most recent stable version	Google Chrome applies updates automatically; salesforce.com makes every effort to test and support the most recent version. There are no configuration recommendations for Chrome. Chrome is not supported for the Console tab or the **Add Google Doc to Salesforce** browser button.
Google Chrome Frame™ plug-in for Microsoft® Internet Explorer® 6 and 7	Supported plug-in for Internet Explorer 6 and 7 only. Google Chrome Frame applies updates automatically; Salesforce supports only the most recent version. For configuration recommendations, see "Installing Google Chrome Frame for Microsoft® Internet Explorer®" in the online help. Chrome Frame plug-in is not supported for the Service Cloud console or Forecasts.
Apple® Safari® version 5.1.x on Mac OS X	There are no configuration recommendations for Safari. Apple Safari on iOS is not supported. Safari is not supported for the Salesforce CRM Call Center CTI Toolkit or the Service Cloud console.

Before You Begin

This workbook is designed to be used with a Developer Edition organization, or *DE org* for short. DE orgs are multipurpose environments with all of the features and permissions that allow you to develop, package, test, and install apps.

1. In your browser go to `developer.force.com/join`.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.
4. Enter a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact, it's usually better if they aren't the same. Your username is your login and your identity on `developer.force.com`, and so you're often better served by choosing a username that describes the work you're doing, such as `develop@workbook.org`, or that describes you, such as `firstname@lastname.com`.
5. Read and then select the checkbox for the `Master Subscription Agreement`.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you'll receive an email with a login link. Click the link and change your password.

> **Note:** One tutorial in this workbook requires one Salesforce user license for creating a new user. If you've signed up for a new DE org, you should be all set. However, if you are using an existing organization where you've already created some users, make sure that you have one Salesforce user license available.
>
> To find out what user licences are available for your organization, click *Your Name* > **Setup** > **Company Profile** > **Company Information** and check the `User Licenses` section. If no Salesforce user license is available, you can create a new DE org. Alternatively, you can use the Salesforce Platform or the Force.com — Free user licenses when creating the new user and custom profile.

Tell Me More....

At the end of each step there is an optional Tell Me More section. If you like to do things quickly, move on to the next step. However, if you're a smell-the-roses type, there's a lot of useful information here and some additional steps to flesh out your app.

- To learn more about Force.com and to access a rich set of resources, visit Developer Force at developer.force.com.
- You can find the latest version of this Workbook and additional tutorials that extend the app at developer.force.com/workbook.

Tutorial #1: Creating a Warehouse App

Level: Beginner; **Duration:** 20-30 minutes
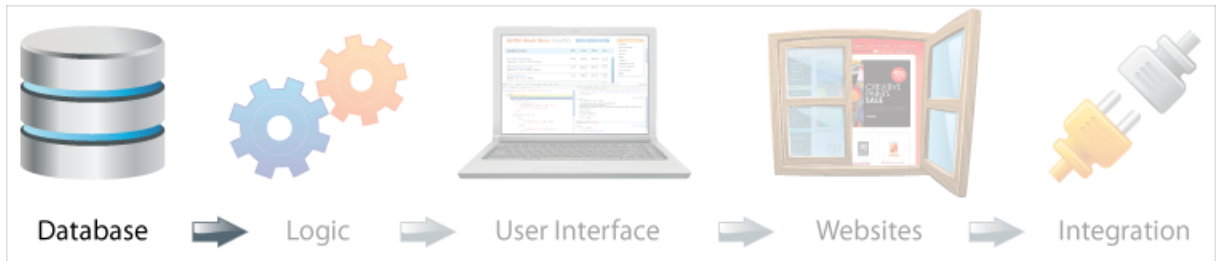
At the heart of this app is what you want to sell: merchandise. So you'll begin by creating a data object that keeps track of all the elements of a particular merchandise item, such as its name, description, price, and so on. On the Force.com platform, these data objects are called *custom objects*. If you are familiar with databases, you can think of them as a table.

An object comes with standard fields, and screens that allow you to list, view and edit information about the object. But you can also add your own fields to track or list just about anything you can think of. When you complete this tutorial, you will have a working app with its own menu, a tab, and a custom object that tracks the name, description, and price of all your merchandise, as well as screens that allow you to view and edit all of this information.

Step 1: Create a Warehouse Custom App

An app is a set of tabs. In this step you'll create a Warehouse app, including a Merchandise object and tab. Later, you'll add fields to the Merchandise object, as well as additional objects and tabs to the Warehouse app.

1. Launch your browser and go to `https://login.salesforce.com`.
2. Enter your username (in the form of an email address) and password.
3. Click `Your Name` > **Setup** in the upper right corner, and then click **Create** > **Apps** in the sidebar menu.
4. Click **Quick Start**.
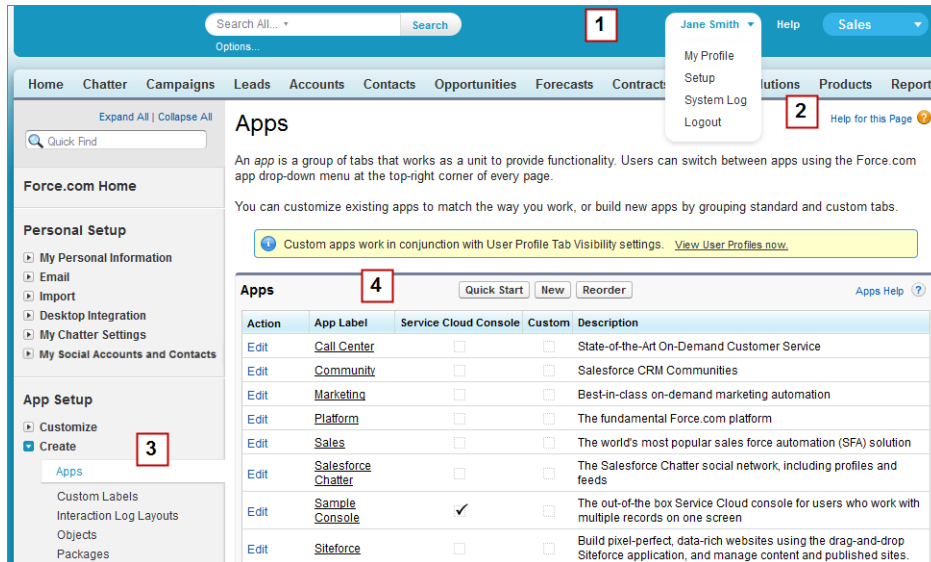
5. In the Force.com Quick Start overlay, enter the app and object details.

   • For the `App Label`, type `Warehouse`
   • For the `Plural Label`, type `Merchandise`.
   • For the `Singular Label`, type `Merchandise`.
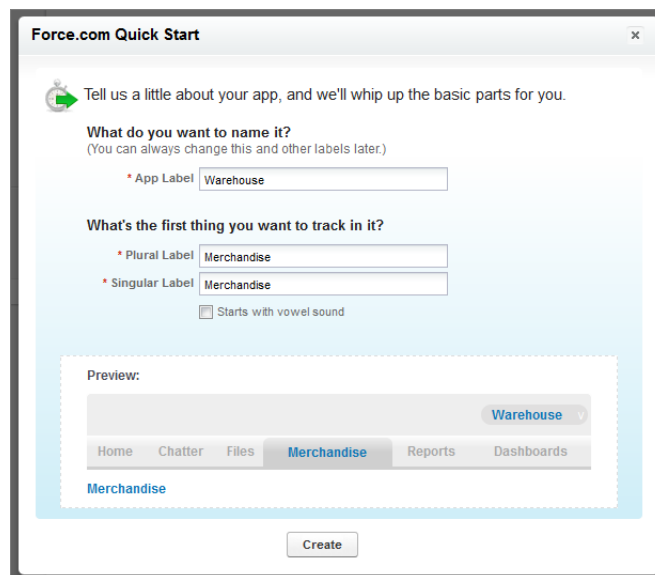
The preview pane shows what your app and tab will look like.

6. Click **Create** to finish creating your new object.

7. Click **Go To My App** to see your new app.

When you click **Go To My App**, the list view page for the Merchandise custom object opens. You can take a short tour, which shows the features of the app as well as how to continue building it. Take a look at the following image to familiarize yourself with the Warehouse custom app.
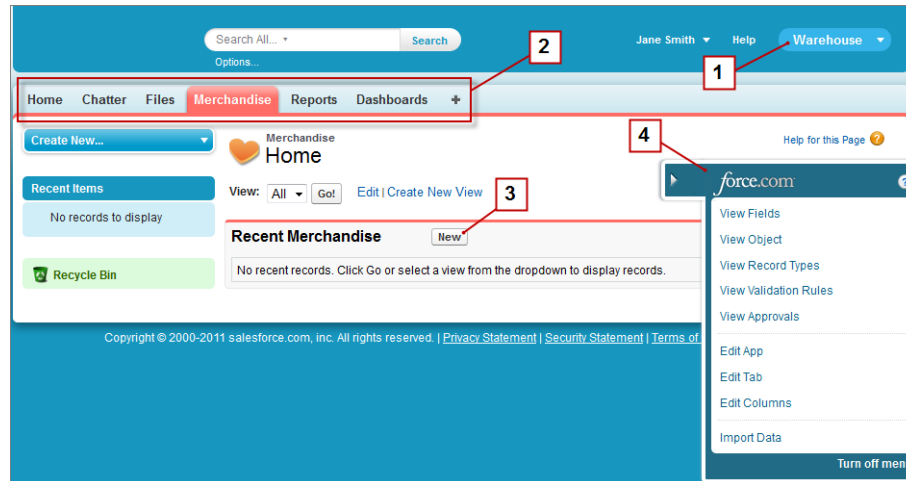
1. **Force.com app menu**—Shows the apps that are available to you. The app you just created is selected.
2. **Tabs**—Provide an easy way to find and organize objects and records. In the Merchandise tab, which is open, you can create, view, and edit records. The other tabs are the standard feature tabs that are included with every app.
3. **Create records**—You can click **New** to add records to your custom object. If you click this button now, you'll see only one data entry field in the object, but you'll create more in the next step.
4. **Force.com Quick Access menu**—Lets you quickly jump to relevant app customization features. It's available from any

   object list view page and record detail page, but only for users with the "Customize Application" permission. Click to show or hide this menu.

Tell Me More....

The tabs in an app don't have to be related to each other. In fact, you can modify custom apps to group all of your most frequently used tabs together in one place. For example, if you refer to the Accounts tab a lot, you can add that to the Warehouse app. You can switch between apps you have created, bought, or installed by selecting them from the menu.

Step 2: Add Description, Price, and Total Inventory Fields

A merchandise object should have fields that are used for tracking various information, such as a description, how much individual units cost, and how many units are in stock. You can add custom fields to list or track just about anything you can think of.

Follow these steps to create custom fields for the description, price, and inventory.

1. Create the Description field.

   a. From the Merchandise list view page, click to open the quick access menu (if it isn't already open).
   b. Hover over **View Fields** and click **New** to launch the New Custom Field wizard.
   c. For Data Type, select `Text Area` and click **Next**.

   d. Fill in the custom field details.

   - For the `Field Label`, type `Description` and press the Tab key.
   - The `Field Name` field should already be populated with the field name: `Description`.
   - Select the `Required` checkbox. One of your business rules says that you can't create new merchandise without providing a description. Making Description a required field fulfills that business rule.
   - Optionally, fill in the `Description` and `Help Text` fields. It's a good idea to fill these in, but for the sake of brevity we leave these out in the remainder of the workbook.
   - Don't fill in the `Default Value` field.

   e. Click **Next**, accept the defaults, and click **Next** again.
   f. Click **Save & New** to save the `Description` field and return to the first step of the wizard.

2. Create the Price field.

   a. For Data Type, select `Currency`, and click **Next**.
   b. Fill in the custom field details:

   - For the `Field Label`, enter `Price`.

- For the `Length` enter `16` and for Decimal Places enter `2`.
- Check the `Required` checkbox.

c. Leave the defaults for the remaining fields, and click **Next**.
d. In the next step, accept the defaults, and click **Next**.
e. In the next step, click **Save & New** to save the `Price` field and to return to the first step of the wizard.

> **Note:** This app has only one Price field, and we'll use that for both the stock price and retail price. However, you can make another Stock Price field now if you want to, just follow the same steps above and use the Stock Price name.

3. Create the Total Inventory field.

   a. For `Data Type`, select `Number` and click **Next**.
   b. Fill in the custom field details:

   - For the `Field Label`, enter `Total Inventory`.
   - Select the `Required` checkbox.

   c. Accept the defaults for the remaining fields, and click **Next**.
   d. In the next step, accept the defaults, and click **Next**.
   e. Click **Save** to create the `Inventory` field and to go to the Merchandise Custom Object page.

Take a look at the following image to familiarize yourself with the Merchandise custom object.

1. **Merchandise detail page**—This page shows you everything you need to know about your Merchandise custom object. Soon you'll add relationships, validation rules, and other neat features to this object.
2. **API name**—When you created the Merchandise object, you didn't specify an API name, but one was generated for you. This name is how the object is referenced programmatically. All custom objects end in __c, which differentiates them from standard objects.
3. **Standard fields**—Some fields are generated automatically; these are standard fields. For example, the Merchandise object has a standard field for Owner, which means it automatically tracks who created each record.

4. **Custom fields**—Includes the fields you just created in this step. Like custom objects, custom fields have API names that end in __c.

At this point we have a nice representation of our warehouse items: they each have a name, a description, and a price. We also recorded the total number of items (Total Inventory) that we have for each piece of merchandise.

Tell Me More....

Why do you need an API name as well as the object and field label? A label is what the user sees, so it should be easy to read and may contain spaces. The API name is used internally in code, and can't contain spaces or illegal characters. For example, a field labeled "Customer ph# :" would be named Customer_ph in the code (the system replaces spaces with underscores and removes the # and : characters).

Step 3: Create a Merchandise Record

At this point you've created a functioning app. When you define an object on Force.com, the platform automatically generates a user interface that lets you create, read, update, and delete records. Now check to see how your new app works.

1. Select the Warehouse app from the Force.com app menu.
2. Click the Merchandise tab and then click **New** to create a new product.
3. Fill in all the fields.

   - In the Merchandise Name field enter Wee Jet.
   - For Description field enter A small plane.
   - In the Price field enter 9.99.
   - In the Total Inventory field enter 2000.

4. Click **Save**.

Tell Me More....

Your warehouse will work just fine with only one product, but it would be more realistic if you had more merchandise. Go ahead and create some more. Tip: click **Clone** on this record to create another similar one, and then use **Save and New** to create more new records rapidly.

Summary

Congratulations! You have just built a Warehouse app that keeps track of products you have in stock. As you can see, it's easy to use the Force.com quick start overlay to create a basic app and the Force.com quick access menu and wizards to extend your app. Although the app is far from complete, it already has a lot of built-in functionality, such as the ability to view and create merchandise. The next step is to use an invoice to track how your merchandise moves in and out of your warehouse.

Tutorial #2: Adding Relationships

Level: Beginner; **Duration:** 20-30 minutes
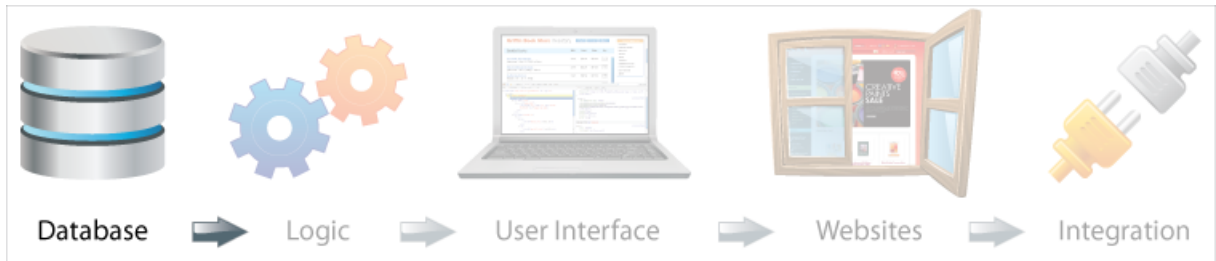
In this tutorial, you create two new objects, an invoice statement and a line item, and associate them to each other in a master-detail relationship. This relationship lets you compose multiple line items and associate them with a single invoice statement. Each line item is an indication of how many units of a particular merchandise item was sold, so you'll also add a relationship between the merchandise and line item objects.

If you're familiar with relational databases, you can think of relationships as foreign keys, the difference being that Force.com takes care of the underlying implementation for you, and all you need to do here is establish how the objects are related.

Prerequisites

Warehouse App

You first need to create the basic Warehouse app as described in Tutorial #1: Creating a Warehouse App on page 3.

Step 1: Create an Invoice Statement Custom Object

An invoice statement is required to move inventory into or out of the warehouse. In this step, you create an invoice statement with a unique number, a status, and a description.

1. Click **Your Name** > **Setup** > **Create** > **Objects**.
2. Click **New Custom Object**.

> **Tip:** If you're in the Merchandise tab, click ◀ to open the quick access menu. Then hover over **View Object** and click **New**.

3. Fill in the custom object definition.

   - In the `Label` field , type `Invoice Statement`.
   - In the `Plural Label` field, type `Invoice Statements`.
   - Select **Starts with vowel sound**.
   - In the `Record Name` field , type `Invoice Number`.
   - In the `Data Type` field , select `Auto Number`.
   - In the `Display Format` field, type `INV-{0000}`.
   - In the `Starting Number` field, type `1`.

4. In the Optional Features section, select **Allow Reports** (you'll create reports in a later tutorial).
5. Click **Save**.
6. Add a Status field.

    a. Scroll down to the Custom Fields & Relationships related list and click **New**.

    b. For Data Type, select `Picklist` and click **Next**.

    c. Fill in the custom field details.

      • In the `Field Label` field , type `Status`.

      • Type the following picklist values in the box provided, with each entry on its own line.

      ```
      Open
      Closed
      Negotiating
      Pending
      ```

      • Select the checkbox for `Use first value as default value`.

   d. Click **Next**.
   e. For field-level security, make sure **Read Only** is selected and then click **Next**.
   f. Click **Save & New**.

7. Now create an optional Description field.

   a. In the Data Type field, select `Text Area` and click **Next**.
   b. In the `Field Label` and `Field Name` fields, enter `Description`.
   c. Click **Next**, accept the defaults, and click **Next** again.
   d. Click **Save** to go the detail page for the Invoice Statement object.

Your Invoice Statement object should now have two custom fields, as shown here.

Tell Me More....

Note how we set the invoice name data type to Auto Number, and also supplied a Display Format. The platform will now automatically assign a number to each new unique record that is created, beginning with the starting number you specify. When displayed, the format will look as follows: INV-0002.

Step 2: Create a Line Item Object

Each invoice is going to be made up of a number of invoice line items, which represent the number of Merchandise items being sold at a particular price. You are first going to create the Line Item object, and then later relate it to the Invoice Statement and Merchandise objects.

1. Click `Your Name` > **Setup** > **Create** > **Objects**.
2. Click **New Custom Object** and fill in the custom object definition.

   - In the `Label` field , enter `Line Item`.
   - In the `Plural Label` field , enter `Line Items`.
   - Change the `Record Name` to `Line Item Number`.
   - Leave the `Data Type` field set to `Text`.

3. In the Optional Features section, select **Allow Reports** and click **Save**.
4. Add a read-only Unit Price field. The field is read-only because the value will be retrieved from the Merchandise object in a later tutorial. We'll call it Unit Price so that we don't get confused with the Merchandise object Price field.

   a. Scroll down to the Custom Fields & Relationships related list and click **New**.
   b. In the `Data Type` field, select `Currency` and click **Next**.
   c. Fill in the custom field details.

      - In the `Field Label` field, enter `Unit Price`.
      - In the `Length` field, enter `16`, and for `Decimal Places` enter `2`.

   d. Click **Next**.
   e. Select the top-level **Read-Only** checkbox to mark this selection for all profiles, click **Next**, and then click **Save & New**.

Note: If **Read-Only** is not available, you probably selected **Required** by mistake. Click **Previous** and deselect **Required**.

5. Follow similar steps to add a Units Sold field.

   a. In the `Data Type` field, select `Number` and click **Next**.

   b. In the `Field Label` field, enter `Units Sold` and click **Next**.

   c. Click **Next**, accepting the defaults.

   d. Click **Save** to return to the detail page of the Line Item custom object.

Tell Me More....

At this point you've created three custom objects: Merchandise, Inventory Statement, and Line Item. On each of those objects you created custom fields to represent text, numbers, and currency. All of those fields have something in common: the values are provided by the user. You also created two custom fields that have system-generated values: the Status picklist, which defaults to Open; and the Invoice_Number field, which is automatically assigned by the Auto-number data type. In the next step you will create two more fields. Unlike the previous fields, these get their values from other objects.

Step 3: Relate the Objects

Now that you have all the objects representing the data model, you want to relate them to each other. The Line Item is related to both an Invoice Statement (a statement is composed of a number of line items) and Merchandise (a line item takes its price from the merchandise).

1. On the detail page of the Line Item object, scroll down to the Custom Fields & Relationships related list and click **New**.

2. In the `Data Type` field, select **Master-Detail Relationship** and click **Next**.

3. In the `Related To` field, select your **Merchandise** custom object and click **Next**.

4. Accept the defaults on the next three screens by clicking **Next**.

5. Deselect the checkbox next to `Merchandise Layout` so that Line Items don't appear on the Merchandise related list.

6. Click **Save & New**.

7. In the `Data Type` field, select **Master-Detail Relationship** and click **Next**.

8. In the **Related To** field, select your **Invoice Statement** custom object and click **Next**.

9. Accept the defaults on the following screens by clicking **Next**, and then click **Save** to return to the Inventory Item detail screen.

Tell Me More....

You have just created two master-detail relationships ensuring that your Invoice Statement records are related to Invoice Line Item records, and that the Invoice Line Items are related to Merchandise. Master-detail relationships also support roll-up summary fields, allowing you to aggregate information about the child records. You'll use that feature in a later tutorial.

Step 4: Add Invoice Statements to the App

Tabs provide an easy way to find and organize objects and records. In this step you'll create a tab for the Invoice Statement object and add it to your Warehouse app. This will expose the user interface that Force.com automatically generates for this object.

1. Within the Setup area, click **Create** > **Tabs**.

   You may notice that a tab for the Merchandise custom object already exists. This was automatically created when you created the Warehouse app. If you don't like the default tab style that was assigned to the Merchandise tab, you can edit it as well.

2. In the Custom Object Tabs related list, click **New** to launch the New Custom Tab wizard.
3. From the `Object` drop-down list, select **Invoice Statement**.
4. For the `Tab Style`, click the lookup button and select the **Form** icon.

5. Accept the remaining defaults, and click **Next** and then **Next** again.
6. On the Add to Custom Apps page, deselect all the checkboxes except `Warehouse`. This will add the Invoice Statements tab to your Warehouse app.
7. Click **Save** to finish creating the tab.

As soon as you create the tab, you can see it at the top of the screen.

Tell Me More...

When you create a tab, you're not stuck with a standard set of icons and colors; you can choose your own color and custom image by clicking **Create your own style**.
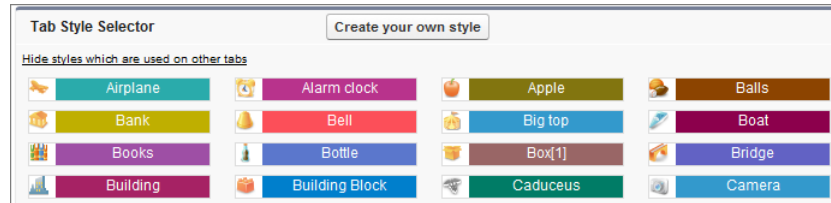
Step 5: Create an Invoice Record

As you saw in the previous tutorial, the platform automatically generates a user interface for the objects you create, so that you can view, edit, delete, and update records. Because you also related the objects, the user interface provides a way of navigating between related records as well.

1. Click the Invoice Statements tab.
2. Click **New**.
3. In the `Description` field, enter `First Invoice` and then click **Save**.

The detail page of your invoice statement should look like this.
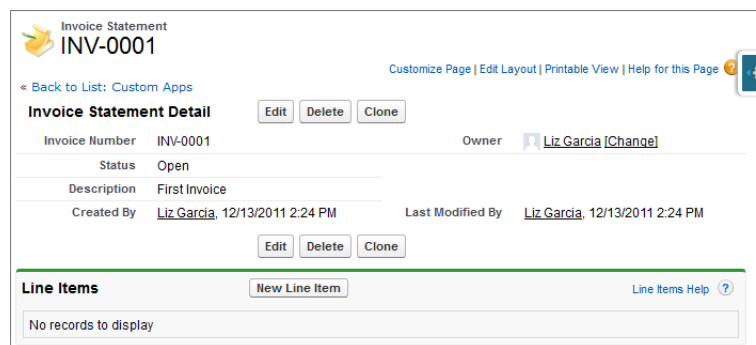
Notice how the invoice was automatically assigned a number, and how the user interface displays an empty Line Items related list below it. The Invoice Statement is linked to the Line Item via a master-detail field—that's how you created this relationship and why there's a related list on the Invoice Statement detail page. Next you'll add a line item to the invoice.

1. Click **New Line Item**.
2. Fill in the fields.

   - In the `Line Item Number` field enter `1`.
   - In the `Unit Price` field enter `10`.
   - In the `Units Sold` field, enter `4`.
   - For the `Merchandise` field, click the lookup button and select a product.

3. Click **Save**.

Tell Me More....

- In the Reports tutorial you'll find it's useful to have multiple invoices that contain different merchandise. You can create more invoices now or later.
- You might be wondering why Line Item Number is a text field, when what you enter is a number. If line items are numbered, why not make it an auto-number field, like Invoice Statements? The short answer is that it's easier to work with text when we work with records, and this tutorial is supposed to be easy. You can make Line Item an auto number field if you want to, but it makes a later tutorial, Tutorial #10: Creating a Public Web Page Using Sites, a bit more difficult.

Summary

In this tutorial, you created relationships between the objects in your data model. The relationships work like foreign keys in relational databases, yet they operate at a more abstract level, letting you concentrate on what matters (the relationships) instead of the underlying implementation. The master-detail relationship allows you to aggregate information, and in the next tutorial you'll see how to sum the price of each invoice line item into the invoice statement. The relationships also provide additional benefits: you can navigate to the related records in a user interface and a query language. You'll learn about that later as well.

Now that you have built the basic app, you can add business logic with formulas and validation rules in Tutorial #3: Using Formulas and Validation Rules

Tutorial #3: Using Formulas and Validation Rules

Level: Beginner; **Duration:** 20-30 minutes
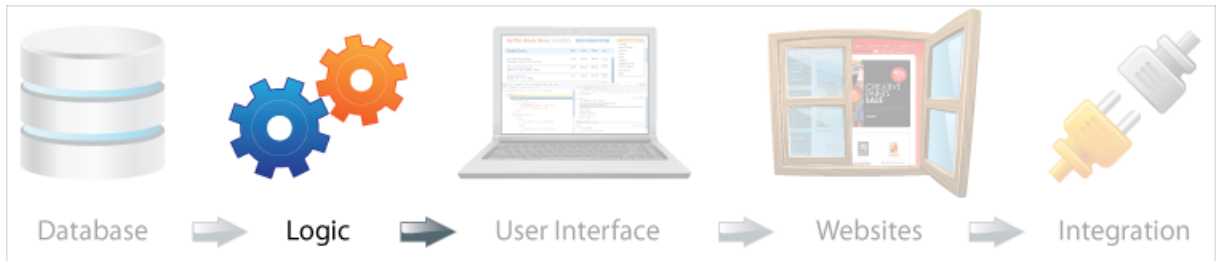
The Force.com platform lets you create formulas and field validation rules to help maintain and enhance the quality of the data entered in your app. Both formula fields and field validation rules use built-in functions that allow you to automatically manipulate your data, validate your data, and calculate other values based on your data. The functions you use in formula fields and field validation rules are a lot like the functions you would use in a spreadsheet program, where you can reference values in other cells of the spreadsheet, perform calculations, and return a result. However, with formula fields and validation rules, you reference fields in your app's records.

In this tutorial, you will enhance the Warehouse app by adding a formula field to automatically calculate the total value for each line item. You will also use this new line item value and a property of the master-detail relationships to automatically total an invoice value. Finally, you will add a formula to perform an inventory check, ensuring that you can't create a line item for more merchandise than you have in stock.

Prerequisites

Warehouse App

> You first need to create the basic Warehouse app and add relationships, as described in Tutorial #2: Adding Relationships on page 10.

Step 1: Calculate a Value for Each Line Item

In the first step of this tutorial, you will add a new calculated field called Value to the line item. This field will multiply the number of items with the price and act as a total for each line item.

1. Navigate to the Line Item custom object page by clicking *Your Name* > **Setup** > **Create** > **Objects** > **Line Item**.
2. Scroll down to the Custom Fields & Relationships related list, and click **New**.
3. Choose `Formula` as the field type, and click **Next**.
4. In the `Field Label` and `Field Name` fields, enter `Value`.
5. In the `Formula Return Type` field, choose **Currency**.
6. Click **Next**.
7. In the `Insert Merge Field` drop-down list, select **Unit_Price**. You should now see `Unit_Price__c` in the text box.
8. Click the **Insert Operator** drop-down list and choose **Multiply**.
9. In the Insert Merge Field drop-down list, select **Units Sold**. You should now see `Unit_Price__c * Units_Sold__c` in the text box.

18

10. Click **Next**, click **Next** again, and then click **Save**.

When you return to the detail page for the custom object, you can see there's a new field called Value.

Tell Me More....

The Formula field type is great for automatically deriving field values from other values, as you have done here. The formula you entered was quite straightforward: a simple multiplication of two field values on the same record. There's also an Advanced Formula tab, which allows you to do much more with these formulas.

Step 2: Calculate a Value for the Invoice Statement With a Roll-Up Summary Field

Now that you have the total for each line item, it makes sense to add them all to get the invoice total. Because the line items have a master-detail relationship with the invoice statement, we can use a roll-up summary field to calculate this value. Roll-up summary is a special type of field that lets you aggregate information about related detail (child) objects. In this case, you want to sum the value of each line item.

1. Navigate back to the Invoice Statement custom object page by clicking *Your Name* > **Setup** > **Create** > **Objects** > **Invoice Statement**.
2. Scroll down to the Custom Fields & Relationships related list, and click **New**.
3. Choose `Roll-Up Summary` as the data type, and click **Next**.
4. In the `Field Label` field, enter `Invoice Value` and click **Next**.
5. In the `Summarized Object` list choose **Line Items**.
6. For `Roll Up Type` field, choose **Sum**.
7. In the `Field to Aggregate` list choose **Value**.
8. Verify that your screen looks like the following and then click **Next**.

9. Click **Next** again and then click **Save**.

Tell Me More....

If you navigate back to an invoice statement record, you'll see your new roll-up summary field in action, displaying the total value of all the invoice line items. If the field shows an hourglass, wait a few seconds and then refresh. You can test this new functionality by adding a new line item.

Step 3: Check Inventory With a Validation Rule

The fields that you define in objects can have validation rules, written in the same formula language that you used to create the formula field. The validation rules can be used to determine what range of input is valid, and to display a message to the user if a field value is not valid. The error condition formulas should evaluate to true when you want to display a message to the user.

1. Navigate back to the Line Item custom object page by clicking *Your Name* > **Setup** > **Create** > **Objects** > **Line Item**.
2. Scroll down to the Validation Rules related list, and click **New**.
3. In the `Rule Name` field, enter `Order in stock`.
4. In the Error Condition Formula area, click **Insert Field** to open the Insert Field popup window.

   a. Select `Line Item >` in the first column.
   b. Select `Merchandise >` in the second column.
   c. Select `Total Inventory` in the third column.
   d. Click **Insert**.
   e. Type the less-than symbol < so that the formula looks like this:

   ```
   Merchandise__r.Total_Inventory__c <
   ```

    f. Click **Insert Field** again.

    g. Select `Line Item >` in the first column.

    h. Select `Units Sold` in the second column.

    i. Click **Insert** and verify the code looks like the following.

```
Merchandise__r.Total_Inventory__c < Units_Sold__c
```

5. Click **Check Syntax** to make sure there are no errors. If you do find errors, fix them before proceeding.

6. In the `Error Message` field, enter `You have ordered more items than we have in stock.`

7. For the `Error Location`, select `Field`, then choose `Units Sold` from the drop-down list.

8. Click **Save**.

Tell Me More....

You can enter a formula directly into the Error Condition Formula area, but as you saw here, you can easily traverse the available objects and find the components you need for the formula. Let's analyze the formula you created.

*   `Mechandise__r`—Because the Merchandise object is related to the Line Item object, the platform automatically provides a relationship field that lets you navigate from a Line Item record to a Merchandise record; that's what the `Mechandise__r` is doing.
*   `Total_Inventory__c`—This is the field you created to track the total amount of stock on a Merchandise record.
*   `Merchandise__r.Total_Inventory__c`—This tells the system to retrieve the value of Total Inventory field on the related Merchandise record.
*   `Units_Sold__c`—This refers to the Units Sold field on the current (Line Item) record.

Putting it all together, the formula checks that the total inventory on the related merchandise record is less than the number of units being sold. As indicated on the Error Condition Formula page, you need to provide a formula that is true if an error should be displayed, and this is just what you want: it will only be true when the total inventory is less than the units sold.

Step 4: Test the Validation Rule

Now test the validation rule you created in Step 3.

1. Click on the Invoice Statements tab and select an existing invoice.
2. Scroll down to Line Items and click **Edit** next to one of the line items.
3. Modify the `Units Sold` value to a number larger than the amount in stock.
4. Click **Save**. You should see an error indicating that there are not enough items in stock. Click **Cancel**.

Step 5: Improve the Validation Rule

The logic in the validation formula is a little bit flawed! Imagine that you edit a record and decrease the number of units sold. In this case, you shouldn't need to check inventory. Or if you increased the number of units sold, you should only need to verify that the number of additional items is in stock. You can improve the validation rule to include these scenarios by using the ISNEW() function in your formula, which determines whether you are creating a new record. If it's not a new record, you'll need to use the PRIORVALUE() function to tell you the prior value of the field before it was modified.

1. Navigate back to the Line Item custom object page by clicking *Your Name* > **Setup** > **Create** > **Objects** > **Line Item**.
2. Scroll down to the Validation Rules related list, and next to `Order in stock` click **Edit**.
3. Replace the existing formula with the following.

```
IF(
ISNEW(),
   Merchandise__r.Total_Inventory__c <  Units_Sold__c ,
   IF (
     Units_Sold__c < PRIORVALUE(Units_Sold__c),
     FALSE,
     Merchandise__r.Total_Inventory__c < (Units_Sold__c - PRIORVALUE(Units_Sold__c))
   )
)
```

4. Click **Check Syntax** to make sure there are no errors.
5. Click **Save**.

Tell Me More....

Let's look at this formula in a little more detail.

- `IF` is a conditional, ensuring that at run time one of the two branches will be taken depending on the condition.
- `ISNEW` is the condition. This function returns true if you're creating a new record, false otherwise.
- If `ISNEW` is true, then you simply check inventory as you did previously. For new records, you need to check that you have not sold more than you have in stock.
- If `ISNEW` is false, you know that you're performing an update to an existing record, not creating a new record, so you perform another conditional check, this time to determine if the number of units has gone up or down, by comparing it to its prior value using the `PRIORVALUE` function.
- If the prior value was greater, then you've updated the record and decreased its units. Since you have enough stock, you return `FALSE` to indicate that there's no validation failure.
- If the prior value was smaller, then you've increased the number of units, so you have to check whether there's enough inventory to hold the difference between the new number of units and the old.
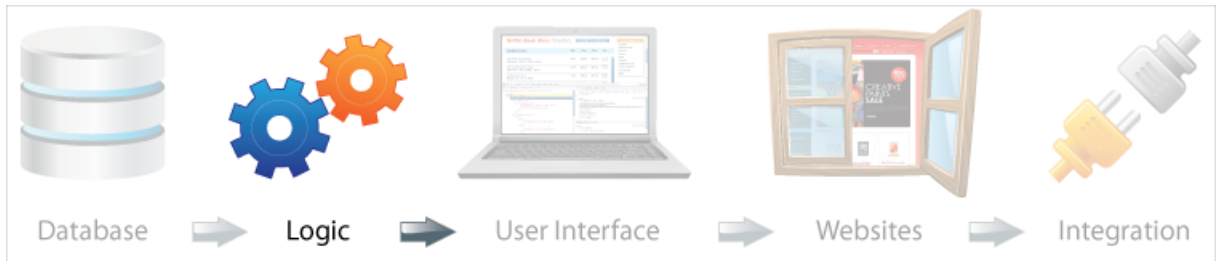
Summary

In this tutorial you created formula fields and validation rules that enhance and validate your app's data. For the first formula you created a subtotal for each line item by multiplying the price of a product by the number of units that were ordered. An invoice could be made up of multiple line items, so you created a roll-up summary field to automatically add up the line item subtotals.

You also learned how to define a validation rule to make sure you have enough items in stock. The first version of the rule was simple, but you enhanced it to work with both new records, and existing records with updates. These kinds of modifications can be applied to any field in your app when you want to make sure the data entered meets certain criteria.

Tutorial #4: Automating Processes Using Workflow

Level: Intermediate; **Duration:** 20–25 minutes

Your company can operate more efficiently with standardized internal procedures and automated business processes. In your Force.com app, you can use workflow rules to automate your procedures and processes. Workflow rules can trigger actions (such as email alerts, tasks, field updates, and outbound messages) based on time triggers, criteria, and formulas.

In this tutorial, you create and test two workflow rules. The first carries the current price of a merchandise item into the newly created line item record, and the second updates the inventory whenever you update a product line item.

Prerequisites

Formulas and Validation

You first need to create the roll-up summary field and validation rules as described in Tutorial #3: Using Formulas and Validation Rules on page 18.

Step 1: Populate the Line Item Price Using a Workflow Rule

At this point, you've created the Line Item and the Merchandise objects with Price fields. You've also marked the Line Item Price field as read-only, though you probably didn't notice that in the user interface because your default user is an administrative user, who has access to all data. Now you'll create a workflow rule to populate the unit price on the line with the unit price on the merchandise at the time the line item is created. You can create this complex logic quite easily with a declarative workflow rule.

1. Click *Your Name* > **Setup** > **Create** > **Workflow & Approvals** > **Workflow Rules**. If you see the Understanding Workflow page, click **Continue**.
2. On the All Workflow Rules page, click **New Rule**.
3. In Step 1 of the Workflow Rule wizard, select `Line Item` as the object and click **Next**.
4. In the `Rule Name` field, enter `Line Item Created`.
5. For `Evaluate the rule when a record is:`, select **created**.
6. In the `Rule Criteria` field, select `formula evaluates to true`.
7. In the text box, enter `true`.
8. Make sure your page looks like the following and then click **Save & Next**.

9. In Step 3 of the Workflow Rule wizard, in the Immediate Workflow Actions section, click **Add Workflow Action** and choose `New Field Update`. The New Field Update wizard opens.

10. In the `Name` field, enter `Insert Merchandise Price`.

11. In the `Field to Update` drop-down list, select `Line Item` in the first box and `Unit Price` in the second.

12. Select `Use a formula to set the new value`.

13. Click **Show Formula Editor**.

14. Click **Insert Field**.

15. Select `Line Item >` in the first column, `Merchandise >` in the second column, and `Price` in the third column.

16. Click **Insert**.

17. Click **Save** to close the New Field Update wizard and return to Step 3 of the Workflow wizard.

18. On the Specify Workflow Actions page, click **Done**.

19. On the Workflow Rule page, click **Activate**.

> **!** **Important:** Forgetting to activate a new workflow rule is a common mistake. If your rule is not active, its criteria are not evaluated when records are created or saved.

Tell Me More....

If you now create a new line item, save it, and then view it, you will see that line item's unit price will automatically be set to the price of the related merchandise record. Remember that because you are an administrator, you see an input field for price. But your users won't see that field because it was set to read-only.

The reason this workflow rule runs only when a record is created (and never afterward) is that we don't want to increase the invoice price of an item after the item has been added to the invoice. It would confuse a customer to suddenly find out the item's price is higher than what was advertised. However, if the item's price goes down, the customer would appreciate being updated with the new price. You'll make that happen later.

You might be wondering why we've created a rule whose formula always evaluates to true. The reason is that we want the field update to happen every time a record is created, and that's always true. You can also use a formula to evaluate an expression; it just wasn't necessary this time.

Step 2: Update Total Inventory When an Order is Placed

The inventory of Merchandise should be automatically maintained as orders are placed. When you create a new invoice ("Open" status), every new line item needs to decrease the total inventory by the number of units sold. Similarly, updates to an existing line item need to update the Total Inventory by the difference in units sold. You'll do this with another workflow rule.

1. Click `Your Name` > **Setup** > **Create** > **Workflow & Approvals** > **Workflow Rules**.
2. If you see the Understanding Workflow screen, click **Continue**; otherwise proceed to the next step.
3. On the All Workflow Rules page, click **New Rule**.
4. In Step 1 of the Workflow Rule wizard, select `Line Item` as the object and click **Next**.
5. In the `Rule Name` field, enter `Line Item Updated`.
6. For `Evaluate the rule when a record is:` select `created, and every time it's edited`.
7. In the `Rule Criteria` field, select `criteria are met`.
8. In the `Field` drop-down list, select `Invoice Statement: Status`. Then for `Operator`, select `equals`. Under `Value` click the lookup icon and choose `Open`. Click **Insert Selected**.
9. Make sure your screen looks like the following and then click **Save & Next**.

10. Click **Add Workflow Action** and choose **New Field Update**. The New Field Update wizard opens.

11. In the `Name` field, enter `Update Stock Inventory`.

12. In the first `Field to Update` drop-down list, select `Merchandise`. In the second, select `Total Inventory`.

13. Select `Use a formula to set the new value`.

14. Click **Show Formula Editor** and enter the following code.

```
IF (
 ISNEW(),
  Merchandise__r.Total_Inventory__c -  Units_Sold__c ,
  Merchandise__r.Total_Inventory__c - (Units_Sold__c - PRIORVALUE(Units_Sold__c))
)
```

15. Click **Check Syntax** and make corrections if necessary.

16. Click **Save** to close the New Field Update wizard and return to Step 3 of the Workflow wizard.

17. On the Specify Workflow Actions page, click **Done**.

18. On the Workflow Rule page, click **Activate**.

Step 3: Test the Workflow Rules

You can check to make sure your workflows are functioning as expected by creating a new line item. Then verify that its unit price is automatically set and that the total inventory on the stock has decreased.

1. Click the Invoice Statements tab and then click the invoice record you created earlier.

2. Select `New Line Item` and enter the following values.

- In the `Line Item Number` field enter 2.
- Leave the `Unit Price` field blank.
- Set `Units Sold` to 1000.
- Next to the `Merchandise` field, click the lookup icon and select the Wee Jet merchandise record.

3. Click **Save**.

4. Click line item 2. Note that the unit price has been set to the same value as on the Wee Jet record.

5. Click the **Wee Jet** link. Notice that the `Total Inventory` is down to 1000 from 2000.

Summary

Workflow rules help you automate business processes and enforce standards. The workflow rule you created automatically updates price and inventory across different objects.

Tutorial #5: Creating an Approval Process

Level: Intermediate; **Duration:** 20–25 minutes

Database → Logic → User Interface → Websites → Integration
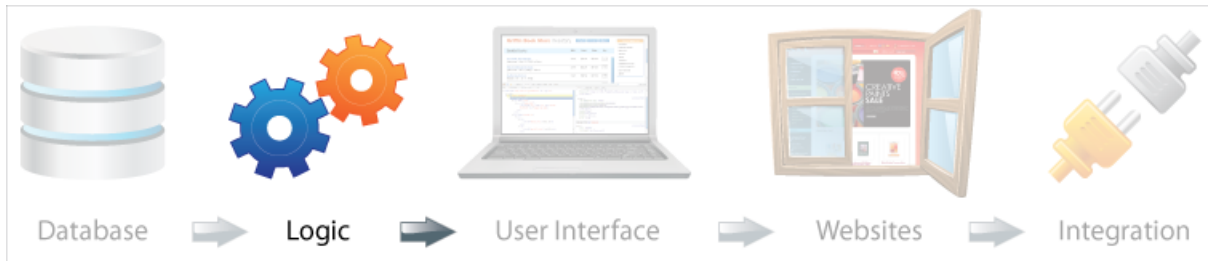
Your company can operate more efficiently with standardized internal procedures. In your Force.com app, you can use approval processes to enforce consistency and compliance within your company's business practices. Approval processes can automate all of your organization's approvals—from simple to complex. In this tutorial, you create an approval process that requires explicit approval from the manager if an invoice is over $2000.

Prerequisites

Formulas and Validation

You first need to create the roll-up summary field and validation rules as described in Tutorial #3: Using Formulas and Validation Rules on page 18.

Step 1: Create an Email Template

One of the business rules that you want to enforce in your workflow is that all invoices totaling over $2000 must be approved by a manager. To implement this rule you need to create two additional elements: an email to be sent to the manager when an invoice exceeds $2000, and an approval process for the manager to follow.

Note: If your default currency is not set to US dollars, just assume we're talking about whatever currency you're using.

In this step you create the email template that will be used by the workflow rule to generate and send the email. You create the approval process in the next steps.

1. Click **`Your Name`** > **Setup** > **Communication Templates** > **Email Templates**.
2. Click **New Template**.
3. In Step 1 of the Email Template wizard, select `Text` and click **Next**.
4. Select `Available For Use`.
5. In the `Email Template Name` field, enter `Large Invoice Template`.
6. In the `Subject` field, enter `A large invoice has been submitted`.
7. In the `Email Body` field, enter the following code and text.

```
{!Invoice_Statement__c.OwnerFullName} submitted for approval an Invoice Statement that
totalled {!Invoice_Statement__c.Invoice_Value__c} on
{!Invoice_Statement__c.LastModifiedDate}
```

8. Verify that your screen looks like the following, and then click **Save**.

Tell Me More....

The email body text that you just entered supports merge fields, for example
`{!Invoice_Statement__c.Invoice_Value__c}`. At the time the email is generated, this value will be dynamically substituted with the actual invoice value that generated the approval process.

Step 2: Create an Approval Process

In this step you create an approval process that requires explicit approval from the manager if an invoice is over 2000. Creating and using an approval process is just as easy as creating a workflow rule.

1. Click `Your Name` > **Setup** > **Create** > **Workflow & Approvals** > **Approval Processes**.
2. In the `Manage Approval Processes For` drop-down list, choose `Invoice Statement`.
3. Click **Create New Approval Process** and choose `Use Jump Start Wizard` from the drop-down list.
4. In the `Name` field, enter `Large Invoice Value`.
5. Next to the `Approval Assignment Email Template` field, click the lookup icon and select the `Large Invoice Template` you just created.
6. Enter the following values in the Specify Entry Criteria area.

    a. In the `Field` drop-down list, choose `Invoice Value`.
    b. In the `Operator` drop-down list, choose `greater than`.
    c. In the `Value` field enter `2000`.

7. Select `Automatically assign an approver using a standard or custom hierarchy field` and choose `Manager` for the hierarchy field. This ensures that if a particular user starts the approval process, the user's manager is assigned as the approver. You'll create a user named Bob Smith in the next step.

8. Click **Save**.

9. You'll see a warning that you must activate the approval process. Click **OK**.

   You've finished creating the approval process, but before it can run, you have to define what happens to records when they're first submitted, when they're approved, and when they're rejected.

10. Click **View Approval Process Detail Page**.

11. Add a field update to the Initial Submission Actions related list:

    a. Click **Add New** > **Field Update**.
    b. For `Name`, enter `Set Initial Approval Status`.
    c. For `Field to update`, select `Status`.
    d. Under Picklist Options, select `A specific value` and choose `Pending`.
    e. Click **Save**.

12. Add a field update to the Final Approval Actions related list:

    a. Click **Add New** > **Field Update**.
    b. For `Name`, enter `Set Final Approval Status`.
    c. For `Field to update`, select `Status`.
    d. Under Picklist Options, select `A specific value` and choose `Closed`.
    e. Click **Save**.

13. Add a field update to the Final Rejection Actions related list:

    a. Click **Add New** > **Field Update**.
    b. For `Name`, enter `Set Final Rejection Status`.
    c. For `Field to update`, select `Status`.
    d. Under Picklist Options, select `A specific value` and choose `Open`.
    e. Click **Save**.

14. Click **Activate** to activate the approval process.

Tell Me More....

Default actions are associated with a few of the approval steps. The Initial Submission actions and Final Approval actions both lock the record, while the Final Rejection action unlocks it. This makes sense: if someone submits a record for approval, it shouldn't be changed in the meantime. Likewise, if the record is rejected, it should be unlocked so changes can be made.

Step 3: Create a Custom Profile

A profile is a collection of permissions and other settings associated with a user or a group of users. Your organization has a number of standard profiles already defined. If you create an app, the permissions and settings to access the app and associated objects are disabled for most profiles. This security setting ensures that access to the app and its data is only explicitly granted to users. You can change object permissions in custom profiles, but not standard profiles.

In this step, you'll create a custom profile that you can assign to users who need to access the Warehouse app and its custom objects. Then in your new custom profile, you'll enable access settings and object permissions so your users can access the app. You'll create the new user in the next step.

1. Click *Your Name* > **Setup** > **Manage Users** > **Profiles**.

2. Select **Clone** next to Standard User.

> **Note:** Each user profile is available for a specific user license. Similarly, the custom profile that you obtain by cloning an existing user profile is also available for the same user license as the user profile it is cloned from. The Standard User profile is available for the Salesforce user license. If you've used up all Salesforce user licenses for your organization already, you can clone the Salesforce Platform User profile or the Force.com — Free User profile instead. To find out what user licences are available for your organization, click *Your Name* > **Setup** > **Company Profile** > **Company Information** and check the User Licenses section.

3. In the Profile Name field, type Basic User.

4. Click **Save**.

5. On the detail page for your new profile, click **Edit**.

6. Under Custom App Settings, select **Visible** for the Warehouse app.

> **Note:** If you cloned the Force.com — Free User profile for your custom profile, select **Default** for the Warehouse app instead since only one custom app can be selected.

7. Under Custom Tab Settings, select **Default On** for Merchandise. (Invoice Statements should already be set to Default On.)

8. Scroll to the bottom of the profile edit page, and under Custom Object Permissions, select the **Read**, **Create**, **Edit**, and **Delete** boxes for the Invoice Statements, Line Items, and Merchandise objects.

9. Click **Save**.

Tell Me More....

You've just seen how easy it is to create and edit a custom profile. If you need to edit many profiles, you can use enhanced profile list views to create a custom list view of your profiles and then edit the profiles from the list. For more information, see "Editing Multiple Profiles with Profile List Views" in the Salesforce online help.

Step 4: Create a User

When you create a Force.com app, it's automatically enabled to allow multiple users to log in. Now you can customize the app by configuring it to function differently depending on the profile of the user logging in. For example, you can grant read-only access to fields to a certain group of users, or make them completely invisible. You can also strictly enforce data sharing, so that a user sees only his or her records or those of his or her manager.

In this step, you create a new user and link that user to your current user's account via the Manager field. You use this configuration to make sure that if the new user creates an invoice that meets certain conditions, the invoice is routed to his or her manager.

1. Click `Your Name` > **Setup** > **Manage Users** > **Users**.
2. On the All Users page, click **New User**.
3. Enter the following information:

   - In the `First Name` field, enter `Bob`.
   - In the `Last Name` field, enter `Smith`.
   - In the `Alias` field, enter `bsmith`.
   - In the `Email` field, enter your own email address, so that you will receive the approval requests routed to Bob Smith.
   - The `Username` field defaults to your email address, but you'll need to create a unique username for Bob, in the form of an imaginary email address.

     Note: Write down Bob's username (his imaginary email address), because you'll be logging in as him shortly.

   - In the `Manager` field, select the user you created when you signed up for your organization.
   - In the `User License` field, select `Salesforce`.

     Note: You won't see Salesforce listed in the `User License` field if you've already used up all the Salesforce licenses allocated for your organization. For this step, you should select the same user license as the one used by the cloned profile, Basic User, that you created in the previous step.

• In the `Profile` field, select `Basic User`.

4. Click **Save**.

You should now receive an email confirming the creation of the new user. There's one more setup step to complete the approval process, so don't log in as Bob Smith yet or you'll have to immediately log back in as the administrator.

Step 5: Test the Approval Process

We can now see the approval process in action. If Bob creates an invoice, and the total value of the invoice is greater than $2000, he;; be able to click Submit for Approval on the invoice statement. This will send an email to his manager and lock the record, preventing Bob from making any more changes.

When the manager logs in, he or she can approve (or reject) the invoice. If the manager approves the invoice, its status is updated to Closed. And if not, you want it remains in the original Open status. In a real-world scenario, your business process might require you to email the invoice owner and ask for confirmation, and so on.

Before testing the approval process, make sure that your Home page can display items that require approval.

1. Click *Your Name* > **Setup** > **Customize** > **Home** > **Home Page Layouts**.
2. Click **Edit** next to your home page layout.
3. Select the `Items to Approve` option, if it is not already selected.
4. Click **Next**, and then click **Save**.

While developing the app, you've been logged in as an administrator. To create a new record and test the approval process, you need to log out of your administrator account and log back in as a standard user.

1. Click *Your Name* > **Logout**
2. Now log in as Bob Smith. If this is your first time logging in as Bob, you may have to change your password.
3. Select the Warehouse app. The Warehouse app will be the default app after logging in if Bob Smith's account is based on a Force.com — Free user license.
4. Click the Invoice Statements tab.

Note: If you don't see any tabs, then you need to change Bob's user type to Basic User. You'll need to log out and then log back in as an admin, as described in the previous step.

5. Click **New** and create an invoice statement.
6. Add a **New Line Item** and enough items to make the invoice value greater than $2000.
7. Click **Save**.
8. Click **Submit for Approval**.
9. Log out of the app.

An email is sent to Bob Smith's designated manager who, for purposes of this tutorial, is you the administrator. Since you're the manager, you can check your email to verify this result.

1. Log in again as the administrator (not Bob).
2. Click the **Home** tab.
3. The pending approval request is at the bottom, you may have to scroll down to see it. The record is locked until you approve the invoice. Go ahead and **Approve** the invoice.

Tell Me More....

From your Approval Process detail page, you can see a nice diagram of the approval process and the actions that will fire at each step.

1. In the familiar *Your Name* > **Setup** area, click **Create** > **Workflow & Approvals** > **Approval Processes** and select the process you created.
2. Click **View Diagram**. The approval process diagram appears in a separate browser window.
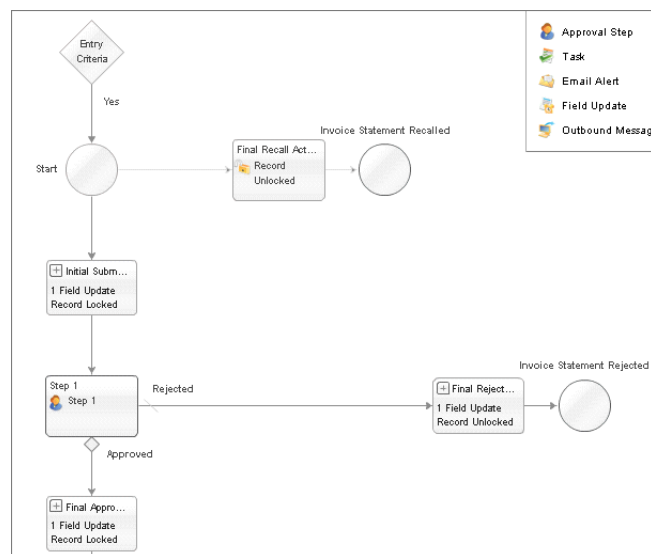
Summary

Approval processes help you enforce your business standards. This email alert was a fairly simple example, but you can change the conditions for when an approval process runs, and the actions that need to be taken.

Tutorial #6: Creating Reports and Dashboards

Level: Beginner; **Duration:** 50–60 minutes

You've built an app that captures and manipulates warehouse data, but you want to report on that data as well. Force.com provides a drag-and-drop report builder that lets you quickly organize and present your data. It's easy to group and summarize your data, and add formulas and charts. You can then share these reports to help business users make more informed decisions.

You may have noticed the Reports and Dashboards tabs that were automatically added when you created your app. The Reports tab provides access to a set of predefined reports, all of the reports that you create, and the reports in folders that you can access. The Dashboards tab shows data from charts, gauges, tables, metrics, or Visualforce pages. Dashboards provide a snapshot of key metrics and performance indicators for your organization.

In this tutorial, you will display your data in reports using three different formats. The first format, tabular, lists your data in a simple table. The next two let you group your data and add a chart. You will also create a dashboard using one of your reports.

Prerequisites

Warehouse App

First create the basic Warehouse app in Tutorial #1 on page 3, then add relationships as described in Tutorial #2: Adding Relationships on page 10 and add workflow as described in Tutorial #4: Automating Processes Using Workflow on page 24.

Data

The reports you create in this tutorial will look better if you have more data. Go ahead and create one or two more merchandise items, and at least one more invoice.

Step 1: Create a Tabular Report

In this step you will create a tabular report. Tabular reports present data in simple rows and columns, much like a spreadsheet. They don't contain groupings, but can be used to show column summaries, like sum, average, maximum, and minimum.

1. Click the Reports tab.
2. Click 📄 and from the drop-down, select **New Report Folder**.
3. In `Report Folder Label`, enter `Merchandise Reports`.
4. Click **Save**.
5. Click **New Report**.
6. In the Select Category panel, select `Other Reports`.
7. In the Select Report Type panel, select `Merchandise` and click **Create**. Report types set the rules for which records to show in reports, based on object relationships. They also determine which fields you can use.

   The report builder appears, with its dynamic preview built from a limited number of records. The `Merchandise Name` field is in the report by default. The default report format is tabular.
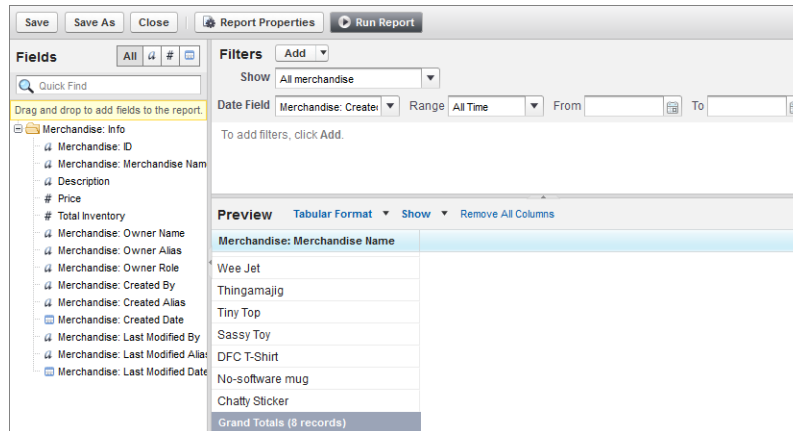
8. Drag each of the following fields from the Fields pane to the Preview pane, placing them beside the Merchandise Name column. To select multiple fields or columns, press CTRL (Windows) or Command (Mac).

   • `Description`

   • `Price`

   • `Total Inventory`

You now have a basic report, but let's add a few bells and whistles like sums and averages.

1. Click [▼] next to Price header and select `Summarize this Field`.
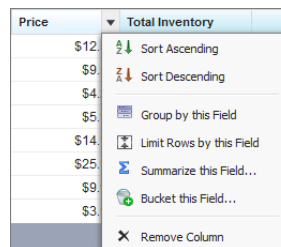
2. Select `Average`.
3. Click **Apply**.
4. Click [▼] next to Total Inventory and select `Summarize this Field`.
5. Select `Sum`.
6. Click **Apply**.

Now, set the scope for the report's data and run it.

1. In the Filters pane, select `All Merchandise` from the View drop-down list to assign the set of records that the report will use.
2. Click **Run Report**.
3. Review the data and click **Save As**.
4. In the `Report Name` field, enter `All Merchandise with Price and Inventory`.
5. Select `Merchandise Reports` from the Report Folder drop-down list.
6. Click **Save and Return to Report**. You will now see your All Merchandise with Price and Inventory report displayed.

Merchandise: Merchandise Name	Description	Price	Total Inventory
Widge It	A thingy	$12.50	1,500
Wee Jet	A small plane	$9.50	2,000
Thingamajig	A very useful tool	$4.20	500
Tiny Top	Small spinning top	$5.99	500
Sassy Toy	Stuffed toy	$14.99	2,000
DFC T-Shirt	Developer Force shirt	$25.99	485
No-software mug	Coffee mug with logo	$9.99	650
Chatty Sticker	Fun stickers	$3.99	3,900
Grand Totals (8 records)			
		avg $10.89	11,535

The procedures you followed in creating this report will be similar to those you follow in the later steps. You selected the objects that you want to report on, selected the report format, determined which fields to display, optionally added summary data, and set the scope of data for the report. The other reports in this tutorial follow similar instructions, with the addition of grouping and charts.

Tell Me More....

- You can order the report data by clicking the column headers to toggle between ascending and descending order. The Grand Totals area of the report indicates the record count as well as the summaries you chose. You can make additional changes to this report by clicking **Customize**.
- You can click through to the data records that are being reported on, a characteristic found in all reports on Force.com. For example, click the name of any merchandise record listed in the report to view its detail page.
- A report folder's sharing settings determine who can access reports in that folder. Modify these settings by clicking **Edit** on the folder in the Reports tab. While you can't give individual users access, you can share the folder with Public Groups, Roles, or Roles and Subordinates. For more information, search "Managing Folders" in the online help.
- You can return to your reports at any time by selecting the Reports tab, selecting the Merchandise Reports folder, and clicking **Go**.

Step 2: Create a Summary Report

You can group report data by up to three levels and add a chart. In this step you'll create a summary report, group data by two levels, and add a combination chart.

First, we'll create a summary report with one grouping.

1. Click the Reports tab.
2. Click **New Report**.
3. Select `Other Reports`, then `Invoice Statements with Line Items and Merchandise`.
4. Click **Create**.
5. The default format is tabular, but we want a summary report. Click **Tabular Format** and choose `Summary` instead.
6. From the Fields pane on the left, within the Line Item folder, select the `Unit Price` field, drag it to the Preview pane, and drop it on the right of `Merchandise Name`. A green line will appear when you can drop the field to create a new column.
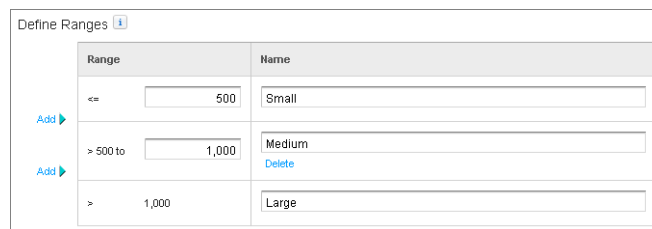
   Warning: If you drop `Unit Price` or `Units Sold` into a grouping by mistake, click the down arrow next to the field name and choose **Remove Group**. You need to drop the field into a column, which is located just above the grouping zone.

7. Click the down arrow ▼ next to `Unit Price`, select `Summarize this Field`, choose `Average`, and then click **Apply**

8. Again in the Fields pane on the left, within the Line Item folder, select the `Units Sold` field, drag it to the Preview pane, and drop it on the right of `Unit Price`.

9. Click the down arrow ▼ next to `Units Sold`, select `Summarize this Field`, choose `Sum`, and then click **Apply**

10. Select the `Merchandise Name` field (either from Fields or Preview panel) and drag it to the area labeled **Drop a field here to create a grouping**. This aggregates data by the unique merchandise item.

The report now includes all the invoices and their associated line items grouped by merchandise item, as well as average price and total units sold.

Next, we'll add another grouping. Say you also want to group data by invoice size. Invoice size is not an available field in the report. Instead of creating a formula or custom field, you can bucket an existing field to get invoice size. We'll create a *Bucket Field* based on `Invoice Value` with ranges for small, medium, and large. We'll use the bucket field to create the grouping.

1. From the Fields pane, drag `Invoice Value` to the left of Unit Price in the Preview pane.

2. Click ▼ on Invoice Value and select `Bucket this Field`.

3. Enter a bucket field name, `Invoice Size`.

4. Define ranges for invoice size as `Small` (500), `Medium` (between 500–1000), and `Large` (greater than 1000).

5. Click **OK**. Locate the bucket field in the Fields pane.
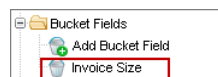
6. Drag the bucket field and drop it above Merchandise Name in the Preview pane for a second grouping.

The report shows data grouped in two levels—first, by invoice size with average price and total units sold. Second, within each first level group, you see invoices and line items by merchandise name showing average price and total units sold.

Invoice Statement: Invoice Number	Line Item: Line Item Number	Invoice Value	Unit Price	Units Sold
▼ ⬡ Invoice Size: Small (1 Record)				
			avg $14.99	30
▼ Merchandise: Merchandise Name: Sassy Toy (1 Record)				
			avg $14.99	30
INV-0013	3	$449.70	$14.99	30
⬡ Invoice Size: Medium (1 Record)				
			avg $5.99	100
Merchandise: Merchandise Name: Tiny Top (1 Record)				
			avg $5.99	100
INV-0014	50	$599.00	$5.99	100
⬡ Invoice Size: Large (4 Records)				
			avg $8.62	264
Merchandise: Merchandise Name: Thingamajig (1 Record)				
			avg $10.00	6
INV-0001	1	$1,099.00	$10.00	6
Merchandise: Merchandise Name: Wee Jet (2 Records)				
			avg $9.75	250
INV-0001	3	$1,099.00	$9.99	100
INV-0015	60	$1,425.00	$9.50	150
Merchandise: Merchandise Name: Widge It (1 Record)				
			avg $5.00	8
INV-0001	2	$1,099.00	$5.00	8
Grand Totals (6 records)				
			avg $9.25	394

Now let's add a fancy combination chart to the report:

1. In the Preview pane, click **Add Chart** to create a chart to represent your data. The Chart Editor dialog box appears.
2. Click the vertical bar chart.
3. In the `Y-Axis` drop-down list, select `Sum of Units Sold`.
4. In the `X-Axis` drop-down list, select `Merchandise: Merchandise Name`. Notice the bucket field, Invoice Size, is also available as there are two groupings.
5. Select `Plot additional values`.
6. In the `Display` drop-down list, select `Line`.
7. In the Value drop-down list, select `Average of Unit Price`.
8. Select `Use second axis`.
9. Click **OK**.
10. Click **Save**.
11. In the `Report Name` field, enter `Merchandise Sold by Invoice No & Price`.
12. In the `Report Description` field, enter `What merchandise has been selling and what's the average unit price?`
13. In the `Report Folder` drop-down list, select `Merchandise Reports`.
14. Click **Save and Run Report**.

The combination chart shows the data, plotting each merchandise item against units sold (across all invoices) as well as superimposing a second graph mapping the average unit price for the item.
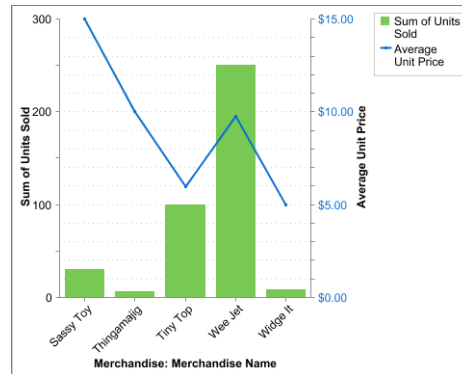
Tell Me More....

You can check rows of interest on your report and click **Drill Down** to filter the report to only those rows.

In the report builder, you can filter a bucket field just like other fields in the report. In this step, for example, drag `Invoice Size` onto the Filters pane. By setting Invoice Size `not equal to Small`, you only see data for line items whose invoice value is either medium or large.

Try adding a *Cross Filter* from the **Add** drop-down in the Filters pane . A cross filter lets you filter on the report's child objects using a simple `with` or `without` condition. In this tutorial, you can add a cross filter to report on line items without invoice statements, for example.

Step 3: Create a Matrix Report

In this step, you will create a matrix report that groups and summarizes both columns and rows. This report will show summary data of merchandise items per day and across different invoice statement status values.

1. Click the Reports tab.
2. Click **New Report**.
3. Select `Other Reports`, then `Merchandise with Line Items and Invoice Statements`.
4. Click **Create**.
5. In the Preview panel, click **Tabular Format** and choose `Matrix` from the drop-down list.
6. Clear the slate by selecting **Remove all Columns** and then click **OK** in the popup.
7. Double-click `Invoice Statement: Invoice Value` to add it to the report. In the popup, select the `Sum` checkbox.
8. Double-click `Line Item: # Units Sold` to add it to the report. In the popup, select the `Sum` checkbox.
9. Create a column grouping for the status of invoice statements by dragging the `Invoice Statement: Status` field to the column grouping drop zone.
10. Create a row grouping by dragging the `Invoice Statement: Created Date` field to the row grouping drop zone.
11. Create a secondary row grouping by dragging the `Merchandise: Merchandise Name` field to the second row grouping drop zone.
12. In the Filters pane, ensure that `All merchandise` is selected from the **Show** drop-down list.
13. Click **Run Report**.

You'll get an astounding amount of information generated in the report. Values are tallied both horizontally and vertically, and within the report, depending on the column (Invoice Status) and row groupings (Created Date and Merchandise Name).

	Invoice Statement: Created Date	Merchandise: Merchandise Name		Invoice Statement: Status Open	Negotiating	Grand Total
☐	8/2/2011	Thingamajig	Sum of Invoice Statement: Invoice Value	$1,099.00	$0.00	$1,099.00
			Sum of Units Sold	6	0	6
			Record Count	1	0	1
		Wee Jet	Sum of Invoice Statement: Invoice Value	$1,099.00	$9,812.50	$10,911.50
			Sum of Units Sold	100	1,000	1,100
			Record Count	1	1	2
		Widge It	Sum of Invoice Statement: Invoice Value	$1,099.00	$9,812.50	$10,911.50
			Sum of Units Sold	8	25	33
			Record Count	1	1	2
		Subtotal	Sum of Invoice Statement: Invoice Value	$1,099.00	$9,812.50	$10,911.50
			Sum of Units Sold	114	1,025	1,139
			Record Count	3	2	5
☐	1/31/2012	Sassy Toy	Sum of Invoice Statement: Invoice Value	$0.00	$449.70	$449.70
			Sum of Units Sold	0	30	30
			Record Count	0	1	1
		Subtotal	Sum of Invoice Statement: Invoice Value	$0.00	$449.70	$449.70
			Sum of Units Sold	0	30	30
			Record Count	0	1	1
		Grand Total	Sum of Invoice Statement: Invoice Value	$1,099.00	$10,262.20	$11,361.20
			Sum of Units Sold	114	1,055	1,169
			Record Count	3	3	6

Now let's add a combination chart to this report:

1. Click **Customize**.
2. Click **Add Chart**. The Chart Editor dialog box appears.
3. Select the Vertical Bar Chart.
4. In the Y-Axis drop-down list, select `Sum of Units Sold`.
5. In the X-Axis drop-down list, select `Invoice Statement: Created Date`.
6. In the Group-By drop-down list, select `Invoice Statement: Status`.
7. Select the `Stacked` graph type.
8. Select `Plot additional values`.
9. For Value, select `Record Count`
10. Select `Use second axis`.
11. Click the Formatting tab to further customize the chart.
12. For `Chart Title`, enter `Units Sold per Day by Invoice Status`.
13. For Legend Position, select `Bottom`.
14. Select `Show Axis Label`.
15. Click **OK**
16. Click **Save**.
17. For `Report Name`, enter `Daily Units Sold by Invoice Status`.
18. For `Report Description`, enter `How many units are we selling each day, by Invoice Status?`
19. For `Report Folder`, select `Merchandise Reports`.
20. Click **Save and Run Report**.

The resulting graph summarizes the units sold, and automatically takes into account the status of each invoice, grouping the invoices over daily periods. Note that you'll need invoices from more than one date to see anything fancy.
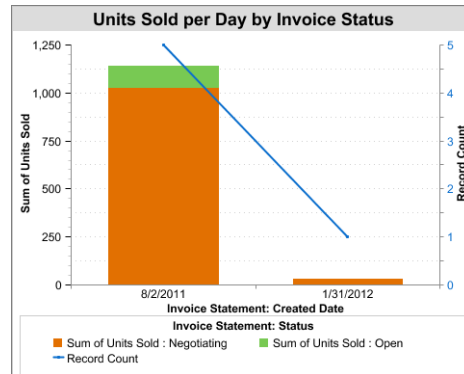
Tell Me More....

- The Sum of Units Sold has a scale on the left side of the chart that correlates to the columns of your chart. Record Count has its own scale on the right side of the chart that correlates to the blue line on your chart.
- The chart and report currently display all records, regardless of when they were created. If you want to narrow the time frame that is represented, you can define it in the upper right corner of the page. Choose either a predefined duration or enter your own custom dates and click **Run Report** to update the page.
- You can quickly change how your chart is displayed by clicking **Edit**.
- Because you defined multiple grouping levels in the report, you'll see multiple summary rows, as you see in Invoice Statement: Status.
- You can deselect **Show** > **Details** to only show summary information, making your report easier to see.
- Take reporting to the next level with the *Joined Report* format. It's available in report builder's Format drop-down on the Preview pane. In a joined report, you can combine multiple report types in a single report, group data, summarize fields, and add a chart.

Step 4: Create a Dashboard

You can show all the reports you've built in the same place by including them in a dashboard, which provides a graphical summary of the data in your app. Your data can be represented by charts, tables, metrics, gauges, and even Visualforce components. Similar to reports, dashboards are organized into folders.

1. Click the Reports tab.
2. Click ▣ˇ and from the drop-down, select **New Dashboard Folder**.
3. For `Dashboard Folder Label`, enter `Merchandise Dashboards`.
4. Click **Save**.
5. Click **New Dashboard**.
6. Click **Dashboard Properties**.
7. For `Title`, enter `Merchandise and Invoice Summary`.
8. Save the dashboard to the `Merchandise Dashboards` folder.
9. Click **OK**.
10. From the Components tab, drag the vertical bar chart to the first column.
11. From the Data Sources tab, click **Reports**, click **+** next to Merchandise Reports, and drag Daily Units Sold by Invoice Status on top of the vertical chart in the first column.

12. On the Vertical Bar Chart, click 🔧.

13. Select `Use chart as defined in source report`.

14. Click **OK**.

15. Add a header to the component by clicking **Edit Header** and entering `Items Sold per Day`.

Let's add another component to the dashboard.

1. From the Components tab, drag the pie chart to the middle column.

2. On the Data Sources tab, type, `merchandise sold`, in the search box.

3. Drag Merchandise Sold by Invoice No and Price on to the Pie Chart.

4. On the Pie Chart, click 🔧. In the Component Data tab, notice auto-selected choices for Values and Wedges. Values, here, inherit data from the summary field (sum of units sold) used in the source report chart.

5. On the Formatting tab, choose these settings.

    - **Sort Rows By**: `Value Ascending`
    - **Legend Position**: `Bottom`
    - **Data Labels**: `Show Values`

6. Click **OK**.

7. Edit the header and type, `Total Merchandise Sold`.

8. Click **Save** and in the pop up, click **Save and Run Dashboard**.

Your dashboard now has two components. You can add up to 20 components to create a more complex dashboard.
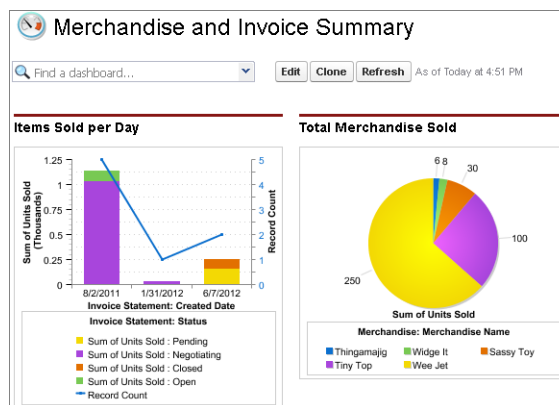
Let's add the dashboard to your Home tab

1. Click **Setup** > **Customize** > **Home** > **Home Page Layouts**.

2. Click **Edit** and select the `Dashboard Snapshot` checkbox.

3. Click **Next**, then **Save**. Select your Home tab to see it in action.

Tell Me More....

- When you set a running user for a dashboard, it runs using the security settings of that single, specific user. All users with access to the dashboard see the same data, regardless of their own personal security settings. To set the running user, click

  ▼ next to the `View dashboard as` field.

- Dashboards can be updated either manually or on a schedule, and can be delivered through email.

- A dashboard won't automatically refresh unless it is set to do so. Each time you view a dashboard, it indicates in the upper right corner when it was last refreshed. To refresh the data in the dashboard, click **Refresh**.
- Try adding a filter when editing the dashboard by clicking **Add Filter**. A filter lets you see different views of dashboard data based on filter conditions. You can add up to three filters per dashboard with up to 10 conditions on a filter. For example, you can filter the entire dashboard by the `Invoice Statement: Status` field. This will show you total and daily merchandise sold for each invoice status. Instead of filtering at the report level, you directly manipulate dashboard data.
- Get dashboards and their report data on the go by downloading Salesforce Mobile Dashboards for your iPad from the Apple App Store or AppExchange.

Summary

In this tutorial you created three reports: a quick tabular report that listed your records with simple column summaries, a summary report that grouped and graphed your data, and a matrix report that provided a multi-dimensional analysis of your data. You also created a dashboard, which you can place on your Home tab for quick reference.

Now that you've built the basic reports, experiment with them by modifying the filters, groupings, and charts. Creating meaningful reports is a skill well worth learning!

Tutorial #7: Adding Programmatic Logic with Apex

Level: Advanced; **Duration:** 20-30 minutes

Apex is a strongly-typed, object-oriented, Java™-like programming language that runs on Force.com. Apex is used to add programmatic business logic to apps. For example you can use it to write triggers, Web services, and program controllers in your app's user interface layer.

You've already added business logic using the declarative workflow environment. In this tutorial you'll create programmatic logic using Apex, which is ideal for juggling multiple records and complex logic.

The business case you'll be working on requires that when a price for a piece of merchandise goes down, you will pass the savings along to your customers. To do this, you'll create an Apex trigger that updates all open invoices whenever the merchandise price decreases. A trigger is a set of code that fires at a particular time in the lifecycle of a record. In this case, you will create a trigger that fires after a merchandise record is updated.

There are two ways to develop apps on Force.com: you can either use the online environment, as you've been doing up to this point, or use the Force.com IDE. If you don't have the IDE installed, it will take a moment to set up. However, if you are already familiar with the IDE, you know that it has syntax highlighting, code insight, and many other features that make development—especially team development—highly productive.

Prerequisites

Basic Knowledge

For this tutorial, it helps to have basic knowledge of an object-oriented programming language like Java or C#, but it is not required. This tutorial can also be completed in the Force.com IDE, so familiarity with Eclipse is useful but not necessary.

Software Requirements

You can use the Force.com IDE for this tutorial if you want to. In that case you'll need the Force.com IDE plugin: wiki.developerforce.com/index.php/Force.com_IDE. If you need help creating a project, see Creating a Project in the Force.com IDE.

"Developer Mode," "Modify All Data," and "Author Apex" permissions

Since this tutorial involves working with Apex, make sure that you have the proper permissions to create Apex classes.

Step 1: Create an Apex Trigger Definition

The first thing you need to do is create the trigger definition, which contains the trigger name, the affected object, and an action that fires the trigger. You can create triggers in the Web interface or in the Force.com IDE. Instructions are provided for both tools.

To create a trigger in the Web interface:

1. Click *Your Name* > **Setup** > **Create** > **Objects** in the sidebar menu.
2. Click your **Merchandise** custom object.
3. On the Merchandise detail page, scroll down to **Triggers** and click **New**.
4. Replace `<name>` and `<events>` so that the code matches the following:

```
trigger HandleProductPriceChange on Merchandise__c (after update) {

}
```

5. If you're using the Web interface, click **Quick Save**, which will save your work and let you continue editing. Saving your work at this point also verifies that you've entered the code correctly, because if you've made syntax errors, the system won't let you save.

To create the trigger in the Force.com IDE:

1. In the Package Explorer, right-click your project and click **New** > **Apex Trigger**.
2. In the dialog box, enter `HandleProductPriceChange` for the name.
3. In the `Object` drop-down list, choose `Merchandise__c`.

   Note: If this object doesn't appear in the list of object, click **Refresh Objects**. (Ignore the warning that pops up about how you must select an operation.)

4. Select `after update`.

5. Click **Finish** and the file opens in the Editor.

Tell Me More....

Your trigger doesn't do anything yet, but it's now ready to accept any logic that you want to execute when a Merchandise record is updated. Before we get to the trigger logic, let's break up the trigger definition and examine each part.

- `HandleProductPriceChange`—The name of the trigger.
- `on Merchandise__c`—The object the trigger acts on— in this case, your `Merchandise__c` custom object.
- `(after update)`—The action that fires the trigger. Apex triggers are fired in response to data actions, such as inserts, updates and deletes—either before or after one of these events. The trigger you defined fires after a record is updated.
- `{ }`—The code that goes between the curly brackets is called the *body* and determines what the trigger does. You'll code the trigger body next.

Step 2: Define a List Variable

The first thing you need to do is define a list variable that will hold a list of line items. You don't need all the line items, so you'll select only those that are in the set of records that triggered this code to run and have their status set to Negotiating.

1. Between the curly braces of your trigger definition, enter the following comment and then declare a list variable.

```
// update invoice line items associated with open invoices
List<Line_Item__c> openLineItems =
 [ ];
```

Note: For the sake of brevity, we won't comment everything here, but it's good practice to comment your code.

2. Between the square brackets, enter a query that pulls information from your Line Item custom object.

```
List<Line_Item__c> openLineItems =
 [SELECT j.Unit_Price__c, j.Merchandise__r.Price__c
 FROM Line_Item__c j
 WHERE j.Invoice_Statement__r.Status__c = 'Negotiating'
 AND j.Merchandise__r.id IN :Trigger.new
  FOR UPDATE];
```

Note: We try to make our code samples fit on the page, so there are often line breaks where they aren't strictly needed. Apex ignores white spaces, so if the format of your code doesn't look exactly the same, that's OK. Also, Apex isn't case sensitive, so if you like to use lowercase, you can enter `select` instead of `SELECT`, for example.

Tell Me More....

The `openLineItems` list holds a list of records from your `Line_Item__c` custom object. What's contained in that list is determined by the query inside the square brackets, which is written in the Salesforce Object Query Language (SOQL). Let's take a look at that query in detail.

- `SELECT`—Determines which fields are retrieved on the object.
- `FROM`—Determines which object or objects you want to access. The "j" in `Line_Item__c j` is an alias: it is a convenient shorthand that allows you to refer to `Line_Item__c` as "j".
- `WHERE`—This is the start of the condition statement. In this case you want to return only those records where the status is Negotiating.
- `AND`—This is the second condition of your statement. It pulls only the unique IDs of the records that are new. The code uses a special variable, `Trigger.new`, which is automatically initialized with the identifiers of the records being updated.
- `FOR UPDATE`—This tells the platform to lock the records, preventing other program or users from making updates to them. The lock remains until the trigger is complete.

Step 3: Iterate Through the List and Modify Price

In the previous step, you created a list of line items and stored it in a variable called `openLineItems`. Now you can iterate through the list using a `for` loop and modify an item's original price if its new price is lower.

1. Declare a `for` loop.

```
for (Line_Item__c li: openLineItems) {
}
```

2. Between the curly brackets, enter a conditional `if` statement.

```
for (Line_Item__c li: openLineItems) {
   if ( li.Merchandise__r.Price__c < li.Unit_Price__c ){
      }
}
```

3. Between the `for` loop's curly braces, enter code that updates the price.

```
for (Line_Item__c li: openLineItems) {
   if ( li.Merchandise__r.Price__c < li.Unit_Price__c ){
        li.Unit_Price__c = li.Merchandise__r.Price__c;
        }
}
```

4. The `for` loop is complete. Now you want to update the line items. Before the trigger's final curly brace, add the following.

```
update openLineItems;
```

5. During development you typically leave a trigger inactive. To try out the trigger, you'll need to activate it.

   - If you're using the Web interface, select the **Is Active** checkbox, which becomes available after saving the trigger.
   - If you're using the IDE, click the **Metadata** tab and change the status value to `Active`.
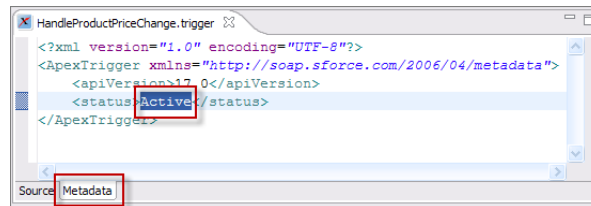
6. Verify that your code looks like the following and then **Save**.

```
trigger HandleProductPriceChange on Merchandise__c (after update) {

    List<Line_Item__c> openLineItems =
        [SELECT j.Unit_Price__c, j.Merchandise__r.Price__c
         FROM Line_Item__c j
         WHERE j.Invoice_Statement__r.Status__c = 'Negotiating'
         AND j.Merchandise__r.id IN :Trigger.new
         FOR UPDATE];

    for (Line_Item__c li: openLineItems) {
        if ( li.Merchandise__r.Price__c < li.Unit_Price__c ){
        li.Unit_Price__c = li.Merchandise__r.Price__c;
        }
    }
    update openLineItems;
}
```

Tell Me More....

The final statement, `update openLineItems`, updates the records in the database. That's simple enough, but what about that `for` loop?

- `for (Line_Item__c li: openLineItems) { }`—Iterates over the list of open line item. As you iterate, the current line item is assigned to the variable `li`.
- `if (li.Merchandise__r.Price__c < li.Unit_Price__c ) { }`—Check to see if the price on the merchandise record is lower than the current price. You only want to take an action if this is true.
- `li.Unit_Price__c = li.Merchandise__r.Price__c;`—Finally, update the unit price on the current line item (assigned to variable `li`) with the new merchandise price.

Step 4: Test the Trigger

Now let's test the trigger in the app. First you need to create a new invoice statement and order at least one product, or modify an existing one. The only requirement is that you change the status field of the invoice statement to Negotiating. Then you'll lower the unit price of a merchandise item used in one of the line items of that invoice, and verify that your line item and invoice values are updated.

1. Click the Invoice Statements tab.
2. Click the name of an existing invoice statement.
3. Change its `Status` to `Negotiating` and click **Save**.
4. Take note of the Total Value of the invoice. Now click on a Line Item, and note its Unit Price.
5. Click the Merchandise tab, and select the Wee Jet record (or any other item used in the line items).
6. Edit the record, lowering the unit price. If you want an extreme reduction, change it to `.01`.
7. Navigate back to the Invoice and the Line Item.
8. Note how the values of the line item and the invoice have automatically changed.

Tell Me More....

You may have noticed that this trigger is a little wasteful: it initializes and operates on a list of all line item records that belong to invoice statements under negotiation. However, just because a merchandise record was updated, that doesn't mean your trigger should update the record. For example, you may have updated the merchandise record's description, or increased its price, and neither of these cases requires the trigger to fire. Ideally this trigger should iterate over the `Trigger.new` set (the list of Merchandise records that have been updated), creating a new set of these records that pass the price criterion, and only then retrieve the related line item records.

Summary

In this tutorial, you built on the app by adding business logic that updates all open invoices when the unit price of a merchandise item goes down. To do this, you defined a trigger that executes whenever a merchandise record is saved with a lower unit price. Triggers are very useful in scenarios such as these—when you need to update multiple records on a particular condition. You can have more than one trigger associated with an object that can be triggered under different events and conditions.

An important part of creating Apex triggers is making sure you have automated tests that verify that your trigger works as it should. You'll learn about testing in

Tutorial #8: Adding Tests to Your App

Level: Advanced; **Duration:** 20-30 minutes
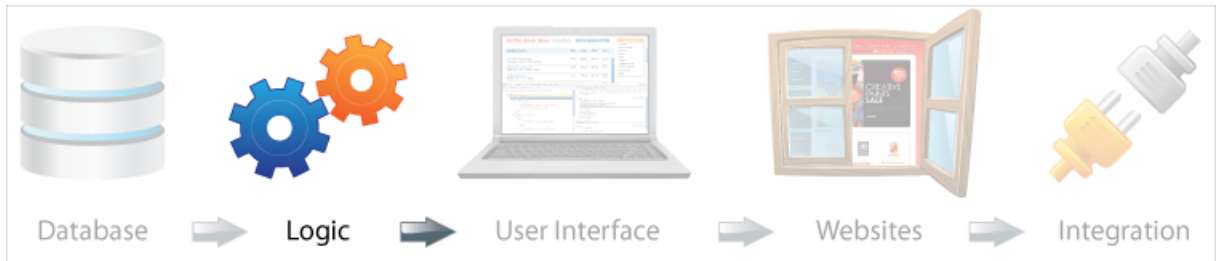
Testing is one of the most important steps in developing any app. Testing ensures that your code behaves as it should and doesn't consume resources unnecessarily. Not only does testing benefit you (you will have more confidence in the apps you write), but it also benefits the platform. To that end, the platform requires that you test any Apex you write before that code can be deployed to a production environment. Tests are also run before a new release of the platform is made available, to ensure that no backward compatibility problems are introduced.

To facilitate testing, Apex supports *unit tests*, which programmatically validate the code behavior and expected results. All Apex code has a set of limits that determine how many resources may be consumed. For example, there is a set limit on the number of queries that can be performed in a trigger. Good tests ensure that your code not only behaves as it should, but also that it doesn't exceed these limits. In this tutorial you'll write and execute a unit test, which exercises the Apex trigger you just created.

Prerequisites

Apex Tutorial

You first need to create an Apex trigger as described in Tutorial #7: Adding Programmatic Logic with Apex on page 48.

Software Requirements

You can use the Force.com IDE for this tutorial if you want to. In that case you'll need the Force.com IDE plugin: wiki.developerforce.com/index.php/Force.com_IDE. If you need help creating a project, see Creating a Project in the Force.com IDE.

Step 1: Create an Apex Test Class

All unit tests are contained in Apex classes. In this step you create the class to contain the unit tests.

To create a new Apex class in the Web interface:

1. Click **Your Name** > **Setup** > **Develop** > **Apex Classes**.
2. Click **New**.
3. In the editor pane, enter the following code.

```
@isTest
private class TestHandleProductPriceChange {
}
```

4. Click **Quick Save** to save your work and continue editing.

To create a new Apex class in the Force.com IDE:

1. In the IDE, right-click your project folder. Then click **New** > **Apex Class**.
2. On the Create Apex Class page, enter `TestHandleProductPriceChange` for the name.
3. In the `Template` field, choose `Test Class`.
4. Click **Finish** to create the class.

Tell Me More....

The `@isTest` annotation tells Force.com that all code within the Apex class is code that tests the rest of your code. You will now create test methods within this class that perform the actual tests.

Step 2: Add Test Methods to the Class

Now you can add a method to the class that will do the actual testing. The trigger that you created, which you want to test, only works when records are updated. So the first thing you need to do is create test records in the database. You want the method to create an Invoice Statement as well as Merchandise and Line Item records, and add these to the database.

1. Start by creating the test method that will contain your three test procedures. Between the curly braces, enter the following code.

    ```
    static testMethod void testPriceChange()  {

    }
    ```

    Note: If you're using the IDE, replace the generated comment and code.

2. Between the curly braces, add the following code, which creates an invoice statement and inserts it into the database.

    ```
    Invoice_Statement__c invoice = new Invoice_Statement__c(Status__c = 'Negotiating');
    insert invoice;
    ```

3. Directly below `insert invoice`, add the code to create new merchandise records.

    ```
    Merchandise__c[] products = new Merchandise__c[]{
     new Merchandise__c(Name = 'item 1', Description__c = 'test product 1', Price__c = 10,
    Total_Inventory__c = 10),
     new Merchandise__c(Name = 'item 2', Description__c = 'test product 2', Price__c = 11,
    Total_Inventory__c = 10)
    };
    insert products;
    ```

4. Then, directly below `insert products`, enter the code to add line items.

    ```
    Line_Item__c[] lineItems = new Line_Item__c[] {
     new Line_Item__c(Invoice_Statement__c = invoice.id, Merchandise__c = products[0].id,
    Unit_Price__c = 10, Units_Sold__c = 3),
     new Line_Item__c(Invoice_Statement__c = invoice.id, Merchandise__c = products[1].id,
    Unit_Price__c = 11, Units_Sold__c = 6)
    ```

```
};
insert lineItems;
```

Tell Me More....

You have now set up the data. By default, Apex test methods don't have access to pre-existing data in the organization, such as the records you created in previous tutorials. In this test method, you created your own test data. You don't have to delete this test data because it isn't committed to the database and is rolled back when the test execution completes. When the test execution finishes, the test data won't be persisted in the organization and won't be available.

Note how the invoice statement that you created has a status set to Negotiating, which is the condition you need for the trigger to fire. You now need to write the code that will fulfill the other conditions for the trigger to fire: you need to lower the price of a piece of merchandise. You'll do that in the next step.

Step 3: Write Code to Execute the Trigger

The code in your test method doesn't really do much yet: it won't cause your trigger to execute, as that only happens when you update the price of a Merchandise record.

1. After `insert lineItems;`, add the following lines.

```
products[0].price__c = 20;
Test.startTest();
update products;
Test.stopTest();
```

> **Note:** This code modifies the price of one of the Merchandise records, raising its price. It then calls `startTest()`, which is part of the testing framework that marks the point in your test code when your test actually begins. The system will not consider the setup code and database operations that precede it to be part of the test. Testing also ensures that your code doesn't exceed certain governor limits—which limits the resources your Apex code may use. By marking where your actual test begins and ends, the system won't count your setup code towards the governor limits, ensuring that your test is a more accurate reflection of how it would behave in a production environment.

2. Now that you have updated the products, you need to check whether the price in the line items has changed. Ideally, it shouldn't have, because the price was raised in the setup code. Add the following code after `Test.stopTest();`

```
lineItems = [SELECT id, unit_price__c FROM Line_Item__c WHERE id IN :lineItems];
system.assert(lineItems[0].unit_price__c == 10);
```

3. Verify that your code looks like the following and then **Save**.

```
@isTest
private class TestHandleProductPriceChange {

    static testMethod void testPriceChange() {
        Invoice_Statement__c invoice = new Invoice_Statement__c
            (Status__c = 'Negotiating');
        insert invoice;

        Merchandise__c[] products = new Merchandise__c[]{
            new Merchandise__c(Name = 'item 1', Description__c =
```

```
            'test product 1', Price__c = 10, Total_Inventory__c = 10),
                new Merchandise__c(Name = 'item 2', Description__c =
            'test product 2', Price__c = 11, Total_Inventory__c = 10)
            };
            insert products;
            Line_Item__c[] lineItems = new Line_Item__c[] {
                new Line_Item__c(Invoice_Statement__c = invoice.id,
            Merchandise__c = products[0].id, Unit_Price__c = 10, Units_Sold__c = 3),
                new Line_Item__c(Invoice_Statement__c = invoice.id,
            Merchandise__c = products[1].id, Unit_Price__c = 11, Units_Sold__c = 6)
            };
            insert lineItems;

            products[0].price__c = 20;
            Test.startTest();
            update products;
            Test.stopTest();

          lineItems = [SELECT id, unit_price__c FROM Line_Item__c WHERE id IN :lineItems];

            system.assert(lineItems[0].unit_price__c == 10);
        }
    }
```

Step 4: Execute the Test

Force.com has a testing framework that lets you execute tests, and also check code coverage. You are now going to execute the tests, and look at the resulting code coverage.

Note: These steps show how to execute steps in the Web interface, but you can also execute tests in the Force.com IDE by right-clicking the class name in the Package Explorer and choosing **Force.com** > **Run Tests**. The remainder of this tutorial uses the Web interface, but you should be able to follow along easily in the IDE.

1. Navigate to your test class by clicking `Your Name` > **Setup** > **Develop** > **Apex Classes**, and then click your test class, `TestHandleProductsPriceChange`.
2. Click **Run Test** and you'll see output similar to the following.

This result tells you a number of important things.

- It indicates whether your tests were successful or not. If the boolean condition in the `system.assert` statement that you added to the test had failed, that result would be flagged here. Adding lots of asserts is a great way for you to test the expected behavior of your code.
- It provides detail about the execution of the test. By looking through the Debug Log, for example, you can see that your `Order_in_Stock` validation rule has fired. And you can see which records you created, how many queries you executed, and more.
- It indicates code coverage: how many lines of code you executed in some other class or trigger.

Note also how the result page shows that you provided 80% coverage of the `HandleProductPriceChange` trigger. That's enough code coverage to deploy, but we strive for perfection. In the next step, you'll view the code coverage to see where you need to add more test cases to bring it up to 100%.

Step 5: View Code Coverage and Improve Tests

You have written two sets of code. You have your trigger, which we'll call *production code*, and you have your code in the test class, which we'll call *test code*. The term *code coverage* refers to how much of your production code is covered by testing code. In other words: when someone runs your testing code, does it execute all, or only some portion of, the production code? If it only executes a portion of the code, that could mean your production code contain bugs in the untested code. You can make these concepts a little more concrete by looking at a graphical code coverage chart.

Note: If you have been using the Force.com IDE for this tutorial, keep using it. Developing in both the IDE and the Web interface at the same time requires that you synchronize with the server every time you switch back and forth. Synchronizing is easy (right-click your class and choose **Force.com** > **Synchronize with Server**), but it's even easier to avoid!

1. In the Code Coverage section, click **80**.

2. The Code Coverage page opens. The blue highlight marks lines of code that *have* been covered (executed) as a result of our test method. The red highlight marks lines that *have not* been executed. In this case, line 13 wasn't executed because we don't lower the price of a merchandise item, we only raise it.

3. That red line is a good indicator that your test isn't complete. So let's modify your test method to improve your code coverage. Close the Code Coverage window and then click **Your Name** > **Setup** > **Develop** > **Apex Classes** and select your test class **TestHandleProductPriceChange**.

4. Click **Edit**.

5. Starting with `products[0]`, replace the code up to the next curly brace with the following.

```
products[0].price__c = 20; // raise price
products[1].price__c = 5;  // lower price
Test.startTest();
update products;
Test.stopTest();

lineItems =
[SELECT id, unit_price__c FROM Line_Item__c WHERE id IN :lineItems];
System.assert(lineItems[0].unit_price__c == 10);  // unchanged
System.assert(lineItems[1].unit_price__c == 5);   // changed!!
```

6. Click **Save**.

7. Now click **Run Test** again, and you'll see that you have 100% code coverage.

Summary

In this tutorial you created tests for the Apex trigger and saw how the integrated testing tools can help you attain 100% code coverage. Creating unit tests as you develop is necessary for deployment and it's also the key to successful long-term development.

But it's important to note that code coverage isn't the only goal for testing; ideally you should also check all possible scenarios, and the test case you have just written misses one. Right now it only checks invoice statements that have a status of Negotiating. It should also check other status values. For example, if an invoice statement *does not* have a status of Negotiating, then raising or lowering the price shouldn't have any effect.

Tutorial #9: Building a Custom User Interface Using Visualforce

Level: Intermediate; **Duration:** 30–45 minutes

Visualforce is a component-based user interface framework for the Force.com platform. In previous tutorials you built and extended your app by using a user interface that is automatically generated. Visualforce gives you a lot more control over the user interface by providing a view framework that includes a tag-based markup language similar to HTML, a library of reusable components that can be extended, and an Apex-based controller model. Visualforce supports the Model-View-Controller (MVC) style of user interface design, and is highly flexible.

In this tutorial, you will use Visualforce to create a new interface for the Warehouse app that displays an inventory count sheet that lets you list all Merchandise inventory, as well as update the counts on each. The purpose of the count sheet is to update the computer system with a physical count of the merchandise, in case they are different.

Prerequisites

Basic Knowledge

For this tutorial, it helps to have basic knowledge of markup languages like HTML and XML, but it is not required.

Step 1: Enable Visualforce Development Mode

Development Mode embeds a Visualforce page editor in your browser. It allows you to see code and preview the page at the same time. Development Mode also adds an Apex editor for editing controllers and extensions.

1. Click *Your Name* > **Setup** > **My Personal Information** > **Personal Information**.
2. Click **Edit**.
3. Select the `Development Mode` checkbox and click **Save**.

Step 2: Create a Visualforce Page

In this step you create a Visualforce page that will serve as an inventory count sheet.

1. In your browser, add the text `/apex/CountSheet` to the URL for your Salesforce instance. For example, if your Salesforce instance is `https://na1.salesforce.com`, the new URL would be `https://na1.salesforce.com/apex/CountSheet`. You will get an error message: Page CountSheet does not exist.
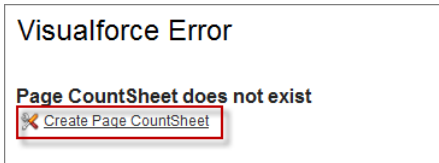
2. Click the **Create Page CountSheet** link to create the new page.
3. Click the Page Editor link in the bottom left corner of the page. The Page Editor tab displays the code and a preview of the new page (which has some default text). It should look like this.
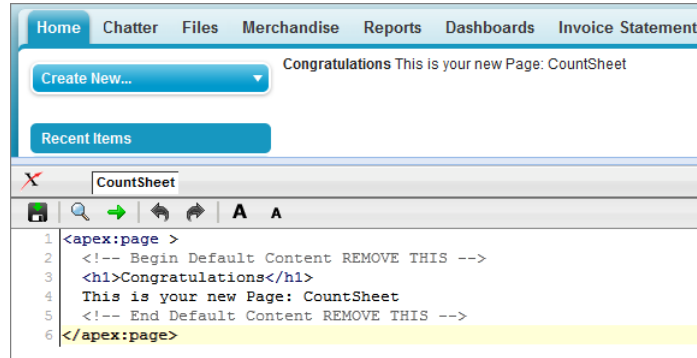
4. You don't really want the heading of the page to say "Congratulations", so change the contents of the `<h1>` tag to `Inventory Count Sheet`, and go ahead and remove the comments. The code for the page should now look like this.

```
<apex:page>

<h1>Inventory Count Sheet</h1>

</apex:page>
```

5. Click the **Save** icon at the top of the Page Editor. The page reloads to reflect your changes.

Tell Me More....

Notice that the code for the page looks a lot like standard HTML. That's because a Visualforce page combines HTML tags, such as `<h1>`, with Visualforce-specific tags, which start with `<apex:>`

Step 3: Add a Stylesheet Static Resource

You want your Warehouse app to look slick, so you're going to use a custom stylesheet (CSS file) to specify the color, font, and arrangement of text on your page. Most Web pages and Web designers use CSS, a standard Web technology, for this purpose, so we've created one for you. In order for your pages to reference a stylesheet, you have to upload it as a *static resource*. A static resource is a file or collection of files that is stored on Force.com. Once your stylesheet is added as a static resource, it can be referenced by any of your Visualforce pages.

To add a style sheet as a static resource:

1. In your browser, go to developer.force.com/workbook/styles. Download the file and save it to your desktop.
2. Back in the app, click *Your Name* > **Setup** > **Develop** > **Static Resources** and click **New**.
3. In the `Name` field, enter `styles`.
4. Click **Choose File** and find the `styles.zip` file you downloaded.
5. In the `Cache Control` picklist choose `Public`.

6. Click **Save**.

> **Note:** If your Visualforce pages are exposed on a public website, then Force.com uses a global content delivery network of cache servers to hold copies of your static files.

Now you need to modify your Visualforce page to reference the stylesheet.

1. Just as you did when you created the page, add the text `/apex/CountSheet` to the URL for your Salesforce instance.
2. Modify the attributes of the `<apex:page>` tag and enter the following code to remove the standard stylesheet, the header, and the sidebar.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false">
```

3. Now you need to tell the page where to find the stylesheet, so add a new line below the first `<apex:page>` tag and type `<apex:`
4. The editor has code insight, which gives you a drop-down list of the elements that are available in this context. Start typing `stylesheet` and when you see `apex:stylesheet`, select it.
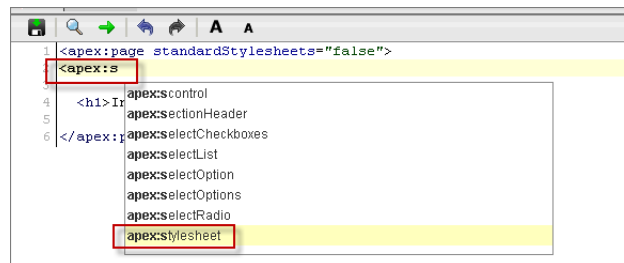
5. Now specify the location of the stylesheet as shown.

```
<apex:stylesheet value="{!URLFOR($Resource.styles, 'styles.css')}" />
```

6. Verify that your code looks like the following:

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false">

<apex:stylesheet value="{!URLFOR($Resource.styles, 'styles.css')}" />

<h1>Inventory Count Sheet</h1>

</apex:page>
```

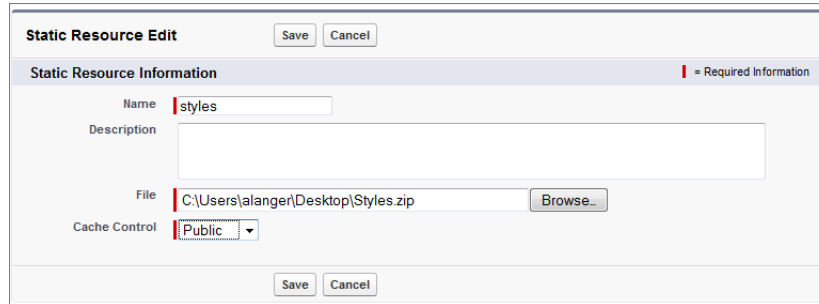7. Click the **Save** icon at the top of the Page Editor.

Note how the page now looks very different, the title is in a different font and location, and the standard header and sidebar are no longer present.

Tell Me More....

Let's take a look at that stylesheet code in a little more detail.

- `$Resources` is a global variable which Visualforce has access to. With `$Resource.styles`, you refer to the resource called "styles" which you created earlier.
- The `URLFOR()` function locates the static resource, and a file within that resource, and calculates the URL that should be generated in your final page. If the syntax looks familiar, it's because you've already encountered it to dynamically evaluate values when the Visualforce page is rendered.

Step 4: Add a Controller to the Page

Visualforce's Model-View-Controller design pattern makes it easy to separate the view and its styling from the underlying database and logic. In MVC, the view (the Visualforce page) interacts with a controller. In our case, the controller is usually an Apex class, which exposes some functionality to the page. For example, the controller may contain the logic that should be executed when a button is clicked. A controller also typically interacts with the model (the database)—exposing data that the view might want to display.

All Force.com objects have default standard controllers that can be used to interact with the data associated with the object, so in many cases you don't need to write the code for the controller yourself. You can extend the standard controllers to add new functionality, or create custom controllers from scratch. In this tutorial you'll use the default controller.

1. If the Page Editor isn't open on your Visualforce page, click **Page Editor** to edit the page.
2. Modify your code to enable the Merchandise__c standard controller by editing the first `<apex:page>` tag. The editor ignores whitespace, so you can enter the text on a new line.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false"
standardController="Merchandise__c">
```

3. Next, add the standard list controller definition.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false"
standardController="Merchandise__c" recordSetVar="products">
```

4. Click the **Save** icon at the top of the Page Editor. You won't notice any change on the page. However, because you've indicated that the page should use a controller, and defined the variable `products`, the variable will be available to you in the body of the page and it will represent a list of Merchandise records.

Tell Me More....

The `recordSetVar` attribute enables a standard list controller, which provides additional controller support for listing a number of records with pagination. Its value, which you've set to products, will ensure that a new variable, `products`, will contain the set of records to be displayed on the Visualforce page.

Step 5: Display the Inventory Count Sheet as a Visualforce Page

You now have all the functionality in place to flesh out the Visualforce page. It will display a table of all the merchandise records, together with an input field on each so that you can update the inventory count.

1. In the line below the `</h1>` tag, start typing `<apex:f` on a new line, and highlight `<apex:form>` when it appears in the drop-down list. The form will allow you to make updates to the page.
2. Press Enter and notice that the system generates the opening and closing tags for you.
3. Place your cursor between the tags and create a data table. Start by typing `<apex:d` and press Enter to select `dataTable` from the drop-down list.
4. Now you need to add some attributes to the `dataTable` tag. The value attribute indicates which list of items the `dataTable` component should iterate over. The `var` attribute assigns each item of that list, for one single iteration, to the `pitem` variable. The `rowClasses` attribute assigns CSS styling names to alternate rows. On one or more lines within the tag, enter the following.

```
<apex:dataTable value="{!products}" var="pitem" rowClasses="odd,even">
```

5. Now you are going to define each column, and determine where it gets its data by looking up the appropriate field in the *pitem* variable. Add the following code between the opening and closing `dataTable` tags.

```
<apex:dataTable value="{!products}" var="pitem" rowClasses="odd,even">
    <apex:column headerValue="Product">
       <apex:outputText value="{!pitem.name}"/>
    </apex:column>
</apex:dataTable>
```

6. Click **Save**, and you will see your table appear.

The `headerValue` attribute has simply provided a header title for the column, and below it you'll see a list of rows: one for each merchandise record. The expression `{!pitem.name}` indicates that we want to display the name field of the current row.

7. Now, after the closing tag for the first column, add two more columns.

```
<apex:column headerValue="Inventory">
   <apex:outputField value="{!pitem.Total_Inventory__c}"/>
</apex:column>
<apex:column headerValue="Physical Count">
   <apex:inputField value="{!pitem.Total_Inventory__c}"/>
   </apex:column>
```

Note: The second column is an `inputField`, not an `outputField`. The `inputField` will display a value, but it will also allow you to change it.

8. Click **Save** and you have an inventory count sheet! It lists all the merchandise records, displays the current inventory, and provides an input field for the physical count.

9. As a final embellishment, add a button that will modify the physical count on any row and refresh the values on the page. To do this, enter the following code directly above the `</apex:form>` line.

```
<br/>
<apex:commandButton action="{!quicksave}"  value="Update Counts" />
```


Tell Me More....

*   The `dataTable` component produces a table with rows, and each row is found by iterating over a list. The standard controller you used for this page was set to `Merchandise__c`, and the `recordSetVar` to `products`. As a result, the controller automatically populated the products list variable with merchandise records retrieved from the database. It's this list that the `dataTable` component uses.
*   You need a way to reference the current row as you iterate over the list. That statement `var="pitem"` assigns a variable called `pitem` that holds the current row.
*   The `rowClasses` and `styleClass` attributes simply use some of the styles from the CSS style sheet that you loaded in the static resource. You can safely remove this—it will just make things look less pretty!
*   Every standard controller has various methods that exist for all Force.com objects. The `commandButton` component displays the button, and invokes a method called `quicksave` on the standard controller, which updates the values on the records. Here, you're updating the physical count of the product and performing a quick save, which updates the product with the new count.
*   Although pagination isn't shown in this example, the functionality is there. If you have enough records to page through them, add the following code below the `commandButton` for page-flipping action.

```
<apex:commandLink action="{!next}" value="Next" rendered="{!hasNext}" />
```

Step 6: Add Inline Editing Support

You now have a Visualforce page that contains a table that displays all the merchandise records and allows you to edit the inventory count through the physical count input field. In this step, you modify this table to add inline editing support for the inventory output field. Also, since inline editing makes the physical count input field unnecessary, you remove the last column, which contains this field. After carrying out this step, you will be able to edit the inventory count by double-clicking a field in the **Inventory** column.

Instead of using an `inputField` to edit the physical count in the last column in the previous step, you can make the inventory column editable by adding an `inlineEditSupport` component as a child component of the `outputField` component. The following procedure shows how to do this.

1. Delete the following markup for the physical count column.

```
<apex:column headerValue="Physical Count">
    <apex:inputField value="{!pitem.Total_Inventory__c}"/>
</apex:column>
```

2. Within the inventory column, break up the `outputField` component so that it has an end tag, as follows.

```
<apex:outputField value="{!pitem.Total_Inventory__c}">

</apex:outputField>
```

3. Between the `outputField`'s start and end tag, insert the `inlineEditSupport` component.

```
<apex:inlineEditSupport event="ondblclick" showOnEdit="update"/>
```

4. Now that you've added the `inlineEditSupport` component, modify the update button to give it an ID and a style class name. The ID is referenced by the `inlineEditSupport` component to show the button during editing. The style class name is used in `styles.css` to hide the update button the first time the page renders. Replace the `commandButton` with the following.

```
<apex:commandButton id="update" action="{!quicksave}" value="Update Counts"
  styleclass="updateButton"/>
```

5. Your Visualforce markup should look like the following.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false"
  standardController="Merchandise__c" recordsetVar="products">
  <apex:stylesheet value="{!URLFOR($Resource.styles, 'styles.css')}"/>
  <h1>Inventory Count Sheet</h1>
  <apex:form>
     <apex:dataTable value="{!products}" var="pitem" rowClasses="odd,even">
        <apex:column headerValue="Product">
           <apex:outputText value="{!pitem.name}"/>
        </apex:column>
        <apex:column headerValue="Inventory">
           <apex:outputField value="{!pitem.Total_Inventory__c}">
              <apex:inlineEditSupport event="ondblclick" showOnEdit="update"/>
           </apex:outputField>
        </apex:column>
     </apex:dataTable>
     <br/>
     <apex:commandButton id="update" action="{!quicksave}" value="Update Counts"
        styleclass="updateButton"/>
  </apex:form>
</apex:page>
```

6. Click **Save**. The page now displays the inventory count table with two columns. Notice that the **Update Counts** button is hidden initially.
7. Double-click any cell in the inventory column. Notice how the field dynamically becomes an input field and the **Update Counts** button appears. Also, an undo button dynamically appears to the right of the field which allows you to cancel out of this edit operation.

8. Modify the count value and click **Update Counts** to commit this update.

Tell Me More....

- The `event` attribute of the `inlineEditSupport` component is set to "ondblclick", which is a DOM event and means that the output field will be made editable when you double-click it. Also, the `showOnEdit` attribute causes the **Update Counts** button to appear on the page during an inline edit. This attribute is set to the ID of the **Update Counts** button.
- The **Update Counts** button is hidden through its style specification in the static resource file `styles.css`. The `styleclass` attribute on `commandButton` links this button to an entry in `styles.css`.

Summary

Congratulations! You have created a new interface for your Warehouse app by creating a Visualforce page that uses a standard controller. Your page is highly configurable. For example, you can easily modify which data is displayed in each row by modifying the column components. The page also makes use of a lot of functionality provided by the standard controller behind the scenes. For example, the controller automatically queries the database and finds all merchandise records and assigns them to the *products* variable. It also provides a way of saving records, through its quicksave method.

Now that you have created some Visualforce components, you can expose this functionality on a public website. You'll find out about that in the Tutorial #10: Creating a Public Web Page Using Sites tutorial.

Tutorial #10: Creating a Public Web Page Using Sites

Level: Intermediate; **Duration:** 20-30 minutes

The app you just created requires that you log in to use it. While Force.com supports your app in a multi-user environment, sometimes you need to eliminate the login—for example on a public website.

Sites enables you to create public websites and apps that are directly integrated with your organization—without requiring users to log in with a username and password. You can publicly expose any information stored in your organization through a branded URL of your choice. You can also make the site's pages match the look and feel of your company's brand. Because sites are hosted on Force.com servers, there are no data integration issues. And because sites are built on native pages, data validation on collected information is performed automatically. You can also enable users to register for, or log into, an associated portal seamlessly from your public site.

In this tutorial, you will create a Visualforce page, enable Sites for your organization, register your Force.com domain name, and expose the Visualforce page you created as a public product catalog on the Web.

Prerequisites

Visualforce Tutorial

You first need to create the Visualforce page, as described in Tutorial #9: Building a Custom User Interface Using Visualforce on page 60.

Step 1: Create a Product Catalog Page

In this step, you clone the inventory count page you created earlier. The new Product Catalog page will show description and price instead of inventory and count.

1. Click *Your Name* > **Setup** > **Develop** > **Pages**. If you're still in the Visualforce editor, click **Back** in your browser until you see the Setup area.
2. Select the **CountSheet** page you created earlier.
3. Click **Clone**.
4. On the Page Editor tab, change the `Label` and `Name` fields to `Catalog`.
5. In the Page Editor, change the contents of the `<h1>` tag to `Merchandise Catalog`.
6. In the editor, find the following lines.

```
<apex:column headerValue="Inventory">
    <apex:outputField value="{!pitem.Total_Inventory__c}"/>
</apex:column>
```

7. Change `Inventory` to `Description` and change `Total_Inventory__c` to `Description__c`. You're keeping the same table, but changing the table heading and the data within. The code should look like the following.

```
<apex:column headerValue="Description">
   <apex:outputField value="{!pitem.Description__c}"/>
</apex:column>
```

8. Now you'll make a similar change on the next column to display the price. Change `Physical Count` to `Price`. Then change `inputfield` to `outputfield`. Then change `Total_Inventory__c` to `Price __c`, as shown.

```
<apex:column headerValue="Price">
   <apex:outputField value="{!pitem.Price__c}"/>
</apex:column>
```

9. Finally, remove the opening and closing `<apex:form>` and `<apex:commandButton>` tags, because you don't want to accept input on this page.

10. Verify that your code looks like the following and then click **Save**.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false"
           standardController="Merchandise__c" recordSetVar="products" >

  <apex:stylesheet value="{!URLFOR($Resource.styles, 'styles.css')}"/>

  <h1>Merchandise Catalog</h1>

  <apex:dataTable value="{!products}" var="pitem" rowClasses="odd,even">
    <apex:column headerValue="Product">
      <apex:outputText value="{!pitem.name}"/>
    </apex:column>
    <apex:column headerValue="Description">
      <apex:outputField value="{!pitem.Description__c}"/>
    </apex:column>
    <apex:column headerValue="Price">
      <apex:outputField value="{!pitem.Price__c}"/>
    </apex:column>
  </apex:dataTable>

</apex:page>
```

Tell Me More....

There are a few important things to note here.

• Lots of components on the Force.com platform can be cloned, and as you just saw, it's easy to clone a Visualforce page.
• The standard controllers for the Visualforce page make all your data readily accessible. By simply changing a couple of values in a table, you can display data from different fields.
• Did you notice how easy it was to change an input field to an output field? The Visualforce standard controller is powerful indeed, but you can also extend the controller to create custom functionality, and even create your own controller from scratch.

Step 2: Register a Force.com Domain Name

Your unique Force.com domain, which hosts your site, is constructed from the unique *domain prefix* that you register, plus `force.com`. For example, if you choose "mycompany" as your domain prefix, your domain name would be `http://www.mycompany.force.com`.

To get started, register your company's Force.com domain by doing the following.

1. Click *Your Name* > **Setup** > **Develop** > **Sites**.
2. Enter a unique name for your Force.com domain. A Force.com domain name can contain only alphanumeric characters and hyphens, and must be unique in your organization. It must begin with a letter, not include spaces, not end in a hyphen, and not contain underscores or periods. Salesforce.com recommends using your company's name or a variation, such as `mycompany`.

   > **Warning:** You can't modify your Force.com domain name after you have registered it.

3. Click **Check Availability** to confirm that the domain name you entered is unique. If it isn't unique, you are prompted to change it.
4. Read and accept the Sites Terms of Use by selecting the checkbox.
5. Click **Register My Force.com Domain**. After you accept the Terms of Use and register your Force.com domain, the changes related to site creation are tracked in your organization's Setup Audit Trail and the Site History related list. It may take up to 48 hours for your registration to take effect.

Congratulations! You are now ready to create your first Force.com site.

Step 3: Create a Force.com Site

Now that you have registered your domain, you can make the Merchandise Catalog Visualforce page you created the home page for your new site.

1. Go to the Sites page by clicking *Your Name* > **Setup** > **Develop** > **Sites**.
2. Click **New**. The Site Edit page appears.
3. On the Site Edit page, fill in the site details:

   a. In the `Site Label` and `Site Name` fields enter `Catalog`.
   b. In the `Active Site Home Page` field enter `Catalog`.
   c. Select the `Active` checkbox.

4. Click **Save**.

Step 4: Configure and Test the Site

Now that you've created your site, and established a default page, you're almost ready to try it out.

The platform has a number of conservative controls in place for data security. One of the controls ensures that data isn't displayed, even on public pages, unless you explicitly enable it. In this step you will toggle this security setting for the Merchandise object, and then visit your new site.

1. Go to the Sites list by clicking `Your Name` > **Setup** > **Develop** > **Sites**.
2. Click the `Site URL` link for the Product Catalog site. This action opens either a new tab or a new window (depending on your browser). However, instead of seeing your Merchandise Catalog page, you'll see a large Authorization Required page. This is because the anonymous website viewer hasn't yet been granted permission to view the data that the page is displaying. Let's fix this. Go back to the setup page.
3. Click the `Site Label` link, it should be called Catalog.

4. Click **Public Access Settings** and then in the Profile Detail section, click **Edit**.

5. Scroll down to the Custom Object Permissions section and select the `Read` permission for the Merchandise object.

6. Click **Save**.

7. Now go back to your website and refresh your browser and you should see your page.

Note: If you still have an authorization required message, you probably didn't set your `Active Site Home Page` to `Catalog` in the previous step.

Tell Me More....

- You've assigned a single Visualforce page to the site, and made it the home page for that site. You can go ahead and create additional Visualforce pages and assign them to the site as well. For example, if you add a page Meme to the site, you will be able to access it with a URL something like `http://workbook-developer-edition.na1.force.com/Meme` (depending on your domain name of course).
- Force.com keeps track of the number of page views a Developer Edition site and imposes a daily bandwidth limit on the site.
- Sites lets you make use of a global content delivery network for fast access and caching of your site pages on production environments. You simply need to modify the `page` component to include a cache instruction.

```
<apex:page cache="true" expires="600">
```

Summary

Congratulations, you have created a public website! Anyone in the world can access the site you just created and look at your home page, all without having to log in. To create the site, you cloned a Visualforce page and then enabled your Developer Edition organization for Sites functionality. Then you registered a unique Force.com domain name, registered the page as your home page, tweaked the security, and tested the site to see that it works.

Now that you have a site with a product catalog, the next logical step is to provide a way for people to order merchandise. You'll do that next in the tutorial Tutorial #11: Creating a Store Front on page 74.

Tutorial #11: Creating a Store Front

Level: Intermediate; **Duration:** 30 minutes
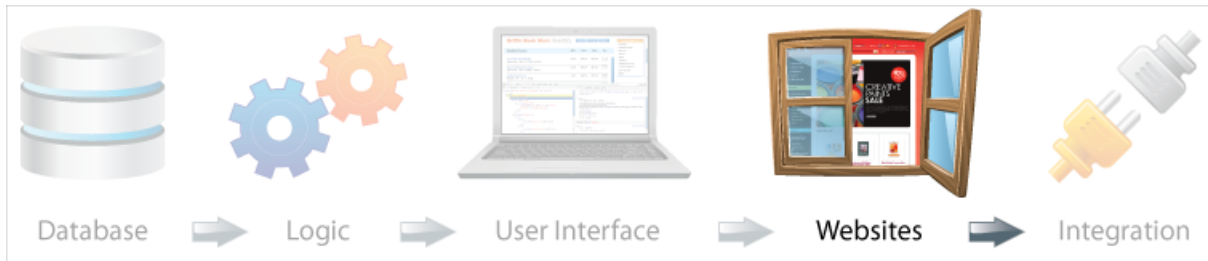
Apex is an object-oriented programming language. Using Apex, you can create classes and methods, make calls to the database, create Web services, send email, and much more. In this tutorial you are going to create a simple Visualforce store front page, which will use a controller you write in Apex. Along the way, you'll create an Apex class, discover more Apex syntax, and learn how to pass values between a Visualforce page and its controller.

Prerequisites

Warehouse App

You first need to create the basic Warehouse app and add relationships, as described in Tutorial #2: Adding Relationships on page 10.

Software Requirements

You can use the Force.com IDE for this tutorial if you want to. In that case you'll need the Force.com IDE plugin: wiki.developerforce.com/index.php/Force.com_IDE. If you need help creating a project, see Creating a Project in the Force.com IDE.

"Developer Mode," "Modify All Data," and "Author Apex" permissions

Since this tutorial involves working with Apex, make sure that you have the proper permissions to create Apex classes.

Step 1: Create a Controller

Instead of using the default controller, as you did in the previous tutorial, you're now going to write the controller code yourself. Controllers typically retrieve the data to be displayed in a Visualforce page, and contain code that will be executed in response to page actions, such as a command button being clicked.

Your rudimentary store front will have two methods: `getProducts()`, which returns the products your store front will display; and `shop()`, which executes when products have been selected.

To create the class in the Web interface:

1. Click *Your Name* > **Setup** > **Develop** > **Apex Classes**.
2. Click **New**.
3. Add the following code as the definition of the class and then click **Quick Save**.

```
public class StoreFront {

}
```

To create the class in the Force.com IDE:

1. In the IDE, right-click your project folder and select **New** > **Apex Class**.
2. On the Create Apex Class page, specify `StoreFront` for the name.
3. Click **Finish**.

You now have a rudimentary class for your controller. It has no methods or fields, so you'll add those in the next step.

Step 2: Add Methods to the Controller

In the last step you created a custom controller. Now you'll add methods to it. You'll start by adding the `shop()` method. It's not going to do anything yet, but you'll need it as a placeholder for the action that occurs when someone clicks the Shop button. You also want the shop to display merchandise items, together with a count, so that the end user can select how many of each item they wish to purchase.

1. Add the following code to the class (just after the `public class StoreFront {` line).

```
public PageReference shop() {
    return null;
}
```

2. Within the current class (on the next line), define a field to hold a list of `DisplayMerchandise`, and a new inner class called `DisplayMerchandise` to hold the data.

```
DisplayMerchandise[] products;
public class DisplayMerchandise {
    public Merchandise__c merchandise { get; set; }
    public Decimal count { get; set; }
    public DisplayMerchandise(Merchandise__c item) {
        this.merchandise = item;
    }
}
```

3. On the next line, define the method `getProducts()`, which is going to initialize the products.

```
public DisplayMerchandise[] getProducts() {
    if (products == null) {
        products = new DisplayMerchandise[]{};
        for (Merchandise__c item :
    [SELECT id, name, description__c, price__c
     FROM Merchandise__c
     WHERE Total_Inventory__c > 0]) {
       products.add(new DisplayMerchandise(item));
        }
    }
    return products;
}
```

4. Verify that your code looks like the following and then click **Save**.

```
public class StoreFront {
    public PageReference shop() {
        return null;
    }
```

```
    DisplayMerchandise[] products;

    public class DisplayMerchandise {
        public Merchandise__c merchandise { get; set; }
        public Decimal count { get; set; }
        public DisplayMerchandise(Merchandise__c item) {
            this.merchandise = item;
        }
    }

    public DisplayMerchandise[] getProducts() {
        if (products == null) {
            products = new DisplayMerchandise[]{};
            for (Merchandise__c item :
                [SELECT id, name, description__c, price__c
                 FROM Merchandise__c
                 WHERE Total_Inventory__c > 0]) {
              products.add(new DisplayMerchandise(item));
            }
        }
     return products;
    }
}
```

You have now finished defining a simple controller. It uses standard Apex classes and methods to access the database.

Tell Me More....

The `DisplayMerchandise` class "wraps" the Merchandise type that you already have in the database, adding a new decimal field. The constructor lets you create a new `DisplayMerchandise` instance by passing in an existing Merchandise record. The instance variable *products* is defined as a list of `DisplayMerchandise` instances.

The `getProducts()` method executes a query (the text within square brackets, also called a SOQL query) returning all Merchandise records that have a positive total inventory. It then iterates over the records returned by the query, adding them to a list of `DisplayMerchandise` products, which is then returned.

Step 3: Create the Store Front

Your store front uses another Visualforce page for the store front, so let's create that now.

1. In the address bar of your browser, add `/apex/StoreFront` to the end of your instance. For example: `https://na1.salesforcecom/apex/StoreFront`. You'll see the familiar error message.
2. Click the **Create Page StoreFront** link.
3. Click the Page Editor tab at the bottom.
4. Replace the text in the editor with the following.

```
<apex:page standardStylesheets="false" showHeader="false" sidebar="false"
        controller="StoreFront" >
    <apex:stylesheet value="{!URLFOR($Resource.styles, 'styles.css')}"/>
    <h1>Store Front</h1>
    <apex:form>

    </apex:form>
</apex:page>
```

Note: The `controller="StoreFront"` attribute ensures that the Visualforce page you are creating will use the `StoreFront` Apex class as its controller.

5. You are now ready to insert the contents of the form between the opening and closing form tags. You will create a table that iterates over the products, and a button to call the `shop` method. Enter this code on the next line, between the open and close `form` tags.

```
<apex:dataTable value="{!products}" var="pitem" rowClasses="odd,even">
        <apex:column headerValue="Product">
            <apex:outputText value="{!pitem.merchandise.name}"/>
        </apex:column>
        <apex:column headerValue="Price">
            <apex:outputText value="{!pitem.merchandise.price__c}"/>
        </apex:column>
        <apex:column headerValue="#Items">
            <apex:inputText value="{!pitem.count}"/>
        </apex:column>
</apex:dataTable>
<br />
<apex:commandButton action="{!shop}" value="Buy" />
```

6. Click **Save**. Your Store Front page should look like this.

Tell Me More....

There's a lot of new stuff here. Let's take a look.

- The value attribute of the `dataTable` is set to "products", indicating that the table should iterate over a list called products. Because you are using a custom controller, Visualforce automatically looks for a method called `getProducts()` in your Apex controller.
- The `getProducts()` method in your controller returns an array of `DisplayMerchandise` objects. Each of these will form a new row, and is assigned to the `pitem` variable in the Visualforce page as the `dataTable` iterates.

- The Visualforce page has an action, `{!shop}`. Because this is an action, a method with the exact same name is used in the Apex controller.

Step 4: Bonus Step—Updating the Page with AJAX

This step is optional. It shows you how Visualforce transparently passes data back to your controller, where it can be processed. For example, the `shop()` method that you wrote in the controller has access to the merchandise counts that were entered by the end-user on the Visualforce page. This step shows you how to access that data in the controller. In this step, you will simply display that data in the page again (using a nifty AJAX page update). In a real-world scenario you could imagine emailing the order, invoking a Web service, adding the items to a shopping basket, and so on.

1. Navigate to your Visualforce page by using its URL, for example `https://na1.salesforce.com/apex/StoreFront`.
2. Click the Page Editor tab.
3. Modify your page by adding the following code after the `</apex:form>` line.

```
<apex:outputPanel id="msg">{!message}</apex:outputPanel>
```

You just created an output panel identified as `msg`, which displays an item from the controller called `"message"`, which hasn't been created yet.
4. Click **Save**. The Page Editor will figure out that you don't have any method or property called "message" and offer to create it for you.
5. Click **Create Apex property StoreFront.message**.
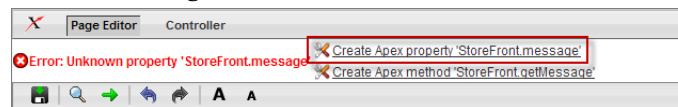
6. Now, change the `commandButton` tag to include a `reRender` attribute.

```
<apex:commandButton action="{!shop}" reRender="msg" value="Buy" />
```

You have now modified the Visualforce page to use an in-place AJAX update. It updates the panel identified as "msg" after it calls the `shop()` method on the controller. Now modify the `shop()` method to update the message property displayed in the panel with a list of items that were selected.

1. Click the Controller tab next to the Page Editor tab.
2. Locate the `shop()` method (lines 4 through 6) and replace it with the following code.

```
public PageReference shop() {
    message = 'You bought: ';
    for (DisplayMerchandise p: products) {
      if (p.count > 0) {
        message += p.merchandise.name + ' (' + p.count + ')     ';
      }
    }
    return null;
    }
```

> **Note:** Notice how this code simply utilizes the `products` variable. Visualforce will automatically ensure that the data you change in the user interface is reflected by the data in the `products` variable.

3. Click the **Save** icon.

4. Now test the shopping cart. Add a number to one of the merchandise items, and click **Buy**. You'll notice that a text field appears beneath the merchandise list, indicating how many products you purchased.

Tell Me More....

This simple addition exposes a lot of powerful functionality.

- As you saw in this step, Visualforce automatically mirrored the data changes on the form back to the products variable. This functionality is extremely powerful, and lets you quickly build forms and other complex input pages.
- When you click the **Buy** button, the panel updates without updating the entire screen. The AJAX effect that does this, which typically requires complex JavaScript manipulation, was accomplished with a simple `reRender` attribute.

Summary

Apex is a powerful object-oriented language, with many of the features typically found in similar languages such as Java. In this tutorial you created an Apex class and used a few of the features of the language, such as arrays, iteration, and querying the database. If you did the bonus step, you modified the Visualforce page to use the data passed back to the controller to display a message using an AJAX update, without refreshing the entire page.

The next thing you might do is create an authenticated portal, so that people can log into your site, create a username and password, and purchase merchandise. In order to do that, you'd need to create a portal, which is outside the scope of this workbook. However, we have included detailed steps on how to create a portal that works with the app you just built. Visit developer.force.com/workbook for the portal tutorial and other new additions to the workbook.

Another thing you might want to do next is to deliver your app, by either packaging for distribution or deploying it to a production organization. For more information, see the *ISVforce Workbook*.