
Visualforce Workbook

Version 6, Spring '16



CONTENTS

Welcome to the Visualforce Workbook	1
Who this Workbook is For	2
Introduction to Visualforce	3
Creating and Listing Visualforce Pages	3
Enable Visualforce Development Mode	3
Create a Visualforce Page	3
Edit the Visualforce Page	4
Find All Visualforce Pages	5
Alternative Page Creation	5
Summary	5
Adding Attributes and Using Auto-Suggest	5
Add Attributes Using Auto-Suggest	6
Add Additional Components	7
Add Nested Components	7
Summary	8
Understanding Simple Variables and Formulas	8
Global Variables	9
Basic Formulas	9
Conditionals	10
Summary	10
Using Standard Controllers	11
Find Identifiers of Records	11
Display Data from a Record	12
Display Other Fields	13
Display Fields from Related Records	13
Summary	14
Using Standard User Interface Components	14
Display a Record or Related Lists	14
Display Fields	15
Display a Table	15
Summary	16
Updating Visualforce Pages with Ajax	16
Identify a Region for Dynamic Updates	16
Add Dynamic Re-Rendering	17
Summary	17
Overriding and Pointing to Pages	18
Override the Standard Display for a Page	18
Embed a Page on a Standard Layout	19

Contents

Create a Button that Links to a Visualforce Page	19
Create Hyperlinks to URLs or Other Visualforce Pages	20
Summary	20
Inputting Data with Forms	21
Create a Basic Form	21
Show Field Labels	22
Display Warning and Error Messages	22
Summary	22
Reusing Pages with Templates	23
Create a Template	23
Use a Template with Another Page	23
Include One Visualforce Page within Another	24
Summary	24
Introduction to Apex	25
Set Up Your Development Environment	25
Install the Enhanced Warehouse Data Model	25
Access the Mobile Browser Web App	26
Download the Salesforce1 Mobile App	26
Using the Developer Console	26
Activating the Developer Console	26
Using the Developer Console to Execute Apex Code	27
Summary	29
Creating and Instantiating Classes	29
Creating an Apex Class Using the Developer Console	29
Calling a Class Method	31
Creating an Apex Class Using the Salesforce User Interface	31
Summary	32
Creating the WarehouseUtils Class	32
Create the WarehouseUtils Apex Class	33
Add a “Stub” findNearbyWarehouses Method	33
Perform a Query and Return the Results	34
Summary and Code Check	35
Testing and Debugging the WarehouseUtils Class	36
Create an Apex Test Class	36
Add a Test Method and Setup Code	36
Test the findNearbyWarehouses Method	38
Run the Test and Review Test Results	39
Find the Bug	41
Write a Test for the Bug	41
Fix the Bug	42
Summary and Code Check	43
Visualforce and Apex In Action	46

Contents

Creating Location-Aware Visualforce Pages	46
Create a Visualforce Page Linked to the WarehouseUtils Class	46
Add Static Resources to the Page	47
Add a Place to Display the Map	48
Add JavaScript to Query for Warehouses	48
Add JavaScript to Build the Map	49
Add JavaScript to Add Warehouse Markers to the Map	50
Summary and Code Check	51
Add the Nearby Warehouses Page to Salesforce1	54
Create a Tab for the Page	55
Add the Tab to Mobile Navigation	55
Try Out the App	55
Summary	57
Visualforce Pages with Apex Controllers	57
Displaying Product Data in a Visualforce Page	57
Using a Custom Apex Controller with a Visualforce Page	59
Using Inner Classes in an Apex Controller	61
Adding Action Methods to an Apex Controller	63
Summary	65
Conclusion and Where to Go From Here	66

WELCOME TO THE VISUALFORCE WORKBOOK

Visualforce is a framework that allows developers to build sophisticated, custom user interfaces that can be hosted natively on the Force.com platform. This workbook provides an introduction to many of the features in Visualforce, as well as a look at how you can use Apex to add complex logic to your Visualforce pages.

You'll learn how to build user interfaces that look like the standard user interface provided by Force.com, as well as how to build your own user interfaces with all the control that HTML, CSS, and JavaScript provide. Along the way you'll find out how to create components, reusable pieces of Visualforce, as well as how to hook Visualforce into your applications. You'll also learn about the Model–View–Controller (MVC) foundations of Visualforce, and the fundamentals of Apex code.

Workbook Version

This workbook is current for Winter '15, and was last revised on September 5, 2014. You should be able to complete all of the tutorials using the Winter '15 version of Force.com (API version 36.0) or later.

To download the latest version of this workbook, go to https://developer.salesforce.com/page/Force.com_workbook.

Before You Begin

These tutorials are designed to work with a Force.com Developer Edition organization, or *DE org* for short. DE orgs are environments with all of the features and permissions that allow you to develop, package, test, and install apps. You can get your own DE org for free at <http://sforce.co/ZfioJ6>, and you can use the techniques that you learn in this workbook in all Force.com environments that support development.

It would also help to have a little context by learning a little about Force.com itself, which you can find in the first few tutorials of the [Force.com Workbook](#).

Finally, you'll need a [browser supported by Salesforce](#). Modern versions of Chrome, Firefox, Safari, and even Internet Explorer should do the trick.

After You Finish

After you finish the workbook, you'll be ready to explore a lot more Visualforce and Force.com development. Here's a quick list of resources.

- Learn more about declarative (clicks, not code) Force.com development from the companion Force.com Workbook at https://developer.salesforce.com/page/Force.com_workbook.
- Download the Visualforce Cheat Sheet at https://developer.salesforce.com/page/Cheat_Sheets.
- Get in-depth documentation for Visualforce in the [Visualforce Developer's Guide](#).
- Start learning the Apex programming language in depth with the [Apex Workbook](#).
- Discover more Force.com and access articles, documentation, and code samples by visiting Developer Force at <http://developer.salesforce.com>.

Who this Workbook is For

This workbook is designed for two audiences.

- Experienced web developers, who have a solid understanding of HTML markup, and probably know how to write code in JavaScript or a back-end language such as PHP, Ruby, C#, or Java. (If you know Apex, you're ahead already.)
- Experienced Salesforce admins, who know Salesforce and have some basic HTML experience, but who might not have the programming background of a web developer.

Both groups can learn a lot from this book.

This book is organized into three sections.

- **Introduction to Visualforce** teaches the basics of Visualforce markup. It's great if you know HTML markup, but you don't need to be a programmer to follow every lesson. You'll work with the built-in Salesforce objects such as Accounts and Contacts, and you'll stick to Visualforce and HTML markup—no programming. You'll be surprised how far “pure Visualforce” takes you!
- **Introduction to Apex** provides a gentle introduction to Apex, the programming language of the Force.com platform, focused on how you use it with Visualforce. If you're a programmer, you'll have no difficulty applying what you already know to quickly understand how to write custom logic for your Force.com apps. Adventurous non-coders should also be able to follow along and understand the basics. You might be inspired to learn a new skill!
- **Visualforce and Apex In Action** shows you how you can use the two together to create apps with custom user interfaces and behavior. There's no getting around it, there's a lot of code to understand in this section. You'll work with Visualforce, Apex, JavaScript, the Google Maps API, and custom objects to create an app your users can access on the go in Salesforce1. That's a lot of buzzwords in one sentence, but we think you'll be surprised and delighted at how easy it is to create a location-aware page for your mobile users.

If you're an experienced developer, you might be tempted to jump to the second section. We recommend you at least read the first section, even if you skip the exercises, to understand the basics of the markup language. Also, there's a lot you can do with Visualforce by itself—code you don't write is code you don't have to maintain.

If you're not a programmer, you might be intimidated by the code in this book. Don't be. Learning Salesforce is an achievement, and if you can do that, you can follow all of the exercises in this book. You don't have to understand every line of code to learn useful techniques.

If you're an admin just getting started with Force.com, you might find this book a little challenging. See the [Force.com Platform Fundamentals](#) book for an introduction to the platform and point-and-click app development.

INTRODUCTION TO VISUALFORCE

Visualforce is a component-based user interface framework for the Force.com platform. Visualforce allows you to build sophisticated user interfaces 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 section you'll learn the basics of the Visualforce markup language. We'll focus on the fundamentals, and work with the built-in objects included with Salesforce—Account, Contact, and so on.

When you're finished with this section, you will have done the following.

- Create new Visualforce pages and edit existing pages.
- Use two different Visualforce editors, and use the auto-suggest tools for adding Visualforce components and attributes.
- Design pages by combining simple built-in Visualforce components into larger page elements and structures.
- Load data from your organization and display it on the page, in detail and list views.
- Create forms that capture changes to data and save it to Salesforce.
- Add your custom pages to Salesforce where your users can access them, including overriding the built-in Salesforce create, edit, and view pages.
- Perform page changes using Ajax to update parts of the page without reloading the whole page.

Creating and Listing Visualforce Pages

In this tutorial, you'll learn how to create and edit your first Visualforce page. The page will be really simple, but this is the start, and we'll soon expand on it. Along the way you'll familiarize yourself with the editor and automatic page creation.

Before you start, please create a free Force.com Developer Edition organization, as indicated earlier in the “Before you Begin” section.

Enable Visualforce Development Mode

Development mode embeds a Visualforce page editor in your browser that 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. From your personal settings, enter *Advanced User Details* in the Quick Find box, then select **Advanced User Details**.
No results? Enter *Personal Information* in the Quick Find box, then select **Personal Information**.
2. Click **Edit**.
3. Select the Development Mode checkbox, then click **Save**.

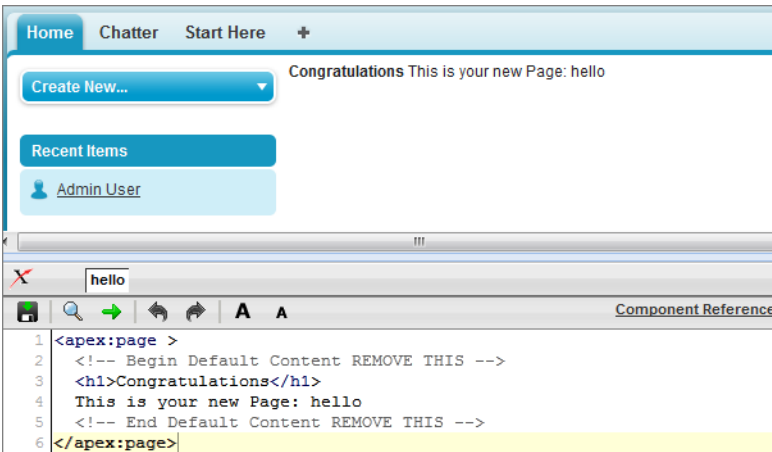
Create a Visualforce Page


Now you are ready to create your first Visualforce page.

1. In your browser, add `/apex/hello` to the URL for your Salesforce instance. For example, if your Salesforce instance is `https://na1.salesforce.com`, the new URL is `https://na1.salesforce.com/apex/hello`. You will see the following error:



2. Click the **Create Page hello** link to create the new page. You will see your new page with some default markup.



 **Note:** If you don't see the Page Editor below the page, just click the **hello** tab in the status bar.

That's it! The page includes some default text, as well as an embedded page editor displaying the source code. This is the primary way you'll be creating pages in this section of the workbook.

Edit the Visualforce Page

Now that you've created the Visualforce page, you need to customize it for your own use. You can edit and preview the changes in real time.

1. You don't want the heading of the page to say "Congratulations," so change the contents of the `<h1>` tag to Hello World, remove the comments, and the "This is your new page" text. The code now looks like this:

```
<apex:page>
  <h1>Hello World</h1>
</apex:page>
```

2. Click the **Save** button at the top of the Page Editor.

The page reloads to reflect your changes. Note that Hello World appears in a large font. This is because of the `<h1>` tag—a standard HTML tag. Visualforce pages are generally composed of two types of tags: tags that identify Visualforce components (such as `<apex:page>`), and tags that are standard HTML.

Development mode, which you enabled in Step 1, makes development fast and easy. You can simply make changes, press Save, and immediately see the changes reflected. You can use a keyboard shortcut too—click CTRL+S to save at any time. You can also click the editor minimize button to see the full page.

When you deploy the page in a production environment, or if you switch off development mode, the editor will no longer be available.

Find All Visualforce Pages

Now that you've created a Visualforce page, you'll need to know where to find it.

1. From Setup, enter *Visualforce Pages* in the **Quick Find** box, then select **Visualforce Pages**.
2. Scroll down to locate the page created in Step 2—`hello`.

This views your page, and even allows you to edit it. However, this editor is different from the one we've seen in the previous steps—it also doesn't let you immediately view the changes (unless you have the page open in a separate tab).

Alternative Page Creation

You can also create a new page from this listing, and then edit it just like you did in Step 2 by navigating to the correct URL—taking into account the name of the page you created. Try it!

1. From Setup, enter *Visualforce Pages* in the **Quick Find** box, then select **Visualforce Pages**, then click **New**.
2. Create and label the page `hello2`.
3. Click **Save**.
4. Navigate to the new page using the URL as you did in Step 2: `https://your-salesforce-instance/apex/hello2`

The Visualforce editor in Setup is good to know about, and a great way to see all your pages. However, the Development Mode editor we used in previous steps is more powerful, and lets you view your changes immediately. We'll use it for the rest of this section of the workbook.

Summary

You now know how to enable development mode, and list and create Visualforce pages. In the next tutorial, you'll learn a little about the page editor, and the basics of Visualforce components, which are the building blocks of any page.

Adding Attributes and Using Auto-Suggest

The page you created in Tutorial #1 shares a characteristic of every Visualforce page—it starts and ends with the `<apex:page>` tag. `<apex:page>` is actually a Visualforce component—and one that must always be present. So all Visualforce pages will look similar to this:

```
<apex:page>
  Your Stuff Here
</apex:page>
```

Note the use of angle brackets, as well as how you indicate where a component starts and ends. The start is simply the component name in angle brackets: `<apex:page>`. The end is the component name prepended with a `/` character in angle brackets: `</apex:page>`. All Visualforce pages follow this same convention—requiring that the pages you create be “well-formed XML” (with a few exceptions). A few components are self-closing—they have this form: `<apex:detail />` (note the position of the `/`). Think of that as a start and end tag wrapped up in one!

You can generally modify the behavior and/or appearance of a component by adding attributes. These are name/value pairs that sit within the start tag. For example, here's an attribute: `sidebar="false"`.

Add Attributes Using Auto-Suggest

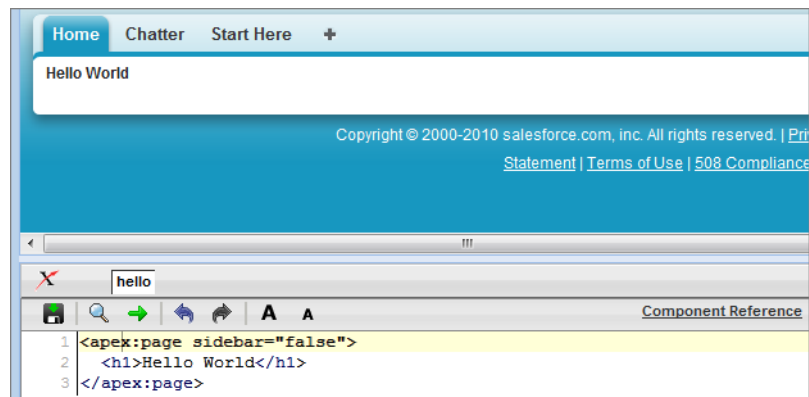
Let's play some more with our first hello page. It turns out that the sidebar attribute is a valid attribute for the `<apex:page>` component.

1. Add `sidebar="false"` within the start tag of the `<apex:page>` component as follows:

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

2. Click **Save**.

Notice that the left hand area of your page has changed—the sidebar has been removed. In effect, the sidebar attribute modifies the behavior and appearance of the `<apex:page>` component.

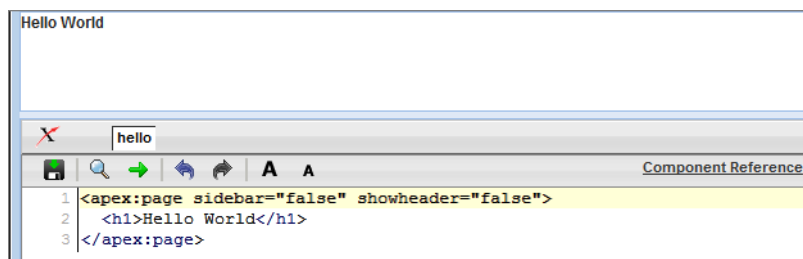


3. Position your cursor just after the final quotation mark ("), and hit the space bar. A helpful list of attributes pop up that are valid for the `<apex:page>` component. Choose the `showHeader` attribute.
4. The attribute is automatically added to your page, and you now need to supply a value for the attribute. Add `false`. Your complete first line should look like this:

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

5. Click **Save** (remember, you can also press CTRL+S as a shortcut).

This time your page looks completely different. By setting the `showHeader` attribute to false, you've not only removed the top header, but all the default styling associated with the page.



Let's put it back the way it was—having the top header is very useful during development.

6. Change the `showHeader` attribute's value to `true`.
7. Click **Save**.

Add Additional Components

You've created a page, used the `<apex:page>` component, and changed its behavior. You'll typically want to use additional components that supply a lot more functionality.

Visualforce comes with a several dozen built-in components, and you can install and build your own components to extend this set. In this lesson you'll learn how to locate them, and use one.

1. Click the **Component Reference** link in the Page Editor. A help popup window displays with all available components.
2. Click `<apex:pageBlock>`. A description of what the component does, and what attributes you can add to change its behavior displays in the Component Details tab.
3. Click the Usage tab to see an example of how to use the component. You'll notice that the `<apex:pageBlock>` component is often used with the `<apex:pageBlockSection>` component. Click `<apex:pageBlockSection>` to learn more about that component.

In general, you'll dip into the component reference whenever you need to. You'll soon learn what the major components do—and while some of them take a large number of attributes, in practice you will only use a handful.

Now add both components to your page. We're going to go a little faster here—see if you can do this without looking at the final code below.

4. Within the `<apex:page>` component, add an `<apex:pageBlock>` component with a `title` attribute set to *A Block Title*.
5. Within the `<apex:pageBlock>` component, add an `<apex:pageBlockSection>` component, with its `title` attribute set to *A Section Title*.
6. Within the `<apex:pageBlockSection>`, add some text, like *I'm three components deep!*
7. Click **Save**. Your final code will look something like this:

```
<apex:page sidebar="false">
  <apex:pageBlock title="A Block Title">
    <apex:pageBlockSection title="A Section Title">
      I'm three components deep!
    </apex:pageBlockSection>
  </apex:pageBlock>
</apex:page>
```

The final page will look something like this:



You can click the tiny disclosure triangle next to A Section Title to minimize that section of the block.

Add Nested Components

Adding additional components is easy.

1. Navigate to the end of the `<apex:pageBlockSection>` component, and add another `<apex:pageBlockSection>` component with its own title. Both `<apex:pageBlockSection>` components must be contained within the same `<apex:pageBlock>` component.
2. Click **Save** and admire your handiwork.

```
<apex:page sidebar="false">
  <apex:pageBlock title="A Block Title">
    <apex:pageBlockSection title="A Section Title">
      I'm three components deep!
    </apex:pageBlockSection>
    <apex:pageBlockSection title="A New Section">
      This is another section.
    </apex:pageBlockSection>
  </apex:pageBlock>
</apex:page>
```

Note the number of “nested” components. The start and the end tag for an `<apex:pageBlockSection>` are both within the start and end tag for the `<apex:pageBlock>` component. And your first `<apex:pageBlockSection>` ends before the next one starts (its end tag, `</apex:pageBlockSection>`, appears before the start of the new one, `<apex:pageBlockSection>`). All of the components on a Visualforce page tend to nest in this way—and the editor tells you when you’ve made a mistake (for example, if you forget an end tag).

Summary

In this tutorial you learned how to change the behavior and appearance of Visualforce components by adding attributes, how to use the auto-suggest feature of the editor, and how to use the Component Reference to look up additional components. You also learned that Visualforce components are often nested within each other.

Learning More

Here are additional Visualforce components that let you build pages that match the platform visual style:

- `<apex:pageBlockButtons>` lets you provide a set of buttons that are styled like standard user interface buttons
- The optional `<apex:pageBlockSectionItem>` represents a single piece of data in a `<apex:pageBlockSection>`
- `<apex:tabPanel>`, `<apex:toolbar>`, and `<apex:panelGrid>` provide other ways of grouping information on a page

Understanding Simple Variables and Formulas

The Visualforce pages you’ve created so far have been static. In general, Visualforce pages are dynamic—they can display data retrieved from the database, or data that changes depending on who is logged on and viewing the page. They can become dynamic through the use of variables and formulas.

This tutorial introduces you to variables, formulas and the expression language syntax used in Visualforce. Variables typically contain information that you have retrieved from objects in the Force.com database, or which the platform has made available to you—for example, the name of the logged-in user. A number of built-in formulas are available to add functionality to your page—you’ll discover some basic formulas in this tutorial too.

Global Variables

Force.com retains information about the logged-in user in a variable called `User`. You can access fields of this `User` variable (and any others) by using a special expression language syntax: `{! $<global variable>.<field name>}`

1. Modify your existing page to include the following line: `{! $User.FirstName}`. Remember that any content must lie within the `<apex:page>` component (between its open and closing tags).
2. Click **Save**.

Your Visualforce page looks something like this:

```
<apex:page sidebar="false">
    {! $User.FirstName}
</apex:page>
```

In the future we'll assume that you know to put any Visualforce markup within the `<apex:page>` tag. We'll also assume that by now you're comfortable enough to "Click Save" and view the result as well!

The `{! ... }` tells Visualforce that whatever lies within the braces is dynamic and written in the expression language, and its value must be calculated and substituted at run time when someone views the page. Visualforce is case-insensitive, and spaces within the `{! ... }` syntax are also ignored. So this is just as effective: `{! $USER.firstname}`.

Here's how to show the first name and last name of the logged-in user: `{! $User.FirstName} {! $User.LastName}`

Basic Formulas

Visualforce lets you embed more than just variables in the expression language. It also supports formulas that let you manipulate values. The `&` character is the formula language operator that concatenates strings.

1. Add this to your Visualforce page: `{! $User.firstname & ' ' & $User.lastname}`

This tells Visualforce to retrieve the `firstname` and `lastname` fields from the global `User` object, and to concatenate them with a space character. The output will be something like: Joe Bloggs.

In general, formulas are slightly more advanced and have a simple syntax that includes the function name, a set of parentheses, and an optional set of parameters.

2. Add this to your Visualforce page:

```
<p> Today's Date is {! TODAY()} </p>
<p> Next week it will be {! TODAY() + 7} </p>
```

You'll see something like this in the output:

```
Today's Date is Wed Feb 08 00:00:00 GMT 2012
Next week it will be Wed Feb 15 00:00:00 GMT 2012
```

The `<p>` tags are standard HTML for creating paragraphs. In other words, we wanted both sentences to be in individual paragraphs, not all on one line. The `TODAY()` function returns the current date as a date data type. Note how the time values are all set to 0. Also note the `+` operator on the date. The expression language assumes you want to add days, so it added 7 days to the date.

3. You can use functions as parameters in other functions, and also have functions that take multiple parameters too. Add this:

```
<p>The year today is {! YEAR(TODAY())}</p>
<p>Tomorrow will be day number  {! DAY(TODAY() + 1)}</p>
<p>Let's find a maximum:  {! MAX(1,2,3,4,5,6,5,4,3,2,1)} </p>
```

```
<p>The square root of 49 is {! Sqrt(49)}</p>
<p>Is it true?  {! CONTAINS('salesforce.com', 'force.com')}</p>
```

The output will look something like this:

```
The year today is 2012
Tomorrow will be day number 9
Let's find a maximum: 6
The square root of 49 is 7.0
Is it true? true
```

The `CONTAINS()` function returns a boolean value: something that is either true or false. It compares two arguments of text and returns true if the first argument contains the second argument. If not, it returns false. In this case, the string “force.com” is contained within the string “salesforce.com”, so it returns true.

Conditionals

Sometimes you want to display something dynamically, based on the value of an expression. For example, if an invoice has no line items, you might want to display the word “none” instead of an empty list, or if some item has expired, you might want to display “late” instead of showing the due date.

You can do this in Visualforce by using a conditional formula expression, such as `IF()`. The `IF()` expression takes three arguments:

- The first is a boolean: something that is either true or false. You’ve seen an example of that in the `CONTAINS()` function.
- The second argument is something that will be returned if the boolean is true.
- The third argument is something that will be returned if the boolean is false.

Insert the following and try to predict what will be displayed if you save the page:

```
{! IF ( CONTAINS('salesforce.com','force.com'), 'Yep', 'Nah') }
{! IF ( DAY(TODAY()) > 14, 'After the 14th', 'On or before the 14th') }
```

You’ll see something like this:

```
Yep
On or before the 14th
```

Of course, this all depends on when you run the code. After the 14th in a month, it looks different.

Summary

Visualforce lets you embed operations that evaluate at runtime using a special expression language syntax: `{! expression}`. Global variables are accessed using the `$variableName` syntax. The expression language lets you manipulate strings, numbers, text, and dates, as well as conditionally execute operations.

Learning More

- The [Formulas Cheat Sheet](#) provides a concise guide to the many formulas you can use.
- The [Visualforce Developer’s Guide](#) has a lot more detail on formulas.

Using Standard Controllers

Visualforce's Model-View-Controller (MVC) 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 can contain the logic to be executed when a button is clicked. A controller also typically interacts with the model (the database)—making available data that the view might want to display.

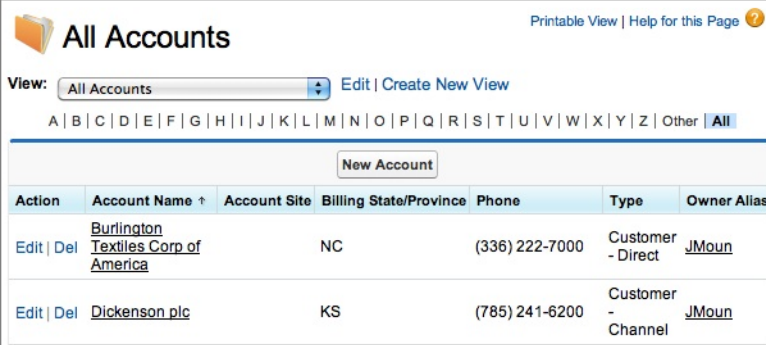
Most Force.com objects have default standard controllers that can be used to interact with the data associated with the object, so in many cases you don't need to write the code for the controller yourself. You can extend the standard controllers to add new functionality, or create custom controllers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

Find Identifiers of Records

When your Visualforce pages interact with other pages in your application, you can automatically pass in the record's identifier, and your Visualforce page can then display that data. Right now your pages are stand-alone, so for your page to display data from a record in the database, it needs to know the record's identifier.

Your Developer Edition environment has a number of objects that store data, available out of the box.

1. For example, switch to the Sales application by choosing **Sales** from the drop down.
2. Now select the Accounts tab. Ensure the pick list shows All Accounts and click **Go** to view all the account records.



Action	Account Name ↑	Account Site	Billing State/Province	Phone	Type	Owner Alias
Edit Del	Burlington Textiles Corp. of America		NC	(336) 222-7000	Customer - Direct	JMoun
Edit Del	Dickenson plc		KS	(785) 241-6200	Customer - Channel	JMoun

3. Click Burlington Textiles (or any other record) to view the details. Your screen displays all the details for that account:

Account
Burlington Textiles Corp of America
 Customize Page | Edit Layout | Printable View | Help for this Page

Show Chatter **New!** | Unfollow

« Back to List: Accounts

[Contacts \[1\]](#) |
 [Opportunities \[1\]](#) |
 [Cases \[2\]](#) |
 [Open Activities \[0\]](#) |
 [Activity History \[0\]](#) |
 [Notes & Attachments \[0\]](#) |
 [Partners \[0\]](#)

Account Detail [Edit](#) [Delete](#) [Include Offline](#)

Account Owner	Jon Mountjoy [Change]	Rating	Warm
Account Name	Burlington Textiles Corp of America [View Hierarchy]	Phone	(336) 222-7000
Parent Account		Fax	(336) 222-8000
Account Number	CD656092	Website	http://www.burlington.com
Account Site		Ticker Symbol	BTXT
Type	Customer - Direct	Ownership	Public
Industry	Apparel	Employees	9,000
Annual Revenue	€350,000,000	SIC Code	546732
% Closed	1	% Pipe + Lost	0.00

Notice that your URL has changed—it now looks something like this: `https://<your salesforce instance>.salesforce.com/0018000000MDfn1`

The identifier is that series of digits at the end, in this case, 0018000000MDfn1. The identifier, or ID as it's often written, is unique across all records in your database. If you know the ID for any record, and have permission, you can often construct a URL to view it by replacing 0018000000MDfn1 with the record's identifier.

When you visited `https://<salesforce instance>.salesforce.com/0018000000MDfn1`, Force.com automatically retrieved the record with identifier 0018000000MDfn1 from the database, and automatically constructed a user interface for it. In the other lessons in this tutorial, you're going to take away some of the automation, and create your own user interface.

Display Data from a Record

Create a new Visualforce page, `accountDisplay`, with the following content:

```
<apex:page standardController="Account">
  <p>Hello {! $User.FirstName}!</p>
  <p>You are viewing the {! account.name} account.</p>
</apex:page>
```

You'll recognize the `{! }` expression syntax from the previous tutorial, and that `$User.FirstName` refers to the First Name field of the User global variable. There are a few new things though:

1. The `standardController="Account"` attribute tells Visualforce to use an automatically-generated controller for the Account object, and to provide access to the controller in the Visualforce page.
2. The `{! account.name}` expression retrieves the value of the account variable's name field. The account variable is automatically made available by the standard controller (it's named after the standard controller's name).

Controllers generally have logic that handles button clicks and interacts with the database. By using the `standardController` attribute, your Visualforce page has access to a rich controller that is automatically generated for you.

The standard controller for the Account object determines when an identifier is being used in the page, and if it is, queries the database and retrieves the record associated with that identifier. It then assigns that record to the account variable so that you can do as you please with the data in your Visualforce page.

When you click **Save**, you will see your first name and an empty account name. This is because you haven't told the Visualforce page which account record to display. Go to your URL and modify it so that you include the ID from Step 1. So instead of something like:

```
https://na3.salesforce.com/apex/accountDisplay
```

change it to something like:

```
https://na3.salesforce.com/apex/accountDisplay?id=0018000000MDfn1
```

In your case, change the identifier 0018000000MDfn1 to whatever you found in Step 1. You might need to change "na3" as well, to whatever your salesforce instance currently is.

Now when you save your work, the account name displays:

Hello Jon!

You are viewing the Burlington Textiles Corp of America account.

Display Other Fields

Your `accountDisplay` page only displays the name field of the Account object. To find other fields to display for the object, from the object management settings for accounts, go to the fields area. Click any field, such as Ticker Symbol. The Field Name field provides the name that you can use in your own Visualforce pages. For example, for this particular field, the name is `TickerSymbol`.

Modify `accountDisplay` to include this field by adding the following paragraph after the existing one:

```
<p>Here's the Ticker Symbol field: {! account.TickerSymbol}</p>
```

Display Fields from Related Records

You can also display data from related records. For example, while viewing the object details for Account, you might have noticed that the Account object has a field called Account Owner, and that its type is `Lookup(User)`. In other words, this field has a relationship to a User record. By clicking the Account Owner field label link, you'll discover its Field Name is `Owner`.

The Owner relationship represents a User. And, if you go to the fields area from the object management settings for users, you'll find that User has a Name field. Let's use this information to display it.

1. Modify `accountDisplay` to include this field by adding the following paragraph after the existing one:

```
<p>Here's the owner of this account: {! account.Owner.Name}</p>
```

The "dot notation" (`account.Owner.Name`) indicates that you want to traverse the relationship between the records. You know that `account.Owner` indicates the Owner field of the account record. The extra name at the end indicates that the owner field isn't a simple field representing a String, but a relationship to another record (it's a `Lookup(User)`), and that you'd like to get the record represented by the value of the Owner field (it's a User record), and display the Name field on that record.



Tip: If you've created your own custom objects (instead of using objects like Account) and want to know how to reference a field, you have to follow a slightly different procedure. Go to the management settings for custom objects. Then select your object, and then the field. The API Name now indicates the name of the field that you must use in your Visualforce pages. For example, if your field was called `Foo`, its API Name is `Foo__c`, and you'd reference it with that name—something like:

```
{! myobject__c.foo__c}.
```

Summary

Standard controllers provide basic, out-of-the-box, controller functionality, including automatic record retrieval. This tutorial showed how to locate the identifier for a record and use the standard controller to display the record's data. The standard controller also contains functionality to save or update a record, which you'll see later.

Learning More

Visualforce also supports [standard list controllers](#), which allow you to create Visualforce pages that can display or act on a set of records, with pagination.

Using Standard User Interface Components

In [Adding Attributes and Using Auto-Suggest](#) you learned about the `<apex:pageBlockSection>` component, and in the previous tutorial you learned how to show some data from an Account record using the expression language. In this tutorial you'll discover additional Visualforce components that produce output that looks and feels like the automatically-generated user interfaces.

Display a Record or Related Lists

Creating a list of records is as easy as typing up a single component.

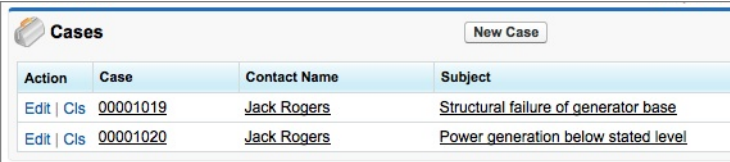
1. Modify your `accountDisplay` Visualforce page to look like this:

```
<apex:page standardController="Account">
    <apex:detail/>
</apex:page>
```

If you access your page with a valid account ID passed in as a parameter, as demonstrated in the previous tutorial (it will look something like this: `https://na3.salesforce.com/apex/AccountDisplay?id=0018000000MDfn1`), then you'll see a lot of output.

2. The `<apex:detail/>` component displays the standard view page for a record. It shows related lists as well, such as contacts. You can switch these off by adding the `relatedList="false"` attribute. Try adding it, click **Save**, and spot the difference.
3. You can show only a particular related list; such as the list of case records that are related to the account record you are viewing. Add the following tag:

```
<apex:relatedList list="Cases" />
```



Action	Case	Contact Name	Subject
Edit Cls	00001019	Jack Rogers	Structural failure of generator base
Edit Cls	00001020	Jack Rogers	Power generation below stated level

Although these tags are very simple, they're doing a lot of work for you—and relying on that standard controller to go and retrieve the data. Sometimes, however, you want more control over page layout.

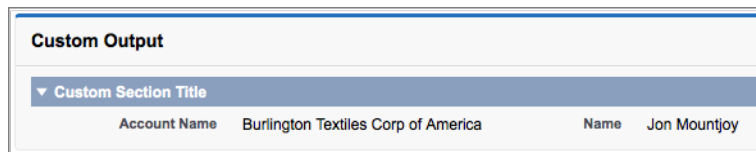
Display Fields

If you want to selectively determine a record's fields for display, use the `<apex:outputField>` component. This component, when embedded in the `<apex:pageBlock>` component, displays not only the value of the field, but also the field's label.

1. Insert the following into your page to see it in action:

```
<apex:pageBlock title="Custom Output">
  <apex:pageBlockSection title="Custom Section Title">
    <apex:outputField value="{!account.Name}" />
    <apex:outputField value="{!account.Owner.Name}" />
  </apex:pageBlockSection>
</apex:pageBlock>
```

Here, `account.Name` specifies the current account record's name field, whereas `account.Owner.Name` looks at the owner field of the account record, and then retrieves that record's name field.



Display a Table

In the previous lessons, you displayed individual fields and a complete record. Sometimes however, you need to display a set of fields across a number of records—for example, the list of contacts associated with the account. In this step you will list the contacts of an account record by iterating over the list and displaying each one individually. It may seem complex initially because there are multiple tags that nest within each other, but you will find it second nature in no time. Don't forget you can always click the **Component Reference** link to learn more about each.

1. First start by adding an `<apex:pageBlock>` component:

```
<apex:pageBlock title="My Account Contacts">
</apex:pageBlock>
```

2. You can save and view the result if you like. Now within this component, insert another one, the `<apex:pageBlockTable>` component:

```
<apex:pageBlockTable value="{! account.contacts}" var="item">
</apex:pageBlockTable>
```

You can think of this component as doing the following: it takes a look at its `value` attribute, and retrieves the list of records that it finds there. In this case, it retrieves the `contacts` field that represents the list of Contact records associated with the current Account record. Then, for each individual Contact record, it assigns it to a variable called `item`. It does this repeatedly until it reaches the end of the list. The key lies in the body of the component. This will be output at each iteration—effectively allowing you to produce something for each individual record.

3. Ideally, you want to insert something inside of the `<apex:pageBlockTable>` component that does something with the current item. Try adding this:

```
<apex:column value="{! item.name}" />
```

The `<apex:column>` component creates a new column within the table. It adds a table header based on the name of the field, and also outputs the values for that field in the rows of the table, one row per record. In this case, you have specified `{! item.name}`, which will show the name field for each of the Account's Contacts.

My Account Contacts	
Name	
Jack Rogers	

Here's what your final code looks like:

```
<apex:pageBlock title="My Account Contacts">
  <apex:pageBlockTable value="{! account.contacts}" var="item">
    <apex:column value="{! item.name}"/>
  </apex:pageBlockTable>
</apex:pageBlock>
```

Contact records also have a field called phone. Try adding a column to display the phone numbers. Of course, if you don't have any contacts associated with the account that you're viewing, or if you haven't supplied an account ID, then it won't display anything.

Summary

The `<apex:detail>` and `<apex:relatedList>` components make it tremendously easy to display records and related lists by utilizing the standard controller to automatically retrieve the record's data. The `<apex:pageBlockTable>` component provides a way to iterate over a list of records, producing output for each record in the list.

Learning More

- Use `<apex:facet>` to customize the caption, headers and footers of a table.
- The `<apex:enhancedList>` and `<apex:listViews>` components provide a way to embed a standard list view of an object's records.

Updating Visualforce Pages with Ajax

Visualforce lets you use Ajax effects, such as partial page updates, without requiring you to implement any complex JavaScript logic. The key element is identifying what needs to be dynamically updated, and then using the `reRender` attribute to dynamically update that region of the page.

Identify a Region for Dynamic Updates

A common technique when using Ajax in Visualforce is to group and identify the region to be dynamically updated. The `<apex:outputPanel>` component is often used for this, together with an `id` attribute for identifying the region.

1. Create a Visualforce page called `Dynamic`, using the following body:

```
<apex:page standardController="Account">
  <apex:pageBlock title="{!account.name}">
    <apex:outputPanel id="contactDetails">
```

```

        <apex:detail subject="{!$CurrentPage.parameters.cid}"
            relatedList="false" title="false"/>
    </apex:outputPanel>
</apex:pageBlock>
</apex:page>

```

2. Ensure that your Visualforce page is called with an identifier for a valid account.

Your Visualforce page won't show much at all except for the account name. Note that the `<apex:outputPanel>` has been given an identifier named `contactDetails`. Also note that the `<apex:detail>` component has a `subject` attribute specified. This attribute is expected to be the identifier of the record whose details you want to display. The expression `{! $CurrentPage.parameters.cid}` returns the `cid` parameter passed to the page. Since you're not yet passing in such a parameter, nothing is rendered.

Add Dynamic Re-Rendering

Now you need to add elements to the page that set the page parameter and dynamically render the region you've named `detail`:

1. Modify your page by adding a new page block beneath your current one:

```

<apex:pageBlock title="Contacts">
    <apex:form>
        <apex:dataList value="{! account.Contacts}" var="contact">
            {! contact.Name}
        </apex:dataList>
    </apex:form>
</apex:pageBlock>

```

This iterates over the list of contacts associated with the account, creating a list that has the name of each contact.

2. Click **Save**.

If you access your page, you'll see the list of contacts. Now you need to make each contact name clickable.

3. Modify the `{! contact.Name}` expression by wrapping it in an `<apex:commandLink>` component:

```

<apex:commandLink re-render="contactDetails">
    {! contact.Name}
    <apex:param name="cid" value="{! contact.id}"/>
</apex:commandLink>

```

There are two important things about this component. First, it uses a `re-render="contactDetails"` attribute to reference the output panel you created earlier. This tells Visualforce to do a partial page update of that region when the name of the contact is clicked. Second, it uses the `<apex:param>` component to pass a parameter, in this case the `id` of the contact.

If you click any of the contacts, the page dynamically updates that contact, displaying its details, without refreshing the entire page.

Summary

Visualforce provides native support for Ajax partial page updates. The key is to identify a region, and then use the `re-render` attribute to ensure that the region is dynamically updated.

Learning More

There's a lot more to the Ajax and JavaScript support:

- `<apex:actionStatus>` lets you display the status of an Ajax request—displaying different values depending on whether it's in-progress or completed.
- `<apex:actionSupport>` lets you specify the user behavior that triggers an Ajax action for a component. Instead of waiting for an `<apex:commandLink>` component to be clicked, for example, the Ajax action can be triggered by a simple mouse rollover of a label.
- `<apex:actionPoller>` specifies a timer that sends an Ajax update request to Force.com according to a time interval that you specify.
- `<apex:actionFunction>` provides support for invoking controller action methods directly from JavaScript code using an Ajax request.
- `<apex:actionRegion>` demarcates the components processed by Force.com when generating an Ajax request.

Overriding and Pointing to Pages

Using Visualforce, you can override pretty much any aspect of the user interface, such as buttons, tabs, or links.

In this tutorial, you'll explore how to use Visualforce pages that you've created to replace standard Salesforce behavior.

Override the Standard Display for a Page

The Visualforce page you created in [Using Standard Controllers](#) can function as a replacement to the standard detail page for an account. You can modify the standard user interface generated by the platform to ensure that your page gets shown instead of the standard page.

1. From the object management settings for accounts, go to Buttons, Links, and Actions.
2. Click **Edit** next to the View item.
3. For **Override With**, select Visualforce Page.
4. From the Visualforce Page drop-down list, select `accountDisplay`.
5. Click **Save**.

The screenshot shows the 'Override Properties' dialog box. It has a title bar with 'Save' and 'Cancel' buttons. The main area contains the following fields:

- Label:** View
- Name:** View
- Default:** Standard Salesforce.com Page
- Override With:** Two radio buttons. The first is 'No Override (use default)'. The second is 'Visualforce Page', which is selected. To the right of the selected radio button is a dropdown menu showing 'accountDisplay [accountDisplay]'.
- Comment:** A large text area.

At the bottom right of the dialog, there are 'Save' and 'Cancel' buttons.

To see this in action, select the Accounts tab and then choose an account. Your page displays instead of the default. You've successfully configured the platform to automatically pass in that ID parameter to your page.

6. Follow the same procedure to reverse the override, so you can view the default page on the next lesson.

Embed a Page on a Standard Layout

Another way to get your page displayed is to embed it within a standard layout for another page. For example, imagine your `accountDisplay` showed an interesting analysis of the account data, and you wanted to embed it within the standard account detail view.


1. From the object management settings for accounts, go to Page Layouts.
2. Click **Edit** next to Account Layout.
3. Select Visualforce Pages in the left column of the user interface elements palette.
4. You'll notice your page appears here (because it uses the Accounts standard controller).
5. Select `accountDisplay`, and drag it to the Account Information panel.
6. Click **Save**.
7. To see this in action, select the Accounts tab and then choose an account. You'll notice the standard display of data, with your Visualforce page embedded within it! Your embedded page ideally needs to accommodate the inline display, so it might look a little plain right now, but notice how the embedded page automatically shows data of the same record—it's also being passed the ID parameter.

Create a Button that Links to a Visualforce Page

Pages like the standard account detail page have buttons, such as Edit and Delete. You can add a new button here that links to *your* page.

1. From the object management settings for accounts, go to Buttons, Links, and Actions.
2. Click **New Button or Link**.
3. Enter `MyButton` for the Label.
4. Enter `My_Button` for the Name.
5. For the Display Type, select `Detail Page Button`.
6. Select Visualforce Page in the Content Source picklist.
7. In the Content picklist that appears, select your `accountDisplay` page.
8. Click **Save**.

- Now that you have your button, you need to add it to a page layout. Repeat the process from [Embed a Page on a Standard Layout](#) but, instead of selecting a Visualforce page, add a button, and select **MyButton**.

 **Note:** Depending on your browser settings, you might get a privacy warning—allow your browser to load pages from the Visualforce domain to avoid these warnings.

You can use a similar procedure to create a link instead of a button, and you can add many buttons and links to standard and custom pages to create just the right navigation and user interface for your app.

Create Hyperlinks to URLs or Other Visualforce Pages

You might want to point from one Visualforce page to another, or to an external URL.

- Modify your Visualforce page to include the `<apex:outputLink>` component to produce a link:

```
<apex:outputLink value="http://developer.salesforce.com/">Click me!</apex:outputLink>
```

- To reference a page, use the expression `{! $Page.pagename}` to determine its URL.
- You can then include a link as follows:

```
<apex:outputLink value="{! $Page.AccountDisplay}">I am me!</apex:outputLink>
```

You can think of `$Page` as a global object that has a field for every page you've created.

Summary

Once you've created your Visualforce page, there are many ways to view it. You can just enter its URL, but you can also make it replace one of the automatically-generated pages, embed it within an existing page layout, or create buttons and hyperlinks that link to it.

Learning More

- Visualforce pages can also be viewed on public-facing web sites by using Force.com Sites. See the [Force.com Workbook](#) for an example.

- Sometimes you want to embed links to default actions, such as creating a new Account. Use the `<apex:outputLink>` component together with `URLFOR()` and the `$Action` global variable. For example:

```
<apex:outputLink value="{! URLFOR($Action.Account.new) }">Create</apex:outputLink>
```

Inputting Data with Forms

In this tutorial you learn how to create input screens using the standard controller, which provides record save and update functionality out of the box. This introduces you to the major input capabilities and their container—the `<apex:form>` component. Creating and Using Custom Controllers extends this and shows how to build forms that interact with your own controllers.

Create a Basic Form

The key to any data input is using a form. In this lesson you'll create the most basic form.

1. Create a new Visualforce page called `MyForm`, which uses a standard controller for the Account object.

```
<apex:page standardController="Account">
</apex:page>
```

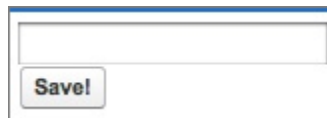
2. Insert an `<apex:form>` component, into which all your input fields will be placed:

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

3. Within the form, add an input field for the name field of an account, as well as command button that saves the form when clicked:

```
<apex:inputField value="{! account.name}"/>
<apex:commandButton action="{! save}" value="Save!"/>
```

This form, although very basic, contains all the essential elements: a form, an input field, and a button that processes the form:



In this case, you use a `<apex:commandButton>` which produces a button. Note that the action element is bound to `{! save}`. This expression language syntax looks similar to the syntax you used to specify a field in a record. However, in this context, it references a method—a piece of code named `save`. Every standard controller automatically supplies a `save()` method—as well as `update()` and other methods—so in this case, the `save()` method on the standard controller is invoked when the button is clicked.

If you enter a value and click **Save**, the values of the input fields are bound to like-named field values in a new record, and that record is inserted. In other words, the `<apex:inputField>` component produces an input field for the name field of a new account record, and when you click **Save**, ensures that the value you entered is automatically saved to that field.

After you click **Save**, the platform displays the newly-created record. Return to your Visualforce page by entering its URL, which will look something like `https://na6.salesforce.com/apex/MyForm`.

Show Field Labels

Visualforce does a lot of work behind the scenes, binding the input field to a field on a new record. It can do more, such as automatically showing the field label (much like `<apex:outputField>` in [Using Standard User Interface Components](#)), as well as automatically changing the input element to match the data type (for example, showing a picklist instead of an input box).

Modify the contents of the `<apex:form>` element so that it reads as follows:

```
<apex:form>
  <apex:pageBlock>
    <apex:pageBlockSection>
      <apex:inputField value="{!account.name}"/>
      <apex:inputField value="{!account.industry}"/>
      <apex:commandButton action="{!save}" value="Save!"/>
    </apex:pageBlockSection>
  </apex:pageBlock>
</apex:form>
```

By encapsulating the input fields within `<apex:pageBlock>` and `<apex:pageBlockSection>` components, Visualforce automatically inserts field labels ("Account Name", "Industry") as well as indicators of whether values are required for the fields, all using the platform styles.

 A screenshot of a Visualforce form. It features two input fields: 'Account Name' and 'Industry'. The 'Account Name' field is a text input with a red vertical bar on the left, indicating it is a required field. The 'Industry' field is a picklist with a dropdown arrow and the text '--None--'. Below the fields is a 'Save!' button.

Display Warning and Error Messages

The `<apex:pageMessages>` component displays all information, warning or error messages that were generated for all components on the current page. In the previous form, the account name was a required field. To ensure that a standard error message is displayed if someone tries to submit the form without supplying an account name, do the following:

1. Update your page by inserting the following line after the `<apex:pageBlock>` tag:

```
<apex:pageMessages/>
```

2. Now click **Save** on the form. An error panel will be displayed:

 A screenshot of the Visualforce form after an error. At the top, there is a yellow error panel with a red exclamation mark icon and the text 'Error: Account Name: You must enter a value'. Below this, the form is titled 'Account Details'. The 'Account Name' field now has a red border and a red error message 'Error: You must enter a value' below it. The 'Industry' field and the 'Save!' button remain the same.

Summary

Visualforce's standard controllers contain methods that make it easy to save and update records. By using the `<apex:form>` and `<apex:inputField>` components, you can easily bind input fields to new records using the standard controllers. The user interface automatically produces input components that match the type of the field—for example displaying a calendar input for a Date type.

field. The `<apex:pageMessages>` component can be used to group and display the information, warning and error messages across all components in the page.

Learning More

- You can use the `<apex:commandLink>` instead of the `<apex:commandButton>` component within a form to provide a link for form processing.
- Use the `quicksave()` method instead of the `save()` method to insert or update an existing record without redirecting the user to the new record.
- Use the `<apex:pageBlockButtons>` component to place command buttons when using the `<apex:pageBlock>` component.
- Use the `<apex:pageMessage>` component (the singular, not the plural) to create custom messages.

Reusing Pages with Templates

Many web sites have a design element that appears on every page, for example a banner or sidebar. You can duplicate this effect in Visualforce by creating a skeleton template that allows other Visualforce pages to implement different content within the same standard structure. Each page that uses the template can substitute different content for the placeholders within the template.

Create a Template

Templates are Visualforce pages containing special tags that designate placeholder text insertion positions. In this lesson you create a simple template page that uses the `<apex:insert>` component to specify the position of placeholder text.

1. Create a new Visualforce page called `BasicTemplate`.
2. Use the following as the body of the page:

```
<apex:page>
  <h1>My Fancy Site</h1>
  <apex:insert name="body"/>
</apex:page>
```

The key here is the `<apex:insert>` component. You won't visit this page (unless developing it) directly. Rather, create another Visualforce page that embeds this template, inserting different values for each of the `<apex:insert>` components. Note that each such component is named. In the above template, you have a single insert position named `body`. You can have dozens of such positions.

Use a Template with Another Page

You can now embed the template in a new page, filling in the blanks as you go.

1. Create a new Visualforce page called `MainPage`.
2. Within the page, add the following markup:

```
<apex:page sidebar="false" showHeader="false">
  <apex:composition template="BasicTemplate">
    <apex:define name="body">
      <p>This is a simple page demonstrating that this
        text is substituted, and that a banner is created.</p>
    </apex:define>
```

```
    </apex:composition>
</apex:page>
```

The `<apex:composition>` component fetches the Visualforce template page you created earlier, and the `<apex:define>` component fills the named holes in that template. You can create multiple pages that use the same component, and just vary the placeholder text.

Include One Visualforce Page within Another

Another way to include content from one page into another is to use the `<apex:include>` component. This lets you duplicate the entire contents of another page, without providing any opportunity to make any changes as you did with the templates.

1. Create a new Visualforce page called `EmbedsAnother`.
2. Use the following markup in the page:

```
<apex:page sidebar="false" showHeader="false">
  <p>Test Before</p>
  <apex:include pageName="MainPage"/>
  <p>Test After</p>
</apex:page>
```

Your original `MainPage` will be inserted verbatim.

Summary

Templates are a nice way to encapsulate page elements that need to be reused across several Visualforce pages. Visualforce pages just need to embed the template and define the content for the placeholders within the template. The `<apex:include>` component provides a simpler way of embedding one page within another.

INTRODUCTION TO APEX

Force.com Apex is a strongly-typed, object-oriented programming language that allows you to write code that executes on the Force.com platform. Out of the box, Force.com provides a lot of high-level services, such as Web services, scheduling of code execution, batch processing, triggers—and Visualforce back-end logic. All of these require you to write Apex.

In this section we'll start with some set up, and then write some very simple Apex to introduce you to the tools. Then we'll jump right into the deep end, and you'll learn enough Apex to create your first "real" Apex class. The code you'll add to your organization will provide custom functionality that you'll use in the following section, to create a mobile-aware Visualforce page that you deploy to Salesforce1.

This section does assume you know a little about programming. If you don't, you'll still be able to complete the exercises, but you might not understand every aspect. (And that's OK!)

When you're finished with this section, you will have done the following.

- Open the Developer Console, the advanced development tool for Force.com, and use it to create, edit, and run Apex code.
- Execute Apex code snippets in the Execute Anonymous Apex window. (You'll even know what "anonymous Apex" means!)
- Create Apex classes and methods.
- Know some of the similarities and differences between Apex and other programming languages, such as Java, C#, and PHP.
- Execute a SOQL query in Apex, and process the results of that query.
- Create and run tests that verify the correct behavior of your Apex code, and understand what code coverage is and how to check it.

Set Up Your Development Environment

In this short lesson, you'll prepare your DE org for the exercises that follow. You'll install a package with some supplementary resources, load the Salesforce1 browser testing environment, and install the Salesforce1 mobile app on your mobile device of choice.

Install the Enhanced Warehouse Data Model

To prepare your developer organization for the exercises in this and the following section, you need to import the Warehouse data model and sample data.

You might be familiar with the Warehouse app if you've gone through tutorials in other workbooks, or at a hands-on workshop. The Warehouse app used here is an enhanced version that includes additional custom objects and data, and some supporting code.

1. In your browser go to http://bit.ly/warehouse_schema11
2. If you're already logged in, you're redirected to the Package Installation Details page. Otherwise, log in with your Developer Edition credentials.
3. Click **Continue**, **Next**, **Next**, and **Install**.
4. After the installation finishes, click the Force.com app menu and select **Warehouse**.
5. Click the **Data** tab and then click the **Create Data** button.

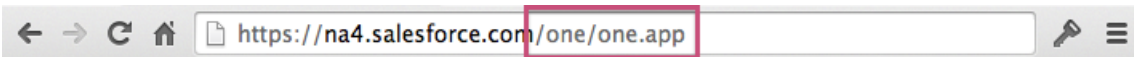
The package contains a pre-built Visualforce page, as well as some supporting resources. You'll learn about them right after your development and testing environments are set up.

Access the Mobile Browser Web App

When developing Visualforce pages for the Salesforce1 mobile app, you don't want to use the familiar `https://<instance>/apex/<pageName>` URL to view the page: you want to see how the pages look in the mobile app.

The best way to test your pages is with the actual mobile app, because it provides the most realistic experience. However, since it's a pain to grab your phone every time you want to see a change, you can open a new browser tab and use the `one.app` mobile browser version.

1. In your browser, open a new tab.
2. Copy and paste your Salesforce instance home URL into the address bar of the new tab, and add `/one/one.app` to the end. For example, if your Salesforce instance has an URL of `https://na4.salesforce.com`, use `https://na4.salesforce.com/one/one.app`.



3. Press Return to load the edited URL.

You should now see the mobile browser version of Salesforce1. As you go through the exercises in this workbook, you can develop in one tab and then test in the other!

Important: The `/one/one.app` version is great for development, but you should always test on the actual devices and browsers that you intend to support.

Download the Salesforce1 Mobile App

For final testing of the app you're about to build, you'll need to install the Salesforce1 mobile app on your device.

If you've already downloaded the Salesforce1 mobile app, you can skip this step.

1. Use your mobile device's browser to go to www.salesforce.com/mobile, select the appropriate platform, and download Salesforce1.
2. Open Salesforce1 from your mobile device.
3. Enter your Salesforce credentials and tap **Log in to Salesforce**.
4. If you're prompted to allow access to your data, tap **OK** and continue.

If you haven't already explored Salesforce1, now is a great time to check it out. Being familiar with its functionality will help you create apps that work well inside it.

Using the Developer Console

The Developer Console lets you execute Apex code statements. It also lets you execute Apex methods within an Apex class or object. In this tutorial you open the Developer Console, execute some basic Apex statements, and toggle a few log settings.

Activating the Developer Console

After logging into your Salesforce environment, the screen displays the current application you're using (in the diagram below, it's **Warehouse**), as well as your name.

Open the Developer Console under **Your Name** or the quick access menu (⚙️).

You can open the Developer Console at any time.

SEE ALSO:

[Salesforce Help: Open the Developer Console](#)

Using the Developer Console to Execute Apex Code

The Developer Console can look overwhelming, but it's just a collection of tools that help you work with code. In this lesson, you'll execute Apex code and view the results in the Log Inspector. The Log Inspector is a useful tool you'll use often.

1. Click **Debug > Open Execute Anonymous Window** or CTRL+E.
2. In the Enter Apex Code window, enter the following text: `System.debug('Hello World');`



Note: `System.debug()` is like using `System.out.println()` in Java (or `printf()` if you've been around a while :-). But, when you're coding in the cloud, where does the output go? Read on!

3. Deselect **Open Log** and then click **Execute**.



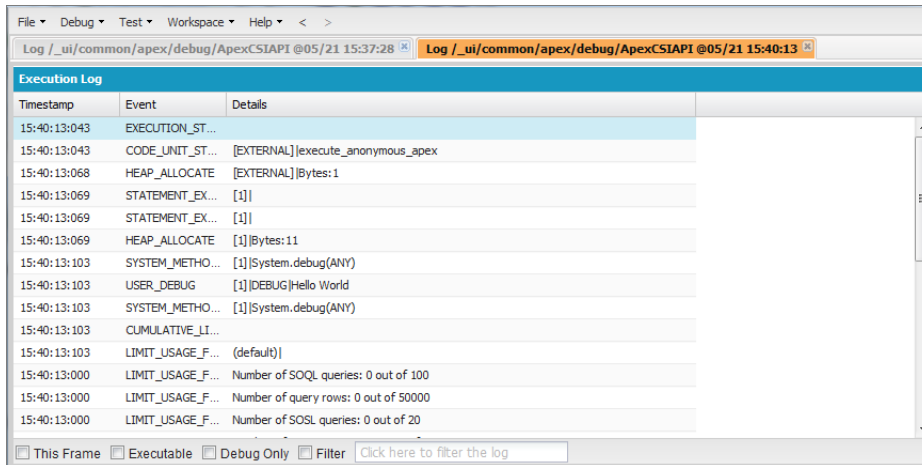
Every time you execute code, a log is created and listed in the *Logs* panel.

Logs						
User	Application	Operation	Time	Status	Read	Size
Admin User	Browser	/_ui/common/apex/...	05/21 15:37:28	Success		39612
Admin User	Browser	/_ui/common/apex/...	05/21 15:40:13	Success		1510

Double-click a log to open it in the Log Inspector. You can open multiple logs at a time to compare results.

Log Inspector is a context-sensitive execution viewer that shows the source of an operation, what triggered the operation, and what occurred afterward. Use this tool to inspect debug logs that include database events, Apex processing, workflow, and validation logic.

The Log Inspector includes predefined perspectives for specific uses. Click **Debug > Switch Perspective** to select a different view, or click CTRL+P to select individual panels. You'll probably use the Execution Log panel the most. It displays the stream of events that occur when code executes. Even a single statement generates a lot of events. The Log Inspector captures many event types: method entry and exit, database and web service interactions, and resource limits. The event type `USER_DEBUG` indicates the execution of a `System.debug()` statement.



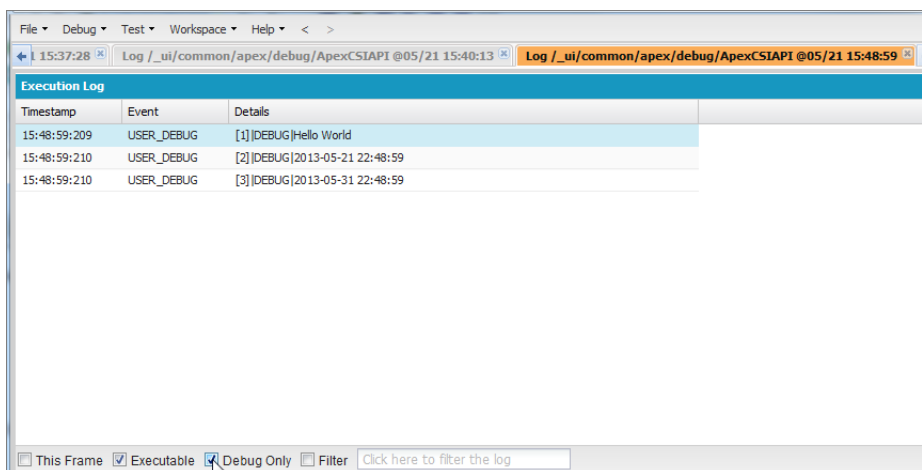
1. Click **Debug > Open Execute Anonymous Window** or CTRL+E and enter the following code:

```
System.debug( 'Hello World' );
System.debug( System.now() );
System.debug( System.now() + 10 );
```

2. Select **Open Log** and click **Execute**.
3. In the Execution Log panel, select **Executable**. This limits the display to only those items that represent executed statements. For example, it filters out the cumulative limits.
4. To filter the list to show only **USER_DEBUG** events, select **Debug Only** or enter **USER** in the **Filter** field.



Note: The filter text is case sensitive.



Congratulations—you have successfully executed code on the Force.com platform and viewed the results!

Tell Me More...

Help in the Developer Console

To learn more about the Developer Console, click **Help > Help Docs...** in the Developer Console.

Anonymous Blocks

The Developer Console allows you to execute code statements on the fly. You can quickly evaluate the results in the **Logs** panel.

The code that you execute in the Developer Console is referred to as an *anonymous block*. Anonymous blocks run as the current user and can fail to compile if the code violates the user's object- and field-level permissions. Note that this isn't the case for Apex classes and triggers.

Summary

To execute Apex code and view the results of the execution, use the Developer Console. The detailed execution results include not only the output generated by the code, but also events that occur along the execution path. Such events include the results of calling another piece of code and interactions with the database.

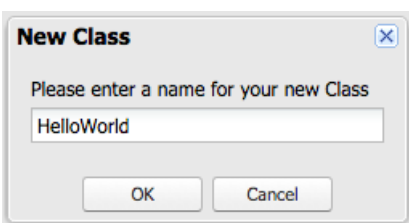
Creating and Instantiating Classes

Apex is an object-oriented programming language, and much of the Apex you write will be contained in classes, sometimes referred to as blueprints or templates for objects. In this tutorial you'll create a simple class with two methods, and then execute them from the Developer Console.

Creating an Apex Class Using the Developer Console

To create an Apex class in the Developer Console:

1. Open the Developer Console.
2. Click **File > New > Apex Class**.
3. Enter `HelloWorld` for the name of the new class and click **OK**.



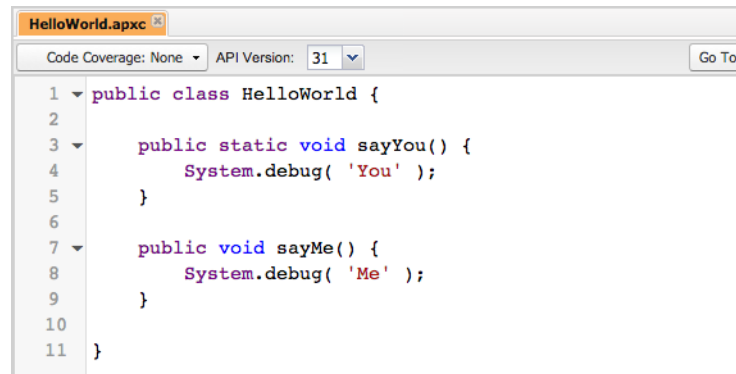
4. A new empty `HelloWorld` class is created. Add a static method to the class by adding the following text between the braces:

```
public static void sayYou() {  
    System.debug( 'You' );  
}
```

5. Add an instance method by adding the following text just before the final closing brace:

```
public void sayMe() {  
    System.debug( 'Me' );  
}
```

- Click **File** > **Save**.



```
1 public class HelloWorld {
2
3     public static void sayYou() {
4         System.debug( 'You' );
5     }
6
7     public void sayMe() {
8         System.debug( 'Me' );
9     }
10
11 }
```

Tell Me More...

- You've created a class called `HelloWorld` with a static method `sayYou()` and an instance method `sayMe()`. Looking at the definition of the methods, you'll see that they call another class, `System`, invoking the method `debug()` on that class, which will output strings.
- If you invoke the `sayYou()` method of *your* class, it invokes the `debug()` method of the `System` class, and you see the output.
- The Developer Console validates your code in the background to ensure that the code is syntactically correct and compiles successfully. Making mistakes, such as typos in your code, is inevitable. If you make a mistake in your code, errors appear in the Problems pane and an exclamation mark is added next to the pane heading: **Problems!**
- Expand the Problems panel to see a list of errors. Clicking on an error takes you to the line of code where this error is found. For example, the following shows the error that appears after you omit the closing parenthesis at the end of the `System.debug` statement.



```
1 public class HelloWorld {
2
3     public static void sayYou() {
4         System.debug( 'You';
5     }
6
7     public void sayMe() {
8         System.debug( 'Me' );
9     }
10
11 }
```

Name	Line	Problem
HelloWorld	4	expecting a right parentheses, found ';'

Re-add the closing parenthesis and notice that the error goes away.

SEE ALSO:

[Salesforce Help: Open the Developer Console](#)

Calling a Class Method

Now that you've created the `HelloWorld` class, follow these steps to call its methods.

1. Execute the following code in the Developer Console Execute Anonymous Window to call the `HelloWorld` class's static method. (See [Activating the Developer Console](#) if you've forgotten how to do this.) If there is any existing code in the entry panel, delete it first. Notice that to call a static method, you don't have to create an instance of the class.

```
HelloWorld.sayYou();
```

2. Open the resulting log.
3. Set the filters to show `USER_DEBUG` events. (Also covered in [Activating the Developer Console](#)). "You" appears in the log:

Execution Log		
Timestamp	Event	Details
23:03:48:156	USER_DEBUG	[4]DEBUGYou

4. Now execute the following code to call the `HelloWorld` class's instance method. Notice that to call an instance method, you first have to create an instance of the `HelloWorld` class.

```
HelloWorld hw = new HelloWorld();
hw.sayMe();
```

5. Open the resulting log and set the filters.

"Me" appears in the Details column. This code creates an instance of the `HelloWorld` class, and assigns it to a variable called `hw`. It then calls the `sayMe()` method on that instance.
6. Clear the filters on both logs, and compare the two execution logs. The most obvious differences are related to creating the `HelloWorld` instance and assigning it to the variable `hw`. Do you see any other differences?

Congratulations—you have now successfully created and executed new code on the Force.com platform!

Creating an Apex Class Using the Salesforce User Interface

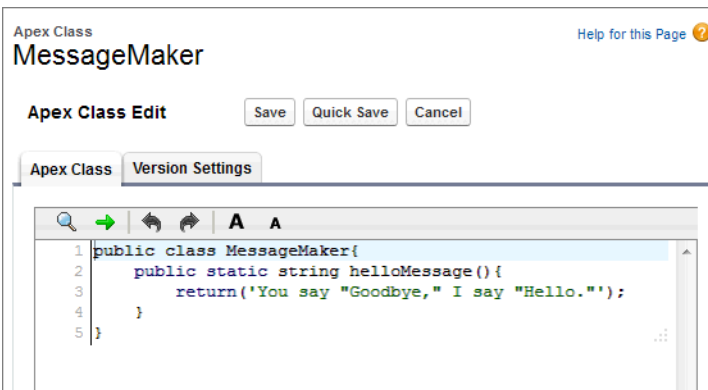
You can also create an Apex class in the Salesforce user interface.

1. From Setup, enter *Apex Classes* in the Quick Find box, then select **Apex Classes**.
2. Click **New**.
3. In the editor pane, enter the following code:

```
public class MessageMaker {
}
```

- Click **Quick Save**. You could have clicked **Save** instead, but that closes the class editor and returns you to the Apex Classes list. **Quick Save** saves the Apex code, making it available to be executed, but lets you continue editing—making it easier to add to and modify the code.
- Add the following code to the class:

```
public static string helloMessage() {
    return('You say "Goodbye," I say "Hello"');
}
```



- Click **Save**.

You can also view the class you've just created in the Developer Console and edit it.

- In the Developer Console, click **File > Open**.
- In the Entity Type panel, click **Classes**, and then double-click **MessageMaker** from the **Entities** panel.

The `MessageMaker` class displays in the source code editor. You can edit the code there by typing directly in the editor and saving the class.

Summary

In this tutorial you learned how to create and list Apex classes. The classes and methods you create can be called from the Developer Console, as well as from other classes and code that you write.

Tell Me More...

- Alternatively, you can use the [Force.com IDE](https://developer.salesforce.com/) to create and execute Apex code. For more information, search for "Force.com IDE" on the Developer Force site: <https://developer.salesforce.com/>.

Creating the `warehouseUtils` Class

In this exercise, we turn to real work. You'll write a new Apex class that searches Salesforce to find records that match a query, and makes those records available for use on a Visualforce page.

Here's the scenario. You're going to write a small app to give mobile technicians that work for the Acme Wireless organization a way to find nearby warehouses. For example, if the technician is out on a call and needs a part, they can use this page to look for warehouses within a 20-mile radius. For each warehouse, a map should display a pin along with the warehouse name, address, and phone number.

The Apex and Visualforce code that you're about to write will do all of that inside Salesforce1 on a mobile device. It's going to be cool.

Create the `WarehouseUtils` Apex Class

First you need to define the new class and give it a constructor method.

Depending on where an Apex class is going to be used, you might need to conform to expected interfaces or conventions. For example, the `WarehouseUtils` class could be used two ways: as a *Visualforce controller extension* from a Visualforce page, and as a *Remote Action* from Visualforce JavaScript remoting.

A controller extension is used to *extend* the capabilities of a Visualforce controller, by adding additional functionality in the form of methods that can be called by the page. A Visualforce page can have only one controller, but can have one, none, or many controller extensions.

To be a controller extension, an Apex class needs to have a constructor that accepts a Visualforce controller as its only parameter. (We'll look at the requirements for Remote Actions later.)

1. From Setup, enter `Apex Classes` in the Quick Find box, then select **Apex Classes**, and then click **New**.
2. In the Editor enter the following code.

```
global with sharing class WarehouseUtils {  
  
    public WarehouseUtils(ApexPages.StandardSetController controller) { }  
  
    // findNearbyWarehouses method goes here  
  
}
```

3. Click **Quick Save**.

The constructor method takes a `ApexPages.StandardSetController` object as its only parameter. This allows the class to be used as a Visualforce controller extension with a Standard List Controller. To also work with a Standard Controller, *overload* the constructor to take a different parameter type. That is, add a second constructor method that takes an `ApexPages.StandardController` parameter.

```
public WarehouseUtils(ApexPages.StandardController controller) { }
```

These constructors are empty, but in a more complex controller extension you would save the controller as an instance variable. Do you think you know enough Apex by now to do that? Give it a try!

Add a "Stub" `findNearbyWarehouses` Method

Next, stub in the method that will be used by the Visualforce page.

Each public and global method in a controller extension is available to be used by an associated Visualforce page. To call the method the page can reference it in an expression, or it can call the method directly using JavaScript remoting.

In this case, we want to create a method that will query for warehouses located near the mobile technician who is using the app. This means the method needs to know where the technician is located, so we'll pass in latitude and longitude values that the Salesforce1 app can provide using the built-in geolocation capabilities of the device it's running on. Visualforce expressions can't take parameters directly, so we're planning to use JavaScript remoting. For now, we'll just write a method "stub" that takes latitude and longitude parameters, and returns a list of warehouse records.

1. In your code editor, replace the comment line `// findNearbyWarehouses method goes here` with the following code.

```
// Find warehouses nearest a geolocation
@RemoteAction
global static List<Warehouse__c> findNearbyWarehouses(String lat, String lon) {

    // Initialize results to an empty list
    List<Warehouse__c> results = new List<Warehouse__c>();

    // method implementation goes here

    // Return the query results
    return(results);
}
```

2. Click **Quick Save**.

Although it doesn't do anything yet, this method definition illustrates the essentials of an Apex method.

- `global`: The scope of the method. Methods to be called by JavaScript remoting, called *Remote Actions*, must be either `global` or `public`.
- `static`: This is a class method, as opposed to an instance method. This means you can call the method without instantiating an object of this class. Remote Action methods must be static.
- `List<Warehouse__c>`: The data type of the method's return value.
- `findNearbyWarehouses`: The name of the method.
- `(String lat, String lon)`: The method's parameters.

The remainder of the method, between the braces—what there is so far!—is the implementation. You'll write that next!

Perform a Query and Return the Results

Now it's time to write the actual method implementation, which will take the latitude and longitude values provided by the user's device and find nearby warehouses.

To search for the relevant records, you need to convert the provided parameters into a complete *SOQL query*. SOQL is the primary query language of the Force.com platform. You can use it in Apex, as we'll do here, but you can use it with other Salesforce APIs as well.

We'll construct the query dynamically, using *string concatenation* to combine the necessary SOQL elements with the parameter values. Then we'll execute the query, and return the results.

1. Inside the method implementation block, replace the comment line `// method implementation goes here` with the following code.

```
// SOQL query to get the nearest warehouses
String queryString =
    'SELECT Id, Name, Location__Longitude__s, Location__Latitude__s, ' +
    'Street_Address__c, Phone__c, City__c ' +
    'FROM Warehouse__c ' +
    'WHERE DISTANCE(Location__c, GEOLOCATION('+lat+', '+lon+'), \'mi\') < 20 ' +
    'ORDER BY DISTANCE(Location__c, GEOLOCATION('+lat+', '+lon+'), \'mi\') ' +
    'LIMIT 10';
```



```
// Run the query
results = database.Query(queryString);
```

2. Click **Quick Save**.

SOQL looks a lot like standard SQL, and if you know SQL already, you'll pick up SOQL easily. See the [Force.com SOQL and SOSL Reference](#) for comprehensive details of the query language.

Two aspects of the query might not be immediately obvious.

- The `GEOLOCATION()` function creates a *geolocation* from a latitude and longitude. A geolocation represents a specific physical location. Here the function is used to combine the latitude and longitude parameters to create a value that represents the location of the user.
- The `DISTANCE()` function calculates the distance between two geolocations. Here it's calculating the distance between the `Warehouse__c.Location__c` geolocation field and the geolocation of the user. The query's `WHERE` clause is looking for `DISTANCE()` values within 20 miles.

Summary and Code Check

You did it! You wrote a new Apex utility class that you'll be able to use with a Visualforce page.

Your completed class should look like the following.

```
global with sharing class WarehouseUtils {

    public WarehouseUtils(ApexPages.StandardSetController controller) { }

    // Find warehouses nearest a geolocation
    @RemoteAction
    global static List<Warehouse__c> findNearbyWarehouses(String lat, String lon) {

        // Initialize results to an empty list
        List<Warehouse__c> results = new List<Warehouse__c>();

        // SOQL query to get the nearest warehouses
        String queryString =
            'SELECT Id, Name, Location__Longitude__s, Location__Latitude__s, ' +
            'Street_Address__c, Phone__c, City__c ' +
            'FROM Warehouse__c ' +
            'WHERE DISTANCE(Location__c, GEOLOCATION(' + lat + ', ' + lon + '), \'mi\') < 20 ' +
            'ORDER BY DISTANCE(Location__c, GEOLOCATION(' + lat + ', ' + lon + '), \'mi\') ' +
            'LIMIT 10';

        // Run the query
        results = database.Query(queryString);

        // Return the query results
        return(results);
    }
}
```

As exciting as it would be to race ahead and use this new code to make a cool Visualforce page, well...there's a bug in the code. (Have you spotted it already?) So, before we go further, you'll want to learn a bit about testing and debugging Apex code.

Testing and Debugging the `WarehouseUtils` Class

In this exercise, we need to take our new Apex class and verify that it functions as intended. Using the Apex unit testing framework, you'll write tests and debug your new code.

Writing unit tests for your code is fundamental to developing Apex code. You must have 75% test coverage to be able to deploy your Apex code to your production organization. In addition, the tests counted as part of the test coverage must pass. Testing is key to ensuring the quality of your application. Furthermore, having a set of tests that you can rerun in the future if you have to make changes to your code allows you to catch any potential regressions to the existing code.

Testing might seem like an obstacle to getting to the "fun" part of your project. But when you see how easy it is, perhaps you'll change your mind.

Create an Apex Test Class

Unit tests are contained in Apex classes and, with a few small additions, look just like regular Apex classes.

Test classes use annotations that mark them as test classes. Test classes don't count against your organization's code size limits.

1. In the Developer Console, click **File > New > Apex Class**.
2. For the class name, enter `TestWarehouseUtils` and click **OK**.
3. In the editor, delete the auto-generated code and replace it with the following.

```
@isTest
private class TestWarehouseUtils {

    // test methods go here

}
```

The `@isTest` annotation tells Force.com that all of the code within the Apex class is test code. It's a best practice to keep your test code `private`. The Apex test framework can find and run your tests, but nothing else should be able to.



Note: If you create test helper or utility classes that are used by separate test classes, they'll need to be `public`.

Add a Test Method and Setup Code

Define test methods within your test class to add them to your organization's test suite.

The `WarehouseUtils` class has only one method, but there are a few things we'd like to test it for. Calling the method should return all warehouses located within 20 miles of a specific location. The method should also return none of the warehouses that are outside of that 20 mile radius. Finally, locations that appear nearby one location should no longer be nearby if the requested location changes to being far away. We'll create two test methods to cover these expectations.

In order to test these expectations, we'll need a few test warehouses with known distances from our test location. The Apex test framework makes it easy to create tests that use test data, and *only* test data, during the course of the test execution. This is called *test isolation*. Your organization's data is hidden from tests by default, and your test data and any changes to data that the tests perform are all rolled back at the end of test execution. But since we're not testing against the data in your organization, our tests will have to create their own data. We'll create a few helper methods to handle that, too.

1. In the test class you created in the last step, we'll add two new test method stubs and a few helper methods. Inside the class definition block, replace the comment line `// test methods go here` with the following code.

```
// test that we find only warehouses that are within 20 miles
static testMethod void testFindWarehousesWithinTwentyMiles() {

    // test for when close to warehouses here

}

// test that we don't find anything when further than 20 miles
static testMethod void testDontFindWarehousesFurtherThanTwentyMiles() {

    // test for when far from warehouses here

}

// helper methods to create test data
static Warehouse__c createTestWarehouse(String name, Decimal lat, Decimal lon) {
    Warehouse__c w = new Warehouse__c(
        Name = name,
        Location__Latitude__s = lat,
        Location__Longitude__s = lon
    );
    insert w;
    return w;
}

static Warehouse__c createClosestTestWarehouse() {
    // Federal Reserve Bank of SF, next door to Salesforce HQ
    return(createTestWarehouse('Warehouse1', 37.7927731, -122.4010922));
}

static Warehouse__c createCloseTestWarehouse() {
    // Moscone Center, home of Dreamforce
    return(createTestWarehouse('Warehouse2', 37.783944, -122.401289));
}

static Warehouse__c createTooFarTestWarehouse() {
    // Mount Rushmore, South Dakota
    return(createTestWarehouse('TooFarWarehouse', 43.879102, -103.459067));
}
```

The test method definitions are `static testMethod void testName()`, with no parameters. Right now this test class doesn't test anything—we still need to fill in the actual test code.

The helper methods don't have `testMethod` in their definition, and can take parameters and return values. Other than being `static`, they can be any kind of method you need them to be. Here their only function is to create new, pre-defined warehouse objects and save them into the database.

Your tests and helpers can insert, change, and delete records as much as you need them to fully exercise your code. Remember, all database interaction takes place in an isolated, test-only environment. No changes performed by the tests will be saved permanently.

Test the `findNearbyWarehouses` Method

Write the least amount of code possible that will exercise your code and test its behavior. Test one thing at a time.

You have two test method implementations to write. One will test calling `WarehouseUtils.findNearbyWarehouses` from a location that is close by some warehouses, and one will test calling `WarehouseUtils.findNearbyWarehouses` from a location that's not near any warehouses.

1. Inside the `testFindWarehousesWithinTwentyMiles` method replace the comment line `// test for when close to warehouses` here with the following code.

```
// Salesforce HQ
String myLat = '37.793731';
String myLon = '-122.395002';

// Create test warehouse data
Warehouse__c closestWarehouse = createClosestTestWarehouse();
Warehouse__c closeWarehouse = createCloseTestWarehouse();
Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

// Perform the test execution
Test.startTest();
List<Warehouse__c> nearbyWarehouses =
    WarehouseUtils.findNearbyWarehouses(myLat, myLon);
Test.stopTest();

// Make assertions about expected results

// We expect two warehouses
System.assert(nearbyWarehouses.size() == 2);

// We expect two SPECIFIC warehouses, in order of proximity
System.assert(nearbyWarehouses[0].Name == closestWarehouse.Name);
System.assert(nearbyWarehouses[1].Name == closeWarehouse.Name);

// We do NOT expect to see the warehouse that's too far away
if(0 < nearbyWarehouses.size()) {
    for (Warehouse__c wh : nearbyWarehouses) {
        System.assert(wh.Name != tooFarWarehouse.Name);
    }
}
```

2. Inside the `testDontFindWarehousesFurtherThanTwentyMiles` method replace the comment line `// test for when far from warehouses` here with the following code.

```
// Eiffel Tower, Paris, France
String myLat = '48.85837';
String myLon = '2.294481';

// Create test warehouse data
Warehouse__c closestWarehouse = createClosestTestWarehouse();
Warehouse__c closeWarehouse = createCloseTestWarehouse();
Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

// Perform the test execution
```

```
Test.startTest();
List<Warehouse__c> nearbyWarehouses =
    WarehouseUtils.findNearbyWarehouses(myLat, myLon);
Test.stopTest();

// We expect to see NO warehouses
System.assert(nearbyWarehouses.size() == 0);
```

The test methods follow a simple pattern.

- Perform required setup, including creation of test data.
- Perform the test execution, wrapped inside `Test.startTest()` and `Test.endTest()` test framework calls.
- Compare the results of the test execution with known data. That is, compare actual behavior to expected behavior.

This is a good pattern to follow in your own test code. It's also a best practice to test only one thing at a time, and to put each test into a separate method.

Run the Test and Review Test Results

The Force.com test framework makes it easy to run your tests and provides test run results and test coverage analysis for your tested code.

Running a good test suite and getting clean results is one of the most satisfying things you can do as a programmer. Let's get that warm fuzzy feeling right now.

1. In the Developer Console, click **Test > New Run**.
2. Click **TestWarehouseUtils**.
3. To add all methods in the `TestWarehouseUtils` class to the test run, click **Add Selected**.
4. Click **Run** to execute the test run.

The test result displays in the **Tests** tab. You can expand the test run folder and then expand the test class in the **Tests** tab to see which methods were run. In this case, the class contains two test methods.

5. The Overall Code Coverage pane shows the code coverage of this test class, which is 83%. The output you see is similar to the following. The code coverage for the `WarehouseUtils` class is outlined.

Logs	Tests	Checkpoints	Query Editor	View State	Progress	Problems			
Status	Test Run	Duration	Failures	Total	Overall Code Coverage				>>
✓	707D00000005c44		0	2	Class	Percent	Lines		
✓	707D00000005c49		0	2	RemoteTKController	0%	0/162		
✓	707D00000005c4E		0	2	SfdcLocale	0%	0/44		
✓	TestWarehouseUtils		0	2	WarehouseEditor	0%	0/13		
✓	testDontFindWarehousesFurtherThanTwentyMiles	0:01			WarehouseLocator	0%	0/5		
✓	testFindWarehousesWithinTwentyMiles	0:00			WarehouseUtils	83%	5/6		
					WSResult	0%	0/27		

The result tells you a number of important things.

- It indicates whether your tests passed. If the Boolean condition in the `System.assert` statements in the tests had failed—that is, if the assertion were `false`—then that failure would be flagged here. Adding lots of assertions is a great way to verify the expected behavior of your code.
- It provides detail about the execution of the test. By looking through the associated debug log in the **Logs** tab, you see which methods executed, which records were created or modified, how many queries were executed, and so on.

- It indicates code coverage by percentage and by how many lines of code were executed in each affected class.

The results page shows that we achieved 83% coverage of the `WarehouseUtils` class. That's enough code coverage to deploy, but why not aim for perfection? Let's see what's being missed.

6. In the Overall Code Coverage pane, double-click the line for the `WarehouseUtils` class coverage.

The Code Coverage page opens. Blue highlighting indicates lines of code that were covered (executed) during the test execution. Lines with red highlighting indicates lines of code that weren't executed.

```

1  global with sharing class WarehouseUtils {
2
3  public WarehouseUtils(ApexPages.StandardSetController controller) { }
4
5  @RemoteAction
6  // Find warehouses nearest a geolocation
7  global static List<Warehouse__c> findNearbyWarehouses(String lat, String lon) {
8
9      List<Warehouse__c> results = new List<Warehouse__c>();
10
11     // SOQL query to get the nearest warehouses
12     String queryString =
13         'SELECT Id, Name, Location__Longitude__s, Location__Latitude__s, ' +
14         'Street_Address__c, Phone__c, City__c ' +
15         'FROM Warehouse__c ' +
16         'WHERE DISTANCE(Location__c, GEOLOCATION('+lat+', '+lon+', \'mi\')) < 20 ' +
17         'ORDER BY DISTANCE(Location__c, GEOLOCATION('+lat+', '+lon+', \'mi\')) ' +
18         'LIMIT 10';
19
20     // Run the query
21     results = database.Query(queryString);
22
23     // Return the query results
24     return(results);
25 }
26 }

```

In this case, line 3—our empty constructor method—wasn't executed because we only called static methods, and so never instantiated the class. While an empty constructor is no big deal, getting coverage on it is also no big deal.

7. Add the following new test method to your test class.

```


// test the class constructor
static testMethod void testClassConstructor() {
    Test.startTest();
    WarehouseUtils utils = new WarehouseUtils(null);
    Test.stopTest();

    // We expect that utils is not null
    System.assert(utils != null);
}

```

If you rerun your tests, you should have 100% code coverage. Woo-hoo!

Code coverage refers to how much of your production code (in this case, the `WarehouseUtils` class) is covered by your test code (the test class you just wrote). In other words, when you run your test code, does it execute all, or only some portion of, your production code? If it only executes a portion of the code, that could mean your production code still has bugs in the untested portions. The code coverage view makes that easy to visualize.

 **Note:** Some of the code isn't highlighted either blue or red. What does that mean? For example, the class declaration, the `@RemoteAction` annotation, and comments aren't highlighted, which makes some sense, but neither are the additional lines of the `queryString` expression. What's up with that?

All of these lines are considered non-executable by the compiler, which is doing the work of highlighting. When you break a line of code across multiple lines in your editor, only the first line is highlighted, either way.

Find the Bug

A completely passing test suite doesn't always mean there aren't any bugs. It might just mean you haven't found them. Yet.

Yes, there's a bug in the version of `WarehouseUtils` we currently have. Technically, since it's *your* DE org, it's *your* bug, but we'll admit to leading you a bit astray. Let's find and fix it together.

You might have already figured it out, but just in case, here's a hint. What happens if we call `WarehouseUtils.findNearbyWarehouses` with invalid values for the latitude or longitude?

Let's try it and see. We can quickly run a short snippet of Apex code in the Execute Anonymous window, and see what happens.

1. In the Developer Console, click **Debug > Open Execute Anonymous Window**.
2. Add the following code, and then click **Execute**.

```
List<Warehouse__c> warehouses = WarehouseUtils.findNearbyWarehouses(null, null);
for(Warehouse__c wh : warehouses) {
    System.debug(wh.Name);
}
```

The result is an error in the Execute Anonymous window, "System.QueryException: unexpected token: 'null'". It turns out the `GEOLOCATION` function doesn't like invalid latitude or longitude values. Who knew?

The good news is, this gives us a chance to fix the bug like pros: by writing the test first.

Write a Test for the Bug

Test-first development is the practice of writing tests for a feature *before* you write the code to implement the feature.

Because the feature isn't implemented yet, the tests will fail. Then you implement the feature and run the tests again. When they pass, you have some confidence that you've implemented the feature correctly. Repeating this cycle as you develop new features increases your confidence in the software implementation.

1. In the `TestWarehouseUtils` test class, add the following new test method.

```
// test that we use a default location if the lat or long is invalid
static testMethod void testFindWarehousesDefaultLocation() {
    // Trigger the default location, which should be SF
    String myLat = null;
    String myLon = null;

    // Create test warehouse data
    Warehouse__c closestWarehouse = createClosestTestWarehouse();
    Warehouse__c closeWarehouse = createCloseTestWarehouse();
    Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

    // Perform the test execution
    Test.startTest();
```

```

List<Warehouse__c> nearbyWarehouses =
    WarehouseUtils.findNearbyWarehouses(myLat, myLon);
Test.stopTest();

// Make assertions about expected results

// We expect two warehouses
System.assert(nearbyWarehouses.size() == 2);

// We expect two SPECIFIC warehouses, in order of proximity
System.assert(nearbyWarehouses[0].Name == closestWarehouse.Name);
System.assert(nearbyWarehouses[1].Name == closeWarehouse.Name);

// We do NOT expect to see the warehouse that's too far away
if(0 < nearbyWarehouses.size()) {
    for (Warehouse__c wh : nearbyWarehouses) {
        System.assert(wh.Name != tooFarWarehouse.Name);
    }
}
}

```

2. Save the updated test class, and re-run your tests.

The result of re-running the test suite with the new test should be a failure. This means the test *is* working, and detecting that your production code is *not* working as intended.

Fix the Bug

Once you have a test that verifies and isolates incorrect behavior in your code, it's often straightforward to fix the issue. The reward for doing so is a passing test suite.

We know that the `findNearbyWarehouses` method fails with an error when it's called with blank latitude or longitude values. Checking for missing or empty values is pretty easy to do.

1. In the `WarehouseUtils` class, after `results` is initialized to an empty list and before the query string is assembled, add the following code.

```

// If geolocation parameters are invalid, use San Francisco
if(String.isBlank(lat) || String.isBlank(lon)) {
    lat = '37.793731';
    lon = '-122.395002';
}

```

2. Save your changes, and re-run your test suite.

And, that should do it. Once again, you have a passing test suite.

Before we leave the topic of Apex and testing, take a look at the new code you've just added. Will that `if` condition catch all possible invalid latitude and longitude values? What could you add? Should that all go into the `if` condition? Do you feel ready to add a helper method to the class that could check `lat` and `lon` for validity? How would you do that? Should the helper method be `public` or `private`?

Here's another stretch exercise. There are now a few hard-coded latitude and longitude values in both `WarehouseUtils` and `TestWarehouseUtils`. What assures you that those numbers will stay in sync? What happens if they get out of sync? Think about those assertions. What happens if a typo in a latitude or longitude value causes the test location to "drift" away from one test warehouse and closer to the other?

Summary and Code Check

You just finished writing a test suite for your Apex class. You also learned how to create test runs that execute your tests, and how to check the code coverage of your test suite.

Having a complete set of tests to verify correct behavior of your code is necessary for deployment and it's also the key to successful long-term development.

Your complete test class should look like this.

```
@isTest
private class TestWarehouseUtils {

    // test that we find only warehouses that are within 20 miles
    static testMethod void testFindWarehousesWithinTwentyMiles() {
        // Salesforce HQ
        String myLat = '37.793731';
        String myLon = '-122.395002';

        // Create test warehouse data
        Warehouse__c closestWarehouse = createClosestTestWarehouse();
        Warehouse__c closeWarehouse = createCloseTestWarehouse();
        Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

        // Perform the test execution
        Test.startTest();
        List<Warehouse__c> nearbyWarehouses =
            WarehouseUtils.findNearbyWarehouses(myLat, myLon);
        Test.stopTest();

        // Make assertions about expected results

        // We expect two warehouses
        System.assert(nearbyWarehouses.size() == 2);

        // We expect two SPECIFIC warehouses, in order of proximity
        System.assert(nearbyWarehouses[0].Name == closestWarehouse.Name);
        System.assert(nearbyWarehouses[1].Name == closeWarehouse.Name);

        // We do NOT expect to see the warehouse that's too far away
        if(0 < nearbyWarehouses.size()) {
            for (Warehouse__c wh : nearbyWarehouses) {
                System.assert(wh.Name != tooFarWarehouse.Name);
            }
        }

        // test that we don't find anything further than 20 miles
        static testMethod void testDontFindWarehousesFurtherThanTwentyMiles() {
            // Eiffel Tower, Paris, France
            String myLat = '48.85837';
            String myLon = '2.294481';

            // Create test warehouse data
            Warehouse__c closestWarehouse = createClosestTestWarehouse();
```

```

Warehouse__c closeWarehouse = createCloseTestWarehouse();
Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

// Perform the test execution
Test.startTest();
List<Warehouse__c> nearbyWarehouses =
    WarehouseUtils.findNearbyWarehouses(myLat, myLon);
Test.stopTest();

// We expect to see NO warehouses
System.assert(nearbyWarehouses.size() == 0);
}

// test the class constructor
static testMethod void testClassConstructor() {
    Test.startTest();
    WarehouseUtils utils = new WarehouseUtils(null);
    Test.stopTest();

    // We expect that utils is not null
    System.assert(utils != null);
}

// test that we use a default location if the lat or long is invalid
static testMethod void testFindWarehousesDefaultLocation() {
    // Trigger the default location, which should be SF
    String myLat = null;
    String myLon = null;

    // Create test warehouse data
    Warehouse__c closestWarehouse = createClosestTestWarehouse();
    Warehouse__c closeWarehouse = createCloseTestWarehouse();
    Warehouse__c tooFarWarehouse = createTooFarTestWarehouse();

    // Perform the test execution
    Test.startTest();
    List<Warehouse__c> nearbyWarehouses =
        WarehouseUtils.findNearbyWarehouses(myLat, myLon);
    Test.stopTest();

    // Make assertions about expected results

    // We expect two warehouses
    System.assert(nearbyWarehouses.size() == 2);

    // We expect two SPECIFIC warehouses, in order of proximity
    System.assert(nearbyWarehouses[0].Name == closestWarehouse.Name);
    System.assert(nearbyWarehouses[1].Name == closeWarehouse.Name);

    // We do NOT expect to see the warehouse that's too far away
    if(0 < nearbyWarehouses.size()) {
        for (Warehouse__c wh : nearbyWarehouses) {
            System.assert(wh.Name != tooFarWarehouse.Name);
        }
    }
}

```

```
    }  
}  
  
// helper methods to create test data  
static Warehouse__c createTestWarehouse(String name, Decimal lat, Decimal lon) {  
    Warehouse__c w = new Warehouse__c(  
        Name = name,  
        Location__Latitude__s = lat,  
        Location__Longitude__s = lon  
    );  
    insert w;  
    return w;  
}  
  
static Warehouse__c createClosestTestWarehouse() {  
    // Federal Reserve Bank of SF  
    // Next door to Salesforce HQ  
    return(createTestWarehouse('Warehouse1', 37.7927731, -122.4010922));  
}  
  
static Warehouse__c createCloseTestWarehouse() {  
    // Moscone Center, home of Dreamforce  
    return(createTestWarehouse('Warehouse2', 37.783944, -122.401289));  
}  
  
static Warehouse__c createTooFarTestWarehouse() {  
    // Mount Rushmore, South Dakota  
    return(createTestWarehouse('TooFarWarehouse', 43.879102, -103.459067));  
}  
}
```

VISUALFORCE AND APEX IN ACTION

In the previous sections of the workbook you learned about Visualforce and Apex separately. Like two great tastes that taste great together, Visualforce and Apex are better—more powerful, more flexible, more versatile—when combined. In this section, you'll put Visualforce and Apex together to build a real app that you can use in Salesforce1 on a mobile device.

We'll start by finishing our mobile app that enables mobile technicians to quickly find nearby parts warehouses on their mobile phone. You'll build a Visualforce page, write JavaScript that uses Visualforce's JavaScript remoting to call your Apex method, retrieve the results, and then put them all on a map. Once that's done, you'll package it up and deploy it in Salesforce1. You're going to be surprised just how easy that is!

When you're finished with this section, you will have done the following.

- Link a Visualforce page to back-end Apex code.
- Call Apex methods and use the results on a Visualforce page.
- Write Visualforce controllers and controller extensions using Apex.
- Use Visualforce JavaScript remoting to call Apex code, and convert the results into data for display on the page.
- Add an app you created to Salesforce1 for use by mobile users on their phone or tablet.

Creating Location-Aware Visualforce Pages

You wrote an Apex extension that returns warehouses that are close to a specific latitude and longitude. Now you need an interface for the user to call that query and display the results.

As a reminder, here's our scenario. You're going to write a small app to give mobile technicians that work for the Acme Wireless organization a way to find nearby warehouses. For example, if the technician is out on a call and needs a part, they can use this page to look for warehouses within a 20-mile radius. For each warehouse, a map should display a pin along with the warehouse name, address, and phone number.

There are many ways you could build this app, but to make a mobile-friendly and dynamic page we're going to use the Google Maps API. The JavaScript required to access the API and render maps has already been included in the Enhanced Warehouse as a static resource. We just need to create the page that loads the data and displays the map.

Create a Visualforce Page Linked to the `WarehouseUtils` Class

The first thing to do is create a new Visualforce page and then connect it with the server-side Apex logic. You'll be connecting the Standard List Controller and an extension to the page.

Because we want to deploy this page in Salesforce1, we need to edit a setting to mobile enable the page. This setting is only available in the Setup editor for Visualforce.

1. From Setup, enter *Visualforce Pages* in the **Quick Find** box, then select **Visualforce Pages**.
2. Click **New**.
3. For the Label and Name enter *FindNearbyWarehouses*.
4. Select the checkbox for **Available for Salesforce mobile apps**.

5. In the code editor, replace the generated code with the following.

```
<apex:page sidebar="false" showheader="false"
  standardController="Warehouse__c" recordSetVar="warehouses"
  extensions="WarehouseUtils">

  <!-- resources and styles go here -->

  <!-- JavaScript custom code goes here -->

  <!-- Google Maps target [div] goes here -->

</apex:page>
```

6. Click **Quick Save**.

Now that the page is created and enabled for mobile apps, you can switch to the Developer Console or Development Mode footer to continue editing the page. By now you may have a preference, so use whichever tool works best for you.

Add Static Resources to the Page

You've created the page shell, but before you start writing any JavaScript you'll need to add a reference to several resources the page will use.

These are stored as static resources in Salesforce, and can be associated with the page using the `<apex:includeScript>` component. This component makes sure that JavaScript libraries are included in the rendered HTML's header properly. You're also going to add a small amount of CSS to the page in order to display a full-width version of the map.

1. Inside the `<apex:page>` component replace the comment line `<!-- resources and styles go here -->` with the following code.

```
<!-- Include in Google's Maps API via JavaScript static resource.
  This is for development convenience, not production use.
  See next comment. -->
<apex:includeScript value="{!$Resource.GoogleMapsAPI}" />

<!-- Set YOUR_API_KEY to fix JavaScript errors in production. See
  https://developers.google.com/maps/documentation/javascript/tutorial
  for details of how to obtain a Google Maps API key. -->
<!-- <script type="text/javascript"
  src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=false">
  </script> -->

<!-- Set the map to take up the whole window -->
<style>
  html, body { height: 100%; }
  .page-map, .ui-content, #map-canvas { width: 100%; height:100%; padding: 0; }
  #map-canvas { height: min-height: 100%; }
</style>
```

2. Click **Quick Save**.

The code you've just added references the Google Maps API two different ways. One of them is commented out. The active version is the one included in the static resource, which will work in development. When you're ready to develop a mapping app for real, you'll want to get a Google Maps API key of your own, and replace the `YOUR_API_KEY` string with your real key. Then uncomment that `<script>` tag, and comment out or delete the `<apex:includeScript>` component.

More details about the Google Maps API can be found in the [Google Maps JavaScript API Getting Started](#) guide.

Add a Place to Display the Map

The Google Maps API needs an HTML `<div>` tag "target" to know where to render the graphics.

The Google Maps API renders the map and then inserts it into your page at a place you specify. So, you need to create that placeholder.

1. Just before the closing `</apex:page>` tag, replace the comment line `<!-- Google Maps target [div] goes here -->` with the following code.

```
<!-- All content is rendered by the Google Maps code
      This minimal HTML just provides a target for GMaps to write to -->
<body style="font-family: Arial; border: 0 none;">
  <div id="map-canvas"></div>
</body>
```

2. Click **Quick Save**.

That completes the markup for the page. From here on, it's JavaScript and JavaScript remoting.

Add JavaScript to Query for Warehouses

Now our page is ready for some JavaScript to make it work. You'll start with a function that gets called when the page loads. This function calls the Apex Remote Action method that you created earlier, retrieving a list of warehouses to display.

The code you're about to add is written in JavaScript, but it's using the Visualforce framework behind the scenes. This facility is called *JavaScript remoting*, and it's a terrific way to combine Visualforce with dynamic, interactive pages built with JavaScript.

1. After the `<style>` tag, replace the comment line `<!-- JavaScript custom code goes here -->` with the following code.

```
<script>

    function initialize() {
        var lat, lon;

        // If we can, get the position of the user via device geolocation
        if (navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(function(position) {
                lat = position.coords.latitude;
                lon = position.coords.longitude;

                // Use Visualforce JS Remoting to query for nearby warehouses
                Visualforce.remoting.Manager.invokeAction(
                    '{!$RemoteAction.WarehouseUtils.findNearbyWarehouses}',
                    lat, lon,
                    function(result, event){
                        if (event.status) {
                            console.log(result);
                        }
                    }
                );
            });
        }
    }
}
```

```

        createMap(lat, lon, result);
    } else if (event.type === 'exception') {
        //exception case code
    } else {

    }

    },
    {escape: true}
    );
    });
} else {
    // Set default values for the map if the device
    // doesn't have geolocation capabilities.
    // This is San Francisco:
    lat = 37.77493;
    lon = -122.419416;

    var result = [];
    createMap(lat, lon, result);
}

// createMap function goes here

</script>

```

2. Click **Quick Save**.

The JavaScript function you added does three things.

- First, it uses the `navigator.geolocation` feature in JavaScript to ask the hardware device if it can provide geolocation coordinates. When this code runs, the user will be prompted by their device, requesting permission to share their location.
- Second, if the device query is successful, Visualforce JavaScript remoting is used to call your Remote Action method. You can see it referenced right there in the code, `{!$RemoteAction.WarehouseUtils.findNearbyWarehouses}`, followed by the latitude and longitude parameters the Remote Action expects. How easy—how *cool*—is that?
- Finally, if the device query fails—perhaps the user denied permission to share their location—a default location is defined instead. (Once again, it's San Francisco, home of Salesforce.com.)

Add JavaScript to Build the Map

Now that your code has queried for and retrieved a collection of nearby warehouses, all that's left is to convert the raw data into a map. You might have noticed in the code from the previous step that there are a few references to a `createMap` function. You'll add that next.

1. Before the end `</script>` tag, replace the comment line `// createMap function goes here` with the following code.

```

function createMap(lat, lon, warehouses){
    // Get the map div, and center the map at the proper geolocation
    var currentPosition = new google.maps.LatLng(lat,lon);
    var mapDiv = document.getElementById('map-canvas');
    var map = new google.maps.Map(mapDiv, {
        center: currentPosition,

```

```

        zoom: 13,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    });

    // Set a marker for the current location
    var positionMarker = new google.maps.Marker({
        map: map,
        position: currentPosition,
        icon: 'https://maps.google.com/mapfiles/ms/micons/green.png'
    });

    // Keep track of the map boundary that holds all markers
    var mapBoundary = new google.maps.LatLngBounds();
    mapBoundary.extend(currentPosition);

    // Set markers on the map from the @RemoteAction results
    var warehouse;
    for(var i=0; i<warehouses.length ; i++) {
        warehouse = warehouses[i];
        console.log(warehouses[i]);
        setupMarker();
    }

    // Resize map to neatly fit all of the markers
    map.fitBounds(mapBoundary);

    // setupMarker function goes here

}

```

2. Click **Quick Save**.

This function receives a latitude and longitude for the center of the map—the user’s location—and the results of the query. It creates a new Google Map centered as expected, and then iterates over the results, adding them to the map using the `setupMarker` function. If you haven’t already guessed, that’s the next (and final!) step.

Add JavaScript to Add Warehouse Markers to the Map

The page is nearly complete. Your JavaScript is calling into Apex, getting a list of nearby warehouses, and then using Google to create a map of your current location. Now you just need to put the result markers onto the map.

At the end of the last code snippet you saw a call to the `setupMarker` function, made while iterating through the found warehouses. Here’s the code for that function.

1. Under the `map.fitBounds()` function call and before the end bracket, replace the comment line `// setupMarker function goes here` with the following code.

```

function setupMarker() {
    var warehouseNavUrl;

    // Determine if we are in Salesforce1 and set navigation
    // link appropriately
    try{
        if(sforce.one){
            warehouseNavUrl =

```



```

        'javascript:sforce.one.navigateToSObject(\'\' +
        warehouse.Id + '\')';
    }
} catch(err) {
    console.log(err);
    warehouseNavUrl = '\\\' + warehouse.Id;
}

var warehouseDetails =
    '<a href=\'\' + warehouseNavUrl + \'\'>' +
    warehouse.Name + '</a><br/>' +
    warehouse.Street_Address__c + '<br/>' +
    warehouse.City__c + '<br/>' +
    warehouse.Phone__c;

// Create a panel that appears when the user clicks on the marker
var infowindow = new google.maps.InfoWindow({
    content: warehouseDetails
});

// Add the marker to the map
var marker = new google.maps.Marker({
    map: map,
    position: new google.maps.LatLng(
        warehouse.Location__Latitude__s,
        warehouse.Location__Longitude__s)
});
mapBoundary.extend(marker.getPosition());

// Add the action to open the panel when its marker is clicked
google.maps.event.addListener(marker, 'click', function(){
    infowindow.open(map, marker);
});
}

// page initialization goes here

```

2. Finally, just below that method replace the comment line `// page initialization goes here` with the following code.

```

// Fire the initialize function when the window loads
google.maps.event.addDomListener(window, 'load', initialize);

```

3. Click **Quick Save**.

And with that you should have a map!



Note: The warehouses in the sample data we provided are all located in the San Francisco area. If you're testing the page from another location, be sure to add a few warehouses located within 20 miles of your location.

Summary and Code Check

You should be able to test the page now by going to your instance URL in your browser (for example, <https://na15.salesforce.com/>) and appending `/apex/FindNearbyWarehouses`.

The final page is a lot of JavaScript and a bit of HTML. The standard Visualforce is minimal, but all of the data access was performed through the Visualforce framework using JavaScript remoting.

Here's the entire page if you're not seeing a map in your final version.

```
<apex:page sidebar="false" showheader="false"
    standardController="Warehouse__c" recordSetVar="warehouses"
    extensions="WarehouseUtils">

    <!-- Include in Google's Maps API via JavaScript static resource.
        This is for development convenience, not production use.
        See next comment. -->
    <apex:includeScript value="{!$Resource.GoogleMapsAPI}" />

    <!-- Set YOUR_API_KEY to fix JavaScript errors in production. See
        https://developers.google.com/maps/documentation/javascript/tutorial
        for details of how to obtain a Google Maps API key. -->
    <!-- <script type="text/javascript"
        src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=false">
        </script> -->

    <!-- Set the map to take up the whole window -->
    <style>
        html, body { height: 100%; }
        .page-map, .ui-content, #map-canvas { width: 100%; height:100%; padding: 0; }
        #map-canvas { height: min-height: 100%; }
    </style>

    <script>
        function initialize() {
            var lat, lon;

            // If we can, get the position of the user via device geolocation
            if (navigator.geolocation) {
                navigator.geolocation.getCurrentPosition(function(position) {
                    lat = position.coords.latitude;
                    lon = position.coords.longitude;

                    // Use Visualforce JS Remoting to query for nearby warehouses
                    Visualforce.remoting.Manager.invokeAction(
                        '{!$RemoteAction.WarehouseUtils.findNearbyWarehouses}',
                        lat, lon,
                        function(result, event){
                            if (event.status) {
                                console.log(result);
                                createMap(lat, lon, result);
                            } else if (event.type === 'exception') {
                                //exception case code
                            } else {

                            }
                        },
                        {escape: true}
                    );
                });
            }
        }
    </script>
```

```

    } else {
        // Set default values for the map if the device
        // doesn't have geolocation capabilities.
        // This is San Francisco:
        lat = 37.77493;
        lon = -122.419416;

        var result = [];
        createMap(lat, lon, result);
    }
}

function createMap(lat, lon, warehouses){
    // Get the map div, and center the map at the proper geolocation
    var currentPosition = new google.maps.LatLng(lat,lon);
    var mapDiv = document.getElementById('map-canvas');
    var map = new google.maps.Map(mapDiv, {
        center: currentPosition,
        zoom: 13,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    });

    // Set a marker for the current location
    var positionMarker = new google.maps.Marker({
        map: map,
        position: currentPosition,
        icon: 'https://maps.google.com/mapfiles/ms/micons/green.png'
    });

    // Keep track of the map boundary that holds all markers
    var mapBoundary = new google.maps.LatLngBounds();
    mapBoundary.extend(currentPosition);

    // Set markers on the map from the @RemoteAction results
    var warehouse;
    for(var i=0; i<warehouses.length ; i++) {
        warehouse = warehouses[i];
        console.log(warehouses[i]);
        setupMarker();
    }

    // Resize map to neatly fit all of the markers
    map.fitBounds(mapBoundary);

    function setupMarker(){
        var warehouseNavUrl;

        // Determine if we are in Salesforce and set navigation
        // link appropriately
        try{
            if(sforce.one){
                warehouseNavUrl =
                    'javascript:sforce.one.navigateToSObject(\'' +
                    warehouse.Id + '\')';
            }
        }
    }
}

```

```

        }
    } catch(err) {
        console.log(err);
        warehouseNavUrl = '\\\\' + warehouse.Id;
    }

    var warehouseDetails =
        '<a href="' + warehouseNavUrl + '">' +
        warehouse.Name + '</a><br/>' +
        warehouse.Street_Address__c + '<br/>' +
        warehouse.City__c + '<br/>' +
        warehouse.Phone__c;

    // Create a panel that appears when the user clicks on the marker
    var infowindow = new google.maps.InfoWindow({
        content: warehouseDetails
    });

    // Add the marker to the map
    var marker = new google.maps.Marker({
        map: map,
        position: new google.maps.LatLng(
            warehouse.Location__Latitude__s,
            warehouse.Location__Longitude__s)
    });
    mapBoundary.extend(marker.getPosition());

    // Add the action to open the panel when its marker is clicked
    google.maps.event.addListener(marker, 'click', function(){
        infowindow.open(map, marker);
    });
}

// Fire the initialize function when the window loads
google.maps.event.addDomListener(window, 'load', initialize);

</script>

<!-- All content is rendered by the Google Maps code
      This minimal HTML just provides a target for GMaps to write to -->
<body style="font-family: Arial; border: 0 none;">
    <div id="map-canvas"></div>
</body>

</apex:page>

```

Now that it's working in development, how about adding it to Salesforce1? Onward!

Add the Nearby Warehouses Page to Salesforce1

Now that you have a working nearby warehouses page, you can add it to the mobile app.

There are two steps. First, create a tab to hold the page and make it available in the Salesforce user interface. Second, add the tab to the Salesforce1 navigation menu.

Create a Tab for the Page

Visualforce pages are added to the Salesforce user interface by creating tabs to hold them.

Although we as developers get used to accessing our Visualforce pages using the direct URL, that's not the way our users reach them day to day. Instead, they want to have the pages be accessible from the standard Salesforce user interface. The way you accomplish this is by first creating a new tab to hold the page.

1. From Setup, enter *Tabs* in the *Quick Find* box, then select **Tabs**.
2. In the Visualforce Tabs section, click **New**.
3. In the Visualforce Page drop-down list, select **FindNearbyWarehouses**.
4. In the Tab Label field, enter *Nearby Warehouses*.

The label field is what users see both on the full site and in the mobile app. With that in mind, keep your labels concise.

5. Click into the Tab Style field, and select the **Globe** style.

The icon for this style appears as the icon for the page in the Salesforce1 mobile app's navigation menu.

6. Click **Next**, and **Next** again.
7. Deselect the **Include Tab** checkbox so that the tab isn't included in any of the apps in the desktop version of the site. You only want this tab to appear when users are viewing Salesforce1 on their mobile device.
8. Click **Save**.

Add the Tab to Mobile Navigation

Now that you've created a tab to hold your Visualforce page, you're ready to add the new tab to the Salesforce1 navigation menu.


In this step you add the tab as a navigation menu item in the Salesforce1 mobile app. The menu item will become available to mobile app users who have access to it.

1. From Setup, enter *Navigation* in the *Quick Find* box, then select **Salesforce1 Navigation**.
2. Move **Nearby Warehouses** to the Selected list and then **Save**.

Try Out the App

Your new mobile app is complete! Search for nearby warehouses on your device.

Being able to test your mobile pages inside your desktop browser is great during development. But now that the page is finished and added to Salesforce1, it's important to test it out on the actual devices your users will be using it with.

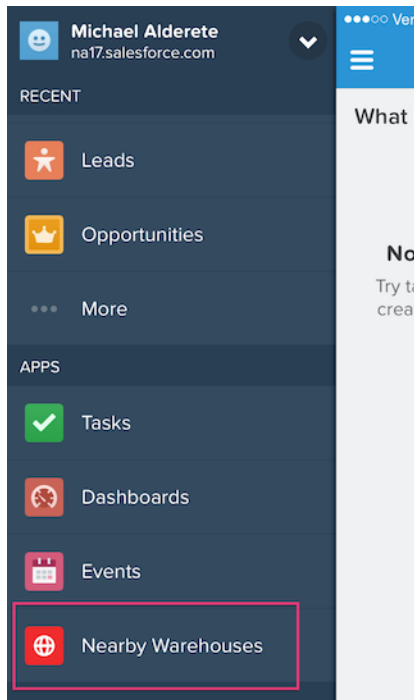
1. Open the Salesforce1 app on your mobile device. Refresh the app by pulling down.
2. Tap  to access the navigation menu.
You should see **Nearby Warehouses** under the Apps section.



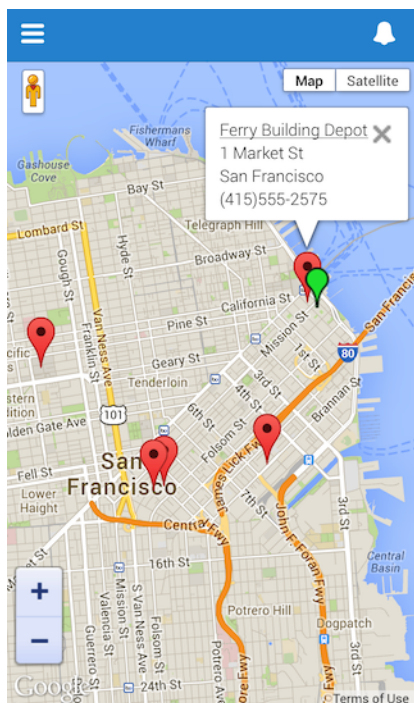
Note:

- If you're using the `/one/one` .app browser version, you may need to refresh the browser to see the page in the navigation menu.
- If you're using the installed mobile app, you may need to log out and log in again to see the change.

3. Tap **Nearby Warehouses**.



4. Click **OK** when you see a prompt that asks to use your current location.
A map that contains warehouse locations within 20 miles appears.





Note: If you don't receive a prompt to share your location, it might be related to your device settings. If that's the case, the geographical area should default to San Francisco. Also, the warehouses in the package sample data are all located in the San Francisco area. If you're testing this from another location, be sure to add a few warehouses located within 20 miles of your location.

Summary

And...that's it! You can see how easy it is to make standard pages and tabs available to your mobile users.

Adding your apps to Salesforce1 is pretty much a point-and-click operation.

Visualforce Pages with Apex Controllers

As you learned earlier in this workbook, Visualforce includes *standard controllers* for every sObject available in your organization. They make it easy for you to create Visualforce pages that handle common features without writing any code beyond the Visualforce itself. For highly customized applications, Visualforce allows you to extend or replace the standard controller with your own Apex code. You can make Visualforce applications available only within your company, or publish them on the Web.

In this tutorial, you'll use Visualforce to create a simple store front page. The page will list products for sale, offer a simple shopping card, and the app and its back-end code will illustrate how Visualforce connects to a controller written in Apex.

Displaying Product Data in a Visualforce Page

In this lesson, you'll extend your first Visualforce page to display a list of products for sale. Although this page might seem fairly simple, there's a lot going on, and we're going to move quickly so we can get to the Apex.

1. In your browser, open your product catalog page at `https://<your-instance>.salesforce.com/apex/Catalog`, and click **Create Page Catalog** to create the new page. Open the Page Editor, if it's not already open.
2. Modify your code to enable the `Merchandise__c` standard controller, by editing the `<apex:page>` tag.

```
<apex:page standardController="Merchandise__c">
```

This connects your page to your `Merchandise__c` custom object, using a built-in controller that provides a lot of basic functionality, like reading, writing, and creating new `Merchandise__c` objects.

3. Next, add the standard list controller definition by setting the `recordSetVar` attribute.

```
<apex:page standardController="Merchandise__c" recordSetVar="products">
```

This configures your controller to work with lists of `Merchandise__c` records all at once, for example, to display a list of products in your catalog. Exactly what we want to do!

4. Click **Save**. You can also press CTRL+S, if you prefer to use the keyboard.
The page reloads, and if the *Merchandise* tab is visible, it becomes selected. Otherwise you won't notice any change on the page. However, because you've set the page to 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__c` records.
5. Replace any code between the two `<apex:page>` tags with a page block that will soon hold the products list.

```
<apex:pageBlock title="Our Products">
```

```
    <apex:pageBlockSection>
```

```

        (Products Go Here)

    </apex:pageBlockSection>

</apex:pageBlock>

```



Note: From here we'll assume that you'll save your changes whenever you want to see how the latest code looks.

- It's time to add the actual list of products. Select the (Products Go Here) placeholder and replace it with a `<apex:pageBlockTable>` component.
- Now you need to add some attributes to the `pageBlockTable` tag. The `value` attribute indicates which list of items the `pageBlockTable` component should iterate over. The `var` attribute assigns each item of that list, for one single iteration, to the `pitem` variable. Add these attributes to the tag.

```
<apex:pageBlockTable value="{!products}" var="pitem">
```

- Now you're 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 `pageBlockTable` tags.

```

<apex:pageBlockTable value="{!products}" var="pitem">
    <apex:column headerValue="Product">
        <apex:outputText value="{!pitem.Name}"/>
    </apex:column>
</apex:pageBlockTable>

```

- Click **Save** and you'll see your product list appear.

Our Products	
Product	
Android	
Desktop	
Laptop	
MacBook Air	
MacBook Pro	

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.

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

```

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

```


11. With three columns, the listing is compressed because the table is narrow. Make it wider by changing the `<apex:pageBlockSection>` tag.

```
<apex:pageBlockSection columns="1">
```

This changes the section from two columns to one, letting the single column be wider.

12. Your code will look like this.

```
<apex:page standardController="Merchandise__c" recordSetVar="products">

    <apex:pageBlock title="Our Products">

        <apex:pageBlockSection columns="1">

            <apex:pageBlockTable value="{!products}" var="pitem">
                <apex:column headerValue="Product">
                    <apex:outputText value="{!pitem.Name}"/>
                </apex:column>
                <apex:column headerValue="Condition">
                    <apex:outputField value="{!pitem.Condition__c}"/>
                </apex:column>
                <apex:column headerValue="Price">
                    <apex:outputField value="{!pitem.Price__c}"/>
                </apex:column>
            </apex:pageBlockTable>

        </apex:pageBlockSection>

    </apex:pageBlock>

</apex:page>
```

And there you have your product catalog!

Tell Me More...

- The `pageBlockTable` 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 `pageBlockTable` component uses.
- You need a way to reference the current record as you iterate over the list. The statement `var="pitem"` assigns a variable called `pitem` that holds the record for the current row.

Using a Custom Apex Controller with a Visualforce Page

You now have a Visualforce page that displays all of your merchandise records. Instead of using the default controller, as you did in the previous tutorial, you're 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.

In this lesson, you'll convert the page from using a standard controller to using your own custom Apex controller. Writing a controller using Apex allows you to go beyond the basic behaviors provided by the standard controller. In the next lesson you'll expand this controller and add some e-commerce features to change the listing into an online store.

To create the new controller class:

1. From Setup, enter *Apex Classes* in the Quick Find box, then select **Apex Classes**.
2. Click **New**.
3. Add the following code as the definition of the class and then click **Quick Save**.

```
public class StoreFrontController {

    List<Merchandise__c> products;

    public List<Merchandise__c> getProducts() {
        if(products == null) {
            products = [SELECT Id, Name, Description__c, Price__c FROM Merchandise__c];
        }
        return products;
    }
}
```

4. Navigate back to your product catalog page at <https://<your-instance>.salesforce.com/apex/Catalog>, and open the Page Editor, if it's not already open.
5. Change the opening `<apex:page>` tag to link your page to your new controller class.

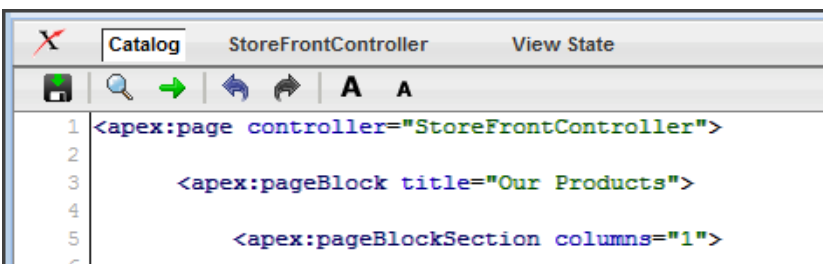
```
<apex:page controller="StoreFrontController">
```

Notice that the attribute name has changed from `standardController` to `controller`. You also remove the `recordSetVar` attribute, because it's only used with standard controllers.

6. Click **Save** to save your changes and reload the page.
The only change you should see is that the *Merchandise* tab is no longer selected.
7. Make the following addition to set the application tab style back to Merchandise.

```
<apex:page controller="StoreFrontController" tabStyle="Merchandise__c">
```

8. Notice that above the Page Editor tool bar there is now a **StoreFrontController** button. Click it to view and edit your page's controller code. Click **Catalog** to return to the Visualforce page code.



You'll use this in the next lessons.

Tell Me More...

- As in the previous lesson, the value attribute of the `pageBlockTable` is set to `{ !products }`, indicating that the table component should iterate over a list called `products`. Because you are using a custom controller, when Visualforce evaluates the `{ !products }` expression, it automatically looks for a method `getProducts()` in your Apex controller.

- The `StoreFrontController` class does the bare minimum to provide the data required by the Visualforce catalog page. It contains that single method, `getProducts()`, which queries the database and returns a list of `Merchandise__c` records.
- The combination of a public instance variable (here, `products`) with a getter method (`getProducts()`) to initialize and provide access to it is a common pattern in Visualforce controllers written in Apex.

Using Inner Classes in an Apex Controller

In the last lesson, you created a custom controller for your Visualforce catalog page. But your controller passes custom objects from the database directly to the view, which isn't ideal. In this lesson, you'll refactor your controller to more correctly use the MVC design pattern, and add some additional features to your page.

1. Click **StoreFrontController** to edit your page's controller code.
2. Revise the definition of the class as follows and then click **Quick Save**.

```
public class StoreFrontController {

    List<DisplayMerchandise> products;

    public List<DisplayMerchandise> getProducts() {
        if(products == null) {
            products = new List<DisplayMerchandise>();
            for(Merchandise__c item : [
                SELECT Id, Name, Description__c, Price__c, Total_Inventory__c
                FROM Merchandise__c]) {
                products.add(new DisplayMerchandise(item));
            }
        }
        return products;
    }

    // Inner class to hold online store details for item
    public class DisplayMerchandise {

        private Merchandise__c merchandise;
        public DisplayMerchandise(Merchandise__c item) {
            this.merchandise = item;
        }

        // Properties for use in the Visualforce view
        public String name {
            get { return merchandise.Name; }
        }
        public String description {
            get { return merchandise.Description__c; }
        }
        public Decimal price {
            get { return merchandise.Price__c; }
        }
        public Boolean inStock {
            get { return (0 < merchandise.Total_Inventory__c); }
        }
        public Integer qtyToBuy { get; set; }
    }
}
```

```

    }
}

```

3. Click **Catalog** to edit your page's Visualforce code.
4. Change the column definitions to work with the property names of the new inner class. Replace the existing column definitions with the following code.

```

<apex:column headerValue="Product">
    <apex:outputText value="{!pitem.Name}"/>
</apex:column>
<apex:column headerValue="Condition">
    <apex:outputText value="{!pitem.Condition}"/>
</apex:column>
<apex:column headerValue="Price">
    <apex:outputText value="{!pitem.Price}"/>
</apex:column>

```

The `outputField` component works automatically with `sObject` fields, but doesn't work at all with custom classes. `outputText` works with any value.

5. Click **Save** to save your changes and reload the page.
You'll notice that the price column is no longer formatted as currency.
6. Change the price `outputText` tag to the following code.

```

<apex:outputText value="{0,number,currency}"/>
    <apex:param value="{!pitem.Price}"/>
</apex:outputText>

```

The `outputText` component can be used to automatically format different data types.

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

```

<apex:page controller="StoreFrontController" tabStyle="Merchandise__c">

    <apex:pageBlock title="Our Products">

        <apex:pageBlockSection columns="1">

            <apex:pageBlockTable value="{!products}" var="pitem">
                <apex:column headerValue="Product">
                    <apex:outputText value="{!pitem.Name}"/>
                </apex:column>
                <apex:column headerValue="Condition">
                    <apex:outputText value="{!pitem.Condition}"/>
                </apex:column>
                <apex:column headerValue="Price" style="text-align: right;">
                    <apex:outputText value="{0,number,currency}"/>
                    <apex:param value="{!pitem.Price}"/>
                </apex:column>
            </apex:pageBlockTable>

        </apex:pageBlockSection>

    </apex:pageBlock>

</apex:page>

```

```

    </apex:pageBlock>

</apex:page>

```

Your catalog page will look something like this.

Our Products		
Product	Condition	Price
Laptop	New	\$500.00
Desktop	New	\$1,000.00
Tablet	New	\$300.00
Rack Server	New	\$3,245.99
Windows Laptop	New	\$445.99
MacBook Air	New	\$1,345.00

Tell Me More...

- The `DisplayMerchandise` class “wraps” the `Merchandise__c` type that you already have in the database, and adds new properties and methods. The constructor lets you create a new `DisplayMerchandise` instance by passing in an existing `Merchandise__c` record. The instance variable `products` is now 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__c` records. It then iterates over the records returned by the query, adding them to a list of `DisplayMerchandise` products, which is then returned.

Adding Action Methods to an Apex Controller

In this lesson, you’ll add action method to your controller to allow it to handle clicking a new **Add to Cart** button, as well as a new method that outputs the contents of a shopping cart. You’ll see how Visualforce transparently passes data back to your controller where it can be processed. On the Visualforce side you’ll add that button to the page, as well as form fields for shoppers to fill in.

1. Click **StoreFrontController** to edit your page’s controller code.
2. Add the following shopping cart code to the definition of `StoreFrontController`, immediately after the `products` instance variable, and then click **Quick Save**.

```

List<DisplayMerchandise> shoppingCart = new List<DisplayMerchandise>();

// Action method to handle purchasing process
public PageReference addToCart() {
    for(DisplayMerchandise p : products) {
        if(0 < p.qtyToBuy) {
            shoppingCart.add(p);
        }
    }
    return null; // stay on the same page
}

public String getCartContents() {
    if(0 == shoppingCart.size()) {
        return '(empty)';
    }
    String msg = '<ul>\n';
    for(DisplayMerchandise p : shoppingCart) {

```

```

        msg += '<li>';
        msg += p.name + ' (' + p.qtyToBuy + ')';
        msg += '</li>\n';
    }
    msg += '</ul>';
    return msg;
}

```

Now you're ready to add a user interface for purchasing to your product catalog.

3. Click **Catalog** to edit your page's Visualforce code.
4. Wrap the product catalog in a form tag, so that the page structure looks like this code.

```

<apex:page controller="StoreFrontController">
    <apex:form>
        <!-- rest of page code -->
    </apex:form>
</apex:page>

```

The `<apex:form>` component enables your page to send user-submitted data back to its controller.

5. Add a fourth column to the products listing table using this code.

```

<apex:column headerValue="Qty to Buy">
    <apex:inputText value="{!pitem.qtyToBuy}" rendered="{! pitem.inStock}"/>
    <apex:outputText value="Out of Stock" rendered="{! NOT(pitem.inStock)}"/>
</apex:column>

```

This column will be a form field for entering a quantity to buy, or an out-of-stock notice, based on the value of the `DisplayMerchandise.inStock()` method for each product.

6. Click **Save** and reload the page.
There's a new column for customers to enter a number of units to buy for each product.
7. Add a shopping cart button by placing the following code just before the `</apex:pageBlock>` tag.

```

<apex:pageBlockSection>
    <apex:commandButton action="{!addToCart}" value="Add to Cart"/>
</apex:pageBlockSection>

```

If you click **Save** and try the form now, everything works...except you can't see any effect, because the shopping cart isn't visible.

8. Add the following code to your page, right above the terminating `</apex:form>` tag.

```

<apex:pageBlock title="Your Cart" id="shopping_cart">
    <apex:outputText value="{!cartContents}" escape="false"/>
</apex:pageBlock>

```

9. Click **Save**, and give the form a try now. You should be able to add items to your shopping cart! In this case, it's just a simple text display. In a real-world scenario, you can imagine emailing the order, invoking a Web service, updating the database, and so on.
10. For a bonus effect, modify the code on the **Add to Cart** `commandButton`.

```

<apex:commandButton action="{!addToCart}" value="Add to Cart" reRender="shopping_cart"/>

```

If you click **Save** and use the form now, the shopping cart is updated via Ajax, instead of by reloading the page.

Our Products

Product	Condition	Price	Qty to Buy
Laptop	New	\$500.00	<input type="text"/>
Desktop	New	\$1,000.00	<input type="text"/>
Tablet	New	\$300.00	<input type="text"/>
Rack Server	New	\$3,245.99	<input type="text"/>
Windows Laptop	New	\$445.99	<input type="text"/>
MacBook Air	New	\$1,345.00	<input type="text" value="1"/>

Add to Cart

Your Cart

- MacBook Air (1)
- iPhone 5S Gold (2)

Tell Me More...

- As you saw in this lesson, 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 **Add to Cart** button, the shopping cart 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.
- If you click **Add to Cart** multiple times with different values in the **Qty to Buy** fields, you'll notice a bug, where products are duplicated in the shopping cart. Knowing what you now know about Apex, can you find and fix the bug? One way might be to change a certain List to a Map, so you can record and check for duplicate IDs. Where would you go to learn the necessary Map methods...?

Summary

In this tutorial, you created a custom user interface for your Warehouse application by writing a Visualforce page with an Apex controller class. You saw how Visualforce pages can use the MVC design pattern, and how Apex classes fit into that pattern. And you saw how easy it was to process submitted form data, manage app and session data, and add convenience methods using an inner class.

CONCLUSION AND WHERE TO GO FROM HERE

Congratulations and thank you for finishing this workbook! Let's take a look at what you learned, and where you might want to go next. This workbook covered a lot of ground, and gives you a great start on becoming a true expert in Force.com development.

- You learned all about creating and editing Visualforce pages and Apex classes, including where to find them in Setup, and how to edit them using multiple tools.
- You used a lot of different Visualforce components, and composed them together in multiple different ways.
- You tried several different ways to architect your Visualforce pages, using both standard Visualforce and JavaScript remoting.
- More importantly, you learned why you might want to use one approach or another to writing your Visualforce pages, depending on where and how it will be used.
- You learned the basics of writing Apex, including creating classes, methods, and tests.
- You leveraged powerful Visualforce runtime features like the Standard Controller, and you wrote Apex code that can replace the Standard Controller when the features it provides aren't right for your app.
- And, most important of all, you hopefully got a taste for building your own custom apps on top of the Force.com Platform.

With powerful built-in functionality, flexible tools, and diverse deployment options, there's a world of opportunity open to you as you begin your career with Visualforce and Apex. You've learned a lot, but there's a lot more available to you.

- First and most importantly, you can find every resource we offer for developers at <https://developer.salesforce.com/>. Bookmark it right now!
- The next step in learning Visualforce is the *Visualforce Developer's Guide*. It's the definitive resource for learning everything about Visualforce, and includes basic, intermediate, and advanced explanations and sample code. It also includes a complete reference to the nearly 150 Visualforce components you can use in your pages and apps. If you're thirsty for Visualforce, this is an almost bottomless well.
- If the code in this book has whet your appetite to write it yourself, there are lots of ways to learn. If books are your thing, we like *Head First Java*. If you'd prefer a formal training class, consider enrolling in *Introduction to Object-Oriented Programming with Force.com Code* (ADM231), a class designed specifically for Salesforce admins who want to learn to create software with Apex.
- To learn more Apex, your next step is the *Apex Workbook*. You've got a big head start on it, but the Apex Workbook offers a more complete look at the language itself, and the many ways you can use it in addition to extending Visualforce.
- Like Visualforce, Apex has a great, in-depth *Force.com Apex Code Developer's Guide* that covers the language in exhaustive detail. It includes a reference to the hundreds of built-in classes that provide higher-level abstractions and services to your application code.
- Salesforce1 is a great way to put your apps in the hands of your mobile users. The *Salesforce1 App Developer Guide* is a comprehensive resource for that exciting platform.
- If you're dreaming of selling your apps in the Salesforce AppExchange, the *ISVforce Guide* is a complete reference to developing and distributing apps that leverage the Force.com platform.

And the list goes on. From [blogs from our engineers and developer marketing team](#), to [developer forums](#), to webinars and videos covering the very latest features, the Force.com Platform offers a rich ecosystem for learning about and building powerful cloud-based applications.