Quick answers to common problems

Visualforce Development Cookbook

Over 75 recipes to help you create powerful custom pages, simplify data-entry, and enrich the Salesforce user interface

Keir Bowden

[PACKT] PUBLISHING enterprise professional expertise distilled

Visualforce Development Cookbook

Over 75 recipes to help you create powerful custom pages, simplify data-entry, and enrich the Salesforce user interface

Keir Bowden

[PACKT] PUBLISHING enterprise
professional expertise distilled

BIRMINGHAM - MUMBAI

Visualforce Development Cookbook

First published: September 2013

Production Reference: 1170913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78217-080-8

`www.packtpub.com`

Cover Image by Javier Barría C. (`jbarriac@yahoo.com`)

Credits

Author
Keir Bowden

Reviewers
Santosh Kumbar
Aruna Lambat
Christopher Alun Lewis
Karanraj Sankaranarayanan
Jitendra Zaa

Acquisition Editor
Edward Gordon

Lead Technical Editor
Amey Varangaonkar

Technical Editor
Amit Ramadas

Project Coordinator
Abhijit Suvarna

Proofreaders
Simran Bhogal
Ameesha Green
Stephen Swaney

Indexers
Monica Ajmera Mehta
Tejal Soni

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Keir Bowden is a 25-year veteran of the IT industry from the United Kingdom. After spending the early part of his career in the Defense industry, he moved into investment-banking systems, implementing systems for Banque Nationale de Paris, CitiGroup, and Deutsche Bank. In the late 1990s, Keir moved into Internet technologies, leading the development of the order management and payment handling systems of one of the first European Internet shopping sites.

Keir started working with Force.com in late 2008 and has been recognized by Salesforce as a Force.com MVP for contribution and leadership in the community. In 2011, he became one of the selected few people worldwide to earn all Salesforce.com certifications, and now serves as a judge on the EMEA Technical Architect Certification Review Boards. Keir is also a prominent blogger on Apex and Visualforce solutions, and regular speaker at events such as Dreamforce and Cloudstock.

Keir is a Chief Technical Officer of BrightGen, a Salesforce.com Platinum Cloud Alliance Partner in the United Kingdom, where he is responsible for current and future technical strategy.

Keir worked as the technical reviewer for *Salesforce CRM Admin Cookbook* by *Packt Publishing* before accepting the challenge of authoring his first book.

I would like to thank my partner, Marie, for putting up with me spending even more time than usual on my computer; something that she hadn't previously thought possible.

About the Reviewers

Santosh Kumbar started working on the Force.com platform in the year 2009. He is an expert in Force.com technologies and currently, he is working as a Senior Software Engineer with a Force.com partner in Pune. He has worked on many Salesforce custom applications for end users, and has also been a part of application development for AppExhange apps. He owns a website `www.santoshkumbar.com`, which is purely built on Visualforce and Apex and also, he is always passionate about exploring new technologies.

Santosh received an Engineering degree from Sapthagiri College of Engineering, Bangalore, specializing in Electronics and Communication. He can be contacted at `santosh224@gmail.com` or found on Twitter `@san_224`.

Aruna Lambat is an enthusiastic Technical Leader working on the Salesforce.com technology with a profound understanding of software design and development. She is passionate about building better products and providing excellent services, leading to a higher rate of customer satisfaction.

She started working on the Salesforce.com platform since 2008. She entered into IT acquaintance in 2004 as a student. She has completed her Master's degree in Computer Applications from Maharashtra, India. She has been associated with the IT industry since 2007 having started her carrier as a Java developer, and later shifted her focus to Cloud computing, specifically in Salesforce.com.

She is a Salesforce Certified Developer (DEV401), Administrator (ADM201), and Advanced Administrator (ADM301/211), giving regular contributions to the Salesforce developer community. She is also certified for Java knowledge as a Sun Certificated Java developer (SCJP), and Sun Certified Web Component Developer (SCWCD).

Before contributing to this book as a reviewer, she worked previously on the following two Salesforce books:

- She helped author for citing the example during the book *Force.com Developer Certification Handbook (DEV401)* by *Packt Publishing*.
- Worked as a Technical Reviewer for the book *Force.com Tips and Tricks* by *Packt Publishing*.

Aruna works with a reputed India-based IT MNC; it is primarily engaged in providing a range of outsourcing services, business process outsourcing, and infrastructure services. Aruna works as a Lead Consultant/Salesforce Application Architect on Salesforce.com technology based customer services.

Aruna resides in Pune, a cultural capital of Maharashtra, also known for its educational facilities and relative prosperity. She is from Nagpur, also known as "Orange City" where her parents are currently staying. She completed her education from this city and achieved success at different points in her career with immense support from her parents. Aruna loves going on nature visits, reading fiction books, playing pool, and catching up with friends in her free time.

Aruna can be contacted via e-mail at `Aruna.Lambat@gmail.com`. Her LinkedIn profile name is Aruna Lambat, her Twitter handle is `@arunalambat`, and she is available on Facebook at `/aruna.lambat`.

My special thanks to my parents, Mr and Mrs Anandrao Lambat, for always being there with me, for their immense help and support, and guiding me through each and every step making it so enlightening.

Christopher Alun Lewis is a Salesforce.com Certified Force.com Advanced Developer with many years' experience developing on the platform. He works for Desynit, a Salesforce partner based in Bristol in the South West of England, where he helps design, architect, and build Force.com solutions for a wide variety of clients.

Christopher is a key contributor to the Salesforce development community. In his spare time, he writes a popular blog (`christopheralunlewis.blogspot.com`), organizes local Force.com developer community meetings, and volunteers his Force.com skills to local charities.

I was delighted when I was invited to review this book. Thanks to Keir for creating a great reference for fellow Force.com developers, and being a constant positive presence in the community.

Karanraj Sankaranarayanan (Karan) is a certified Salesforce.com developer and works as a Salesforce consultant in HCL Technologies. Karan holds a Bachelor's degree in Engineering from Anna University with a specialization in Computer Science. Overall, he has 3 years of experience in the Salesforce platform and the IT industry. He is very much passionate about the Salesforce platform, an active member/contributor of the Salesforce customer community/developer forum, and writes technical blogs too.

He is also the leader of the Chennai Salesforce Platform Developer user group based in Chennai, India. He was one of the reviewers of the book *Force.com Tips and Tricks* by *Packt Publishing*. He can be reached via Twitter (`@karanrajs`) and through Salesforce community `https://success.salesforce.com/profile?u=00530000004fXkCAAU`.

> I would like to thank the author of this book Keir Bowden (also known as Bob buzzard) and Packt Publishing for giving me the wonderful opportunity to review this book. It really has been a great pleasure to be a part of this wonderful book.

Jitendra Zaa (`@ilovenagpur`) is a Force.com developer and owner of the known Salesforce blog `blog.shivasoft.in`. He has worked extensively on almost every area of Force.com such as Integration, Data Loading, AppExchange, and Application Development. He is a Java and Salesforce Certified Developer, Administrator, and Consultant.

Jitendra has more than six years of experience in software development using Salesforce, Java, PHP, ORMB, J2ME, and ASP.NET technologies. He is currently working with Cognizant Technology Solutions, Pune, and graduated from RTM Nagpur University.

> I wish to thank my parents, family, friends, and especially my wife for helping me to set aside time for writing blogs and encouraging me. Also, I would like to thank the Packt Publishing team and the author of this book for giving me this unique opportunity.

Table of Contents

Preface

The Visualforce framework allows developers to build highly customized, personalized, and branded user interfaces for their Salesforce and Force.com applications. Hosted natively on the Force.com platform, Visualforce gives developers complete control over all areas of the user interface, allowing them to satisfy complex business requirements and support multiple devices.

Visualforce pages use a mixture of HTML and Visualforce components, which are processed server side and delivered to the browser as HTML. This allows the use of standard web technologies, such as CSS and JavaScript, to provide an enriched and dynamic user experience.

Visualforce Development Cookbook provides solutions for a variety of challenges faced by Salesforce developers, and demonstrates how easy it is to build rich, interactive pages using Visualforce. Each recipe contains clear, step-by-step instructions along with detailed explanations of the key areas of Visualforce and Apex code that deliver the solution.

Whether you are looking to make a minor addition to the standard page functionality or override it completely, this book will provide you with practical examples that can be readily adapted to a number of scenarios.

What this book covers

Chapter 1, *General Utilities*, covers enhancing or replacing standard functionality with Visualforce, systemizing business processes, guiding users through the creation and ongoing management of data, and writing effective tests.

Chapter 2, *Custom Components*, demonstrates how to create custom Visualforce components to encapsulate functionality for re-use across multiple pages, and techniques to allow communication between component and page controllers.

Chapter 3, *Capturing Data Using Forms*, describes how to capture data entered in a Visualforce page and send this to the server for processing.

Chapter 4, *Managing Records*, offers techniques to streamline and enhance the management of Salesforce data using Visualforce pages, using styling to indicate required fields, and changing pages in response to user actions.

Chapter 5, *Managing Multiple Records*, covers recipes to manage multiple records in a single page, from editing parent and child records through to managing a deep and wide hierarchy.

Chapter 6, *Visualforce Charts*, presents a series of recipes to create charts of increasing complexity, embed a chart into a standard Salesforce page, and add multiple charts to a single page in a similar style to a Salesforce dashboard.

Chapter 7, *JavaScript*, shows how to use JavaScript to provide a variety of client-side enhancements, including confirmation of user actions, instant feedback on user inputs, and animation of content to create tickers and carousels.

Chapter 8, *Force.com Sites*, provides step-by-step instructions to configure a publicly accessible website, allowing visitors to access Salesforce records and extracting boilerplate content out to re-usable templates.

Chapter 9, *jQuery Mobile*, demonstrates how to use Visualforce in conjunction with the jQuery Mobile framework to produce mobile pages to interact with data stored in Salesforce.

What you need for this book

In order to build the recipes in this book, you will need an Enterprise, Unlimited, or Developer (recommended) Edition of Salesforce and System Administrator access. You will also need a supported browser—the latest version of Google Chrome, Mozilla Firefox, Apple Safari 5 or 6, or Internet Explorer 9 or 10.

Who this book is for

This book is intended for intermediate Visualforce developers, who are familiar with the basics of Force.com, Visualforce, and Apex development. An understanding of the basics of HTML, CSS, and JavaScript is also useful for some of the more advanced recipes.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The list of available status field values for the converted lead is created by extracting all values from the `LeadStatus` database table."

A block of code is set as follows:

```
public void uploadImage()
{
  att.parentId = parentId;
  att.Name='image';
  insertatt;

  att=new Attachment();
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
General Utilities

In this chapter, we will cover the following recipes:

- Overriding standard buttons
- Data-driven styling
- Turning off an action poller
- Visualforce in the sidebar
- Passing parameters to action methods
- Reacting to URL parameters
- Passing parameters between Visualforce pages
- Opening a pop-up window
- Adding a launch page
- Testing a custom controller
- Testing a controller extension

Introduction

This chapter provides solutions for a variety of situations that Visualforce developers are likely to encounter on a regular basis. Enhancing or replacing standard functionality with Visualforce enriches the user experience, improving user productivity and adoption. Visualforce also allows business processes to be highly systemized, guiding users through the creation and the ongoing management of data. Writing effective tests for Visualforce controllers is a key skill that allows developers to deploy Visualforce pages to production, and be confident that they will work as intended.

Overriding standard buttons

Two common complaints from users are that the information they are interested in requires a number of clicks to access, or that there is too much information on a single page, resulting in a cluttered layout that requires significant scrolling. This is an area where a **Visualforce override** can make a significant difference by traversing relationships to display information from a number of records on a single page.

Salesforce allows the standard pages associated with sObject record actions, such as view and edit, to be overridden with Visualforce pages. This is typically used to display the record in a branded or customized format; for example, to display the details and related lists in separate tabs.

In this recipe, we will override the standard page associated with viewing an account record with a Visualforce page that not only provides a tabbed user interface, but also lifts up additional activity information from the related contact list and line item information from the related opportunity lists. Further, the related opportunities displayed will be limited to those which are open.

[Only Visualforce pages that use the standard controller for the sObject can override standard pages.]

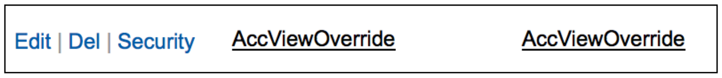
Getting ready

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

How to do it...

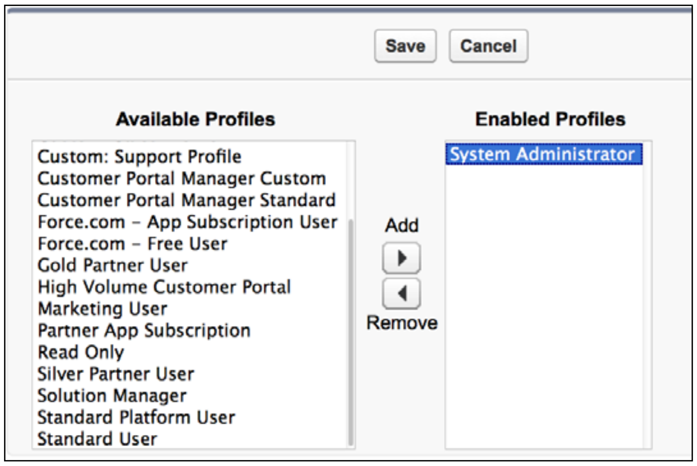
1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `AccViewOverride` in the **Label** field.
4. Accept the default **AccViewOverride** that is automatically generated for the **Name** field.
5. Paste the contents of the `AccViewOverride.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Then, navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

7. Locate the entry for the **AccViewOverride** page and click on the **Security** link.



Edit | Del | Security AccViewOverride AccViewOverride

8. On the resulting page, select which profiles should have access and click on the **Save** button.



Available Profiles:
Custom: Support Profile
Customer Portal Manager Custom
Customer Portal Manager Standard
Force.com – App Subscription User
Force.com – Free User
Gold Partner User
High Volume Customer Portal
Marketing User
Partner App Subscription
Read Only
Silver Partner User
Solution Manager
Standard Platform User
Standard User

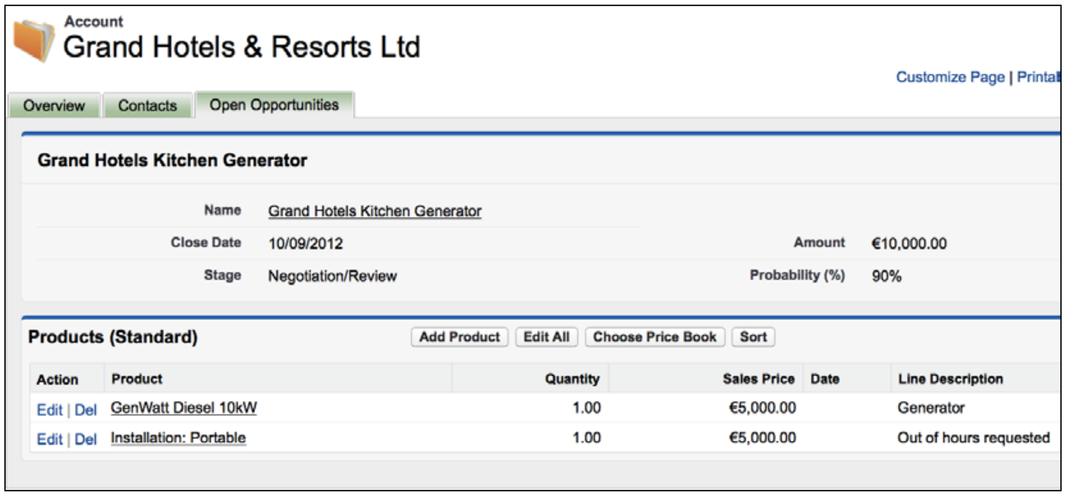
Enabled Profiles:
System Administrator

[As the record view override applies to all users, ensure that all profiles are given access to the Visualforce page. Any user with a profile that does not have access will receive an **Insufficient Privileges** error when attempting to view an account record.]

9. Now that the Visualforce page is complete, configure the account view override. Navigate to **Your Name** | **Setup** | **Customize** | **Accounts** | **Buttons, Links and Actions**.
10. Locate the **View** entry on the resulting page and click on the **Edit** link.
11. On the following page, locate the **Override With** entry, check the **Visualforce Page** radio button, and choose **AccViewOverride** from the list of available pages.
12. Click on the **Save** button.

How it works...

When a user clicks on an account record link anywhere in Salesforce, the tabbed page with details from related records is displayed, as shown in the following screenshot:



Grand Hotels Kitchen Generator

Name	Grand Hotels Kitchen Generator		
Close Date	10/09/2012	Amount	€10,000.00
Stage	Negotiation/Review	Probability (%)	90%

Products (Standard)

Action	Product	Quantity	Sales Price	Date	Line Description
Edit | Del	GenWatt Diesel 10kW	1.00	€5,000.00		Generator
Edit | Del	Installation: Portable	1.00	€5,000.00		Out of hours requested

The key areas of the code are the tabs for the related records. The **Open Opportunities** tab iterates the opportunities related list, and generates an `<apex:pageblock />` for each opportunity that is currently open by encapsulating this inside a conditionally rendered `<apex:outputPanel />`.

```
<apex:repeat value="{!Account.Opportunities}" var="opp">
  <apex:outputPanel rendered="{!NOT(opp.IsClosed)}">
    <apex:pageBlock title="{!opp.Name}">
```

Then, the standard `<apex:relatedList />` component is used to generate the opportunity product list by specifying the current value of the opportunity iterator as the subject of the component.

```
<apex:relatedList subject="{!opp}" list="OpportunityLineItems" />
```

Data-driven styling

A useful technique when creating a custom user interface with Visualforce is to conditionally style important pieces of information to draw the user's attention to them as soon as a page is rendered.

Most Visualforce developers are familiar with using **merge fields** to provide sObject field values to output tags, or to decide if a section of a page should be rendered. In the tag shown below, the merge field, `{!account.Name}`, will be replaced with the contents of the name field from the account sObject:

```
<apex:outputField value="{!account.Name}"/>
```

Merge fields can also contain formula operators and be used to dynamically style data when it is displayed.

In this recipe we will display a table of campaign records and style the campaign cost in green if it was within budget, or red if it was over budget.

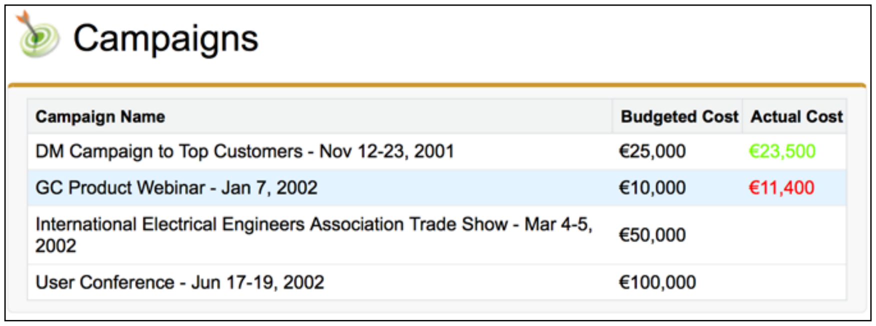
How to do it...

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `ConditionalColour` in the **Label** field.
4. Accept the default **ConditionalColour** that is automatically generated for the **Name** field.
5. Paste the contents of the `ConditionalColour.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Click on the **Save** button to save the page.
7. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
8. Locate the entry for the **ConditionalColour** page and click on the **Security** link.
9. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ConditionalColour** page: `https://<instance>/apex/ConditionalColour`. Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

A list of campaigns is displayed, with the campaign cost rendered in red or green depending on whether it came in on or over budget.



Campaigns

Campaign Name	Budgeted Cost	Actual Cost
DM Campaign to Top Customers - Nov 12-23, 2001	€25,000	€23,500
GC Product Webinar - Jan 7, 2002	€10,000	€11,400
International Electrical Engineers Association Trade Show - Mar 4-5, 2002	€50,000	
User Conference - Jun 17-19, 2002	€100,000	

Conditional styling is applied to the **Actual Cost** column by comparing the actual cost with the budgeted cost.

```
<apex:column style="color:
   {!IF(AND(NOT(ISNULL(campaign.ActualCost)),
   campaign.ActualCost<=campaign.BudgetedCost),
  "lawngreen", "red")}" value="{!campaign.ActualCost}"/>
```

See also

- The *Data-driven decimal places* recipe in *Chapter 2*, *Custom Components* shows how to format numeric values to a specified number of decimal places.

Turning off an action poller

The standard Visualforce `<apex:actionPoller/>` component sends AJAX requests to the server based on the specified time interval. An example use case is a countdown timer that sends the user to another page when the timer expires. But what if the action poller should stop when a condition in the controller becomes true, for example, when a batch apex job completes or an update is received from a third-party system?

In this recipe, we will simulate the progression of a payment through a number of states. An **action poller** will be used to retrieve the latest state from the server and display it to the user. Once the payment reaches the state **Complete**, the action poller will be disabled.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `PollerController.cls` Apex class from the code download into the Apex Class area.

   Note that there is nowhere to specify a name for the class when creating through the setup pages; the class name is derived from the Apex code.

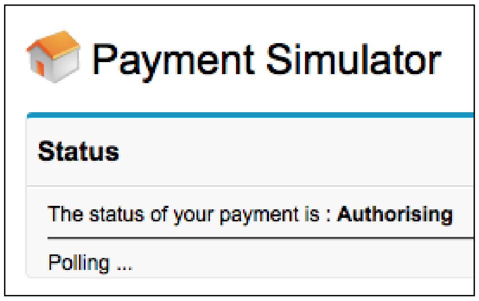
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionPoller` in the **Label** field.
8. Accept the default **ActionPoller** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionPoller.page` file from the code download into the Visualforce Markup area.
10. Click on the **Save** button to save the page.
11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **ActionPoller** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionPoller** page: `https://<instance>/apex/ActionPoller`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

The page polls the server for the current state, displaying the message **Polling ...** when the action poller executes as shown in the following screenshot:



The status of your payment is : **Authorising**

Polling ...

Once the current state reaches **Complete**, the action poller terminates.

The key to this recipe is the `enabled` attribute on the `actionPoller` component.

```
<apex:actionPoller action="{!movePayment}"
     rerender="payment" interval="5" status="status"
     enabled="{!paymentState!='Complete'}"/>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

This merge field references the `paymentState` property from the custom controller, which is evaluated each time the action poller executes until it becomes false. At this time the action poller is permanently disabled.

The **Polling ...** message is generated by the `actionStatus` component associated with the action poller. This component has a `startText` attribute but not a `stopText` attribute, which means that the text will only be displayed while the AJAX request is in progress.

```
<apex:actionStatus startText="Polling ..." id="status"/>
```

See also

- The *Using action functions* recipe in *Chapter 7*, *JavaScript* shows how to execute a controller.

Visualforce in the sidebar

Visualforce is commonly used to produce custom pages that override or supplement standard platform functionality. Visualforce pages can also be incorporated into any HTML markup through use of an **iframe**.

[An **iframe**, or **inline frame**, nests an HTML document inside another HTML document. For more information, visit `http://reference.sitepoint.com/html/iframe`.]

In this recipe, we will add a Visualforce page to a Salesforce sidebar component. This page will display the number of currently open cases in the organization, and will be styled and sized to fit seamlessly into the sidebar.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

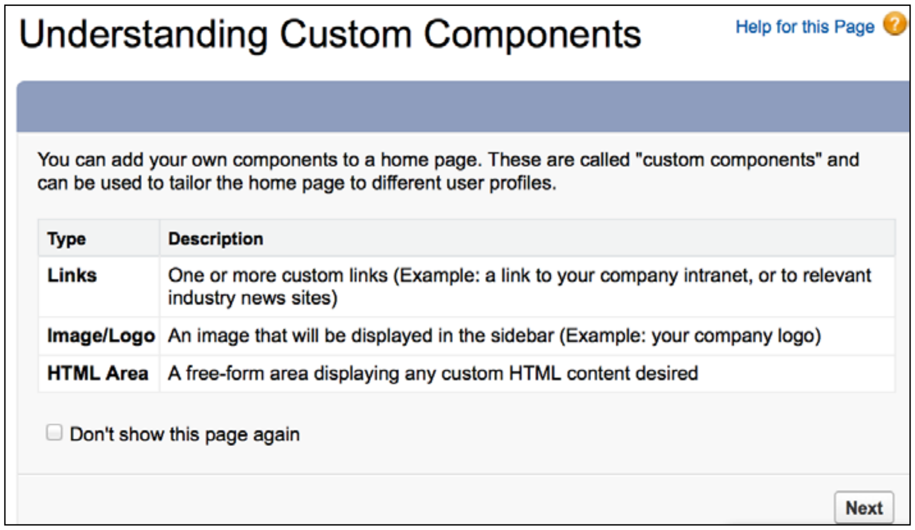
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `CasesSidebarController.cls` Apex class from the code download into the Apex Class area.
4. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
5. Click on the **New** button.
6. Enter `CasesSidebar` in the **Label** field.
7. Accept the default **CasesSidebar** that is automatically generated for the **Name** field.
8. Paste the contents of the `CasesSidebar.page` file from the code download into the Visualforce Markup area.
9. Click on the **Save** button to save the page.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **CasesSidebar** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

[Ensure that all profiles whose sidebar will display the Visualforce page are given access. Any user with a profile that does not have access will see an **Insufficient Privileges** error in their sidebar.]

13. Next, create the home page component by navigating to the Home Page Components setup page by clicking on **Your Name** | **Setup** | **Customize** | **Home** | **Home Page Components**.
14. Scroll down to the **Custom Components** section and click on the **New** button.
15. If the **Understanding Custom Components** information screen appears, as shown in the following screenshot, click on the **Next** button.



Understanding Custom Components

You can add your own components to a home page. These are called "custom components" and can be used to tailor the home page to different user profiles.

Type	Description
Links	One or more custom links (Example: a link to your company intranet, or to relevant industry news sites)
Image/Logo	An image that will be displayed in the sidebar (Example: your company logo)
HTML Area	A free-form area displaying any custom HTML content desired

[To stop this information screen appearing each time you create a home page component, select the **Don't show this page again** box before clicking on the **Next** button.]

16. On the next page, **Step 1. New Custom Components**, enter `Case Count by Status` in the **Name** field, select the **HTML Area** option, and click on the **Next** button.

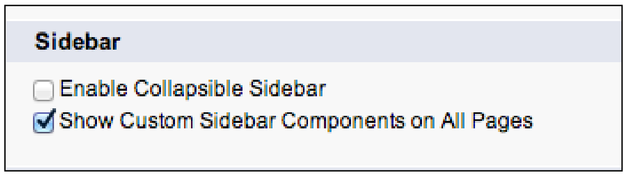
17. On the next page, **Step 2. New Custom Components**, select the **Narrow (Left) Column** option.
18. Select the **Show HTML** box.
19. Paste the following markup into the editable area:

```
<iframe style="border: none" src="/apex/CasesSidebar"
seamless=""></iframe>
```

20. Click on the **Save** button.
21. Next, add the new component to one or more home page layouts. Navigate to **Your Name** | **Setup** | **Customize** | **Home** | **Home Page Layouts**.
22. Locate the name of the home page layout you wish to add the component to and click on the **Edit** link.
23. On the resulting page, **Step 1. Select the Components to show**, select the **Case Count by Status** box in the **Select Narrow Components to Show** section and click on the **Next** button.
24. On the next page, **Step 2. Order the Components**, use the arrow buttons to move the **Case Count by Status** component to the desired position in the **Narrow (Left) Column** list and click on the **Save** button.
25. Repeat steps 22 to 24 for any other home page layouts that will contain the sidebar component.

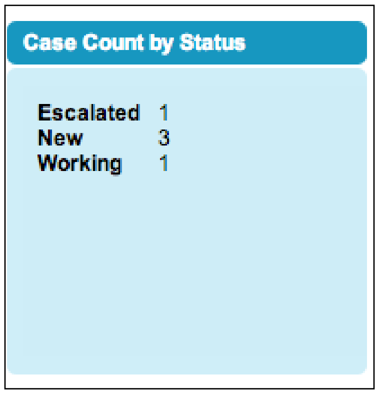
This will add the component to the sidebar of the home page only. To add it to the sidebar of all pages, a change must be made to the user interface settings.

26. Navigate to **Your Name** | **Setup** | **Customize** | **User Interface** and locate the **Sidebar** section.
27. Select the **Show Custom Sidebar Components on All Pages** box as shown in the following screenshot, and click on the **Save** button.


Sidebar
☐ Enable Collapsible Sidebar
☑ Show Custom Sidebar Components on All Pages

How it works...

The component appears in the sidebar on all pages, showing the number of cases open for each nonclosed status, as shown in the following screenshot:



Escalated	1
New	3
Working	1

There's more...

The case counts displayed in the sidebar will be retrieved when the page is displayed, but will remain static from that point. An action poller can be used to automatically refresh the counts at regular intervals. However, this will introduce a security risk, as each time the poller retrieves the updated information it will refresh the user's session. This means that if a user were to leave their workstation unattended, the Salesforce session will never expire. If this mechanism is used, it is important to remind users of the importance of locking their workstation should they leave it unattended.

Passing parameters to action methods

When developers move to Apex/Visualforce from traditional programming languages, such as Java or C#, a concept many struggle with is how to pass parameters from a Visualforce page to a controller action method.

Passing parameters to an action method is key when a Visualforce page allows a user to manage a list of records and carry out actions on specific records. Without this, the action method cannot determine which record to apply the action to.

In this recipe, we will output a list of opportunities and for each open opportunity, provide a button to update the opportunity status to **Closed Won**. This button will invoke an action method to remove the list element and will also send a parameter to the controller to identify which opportunity to update.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

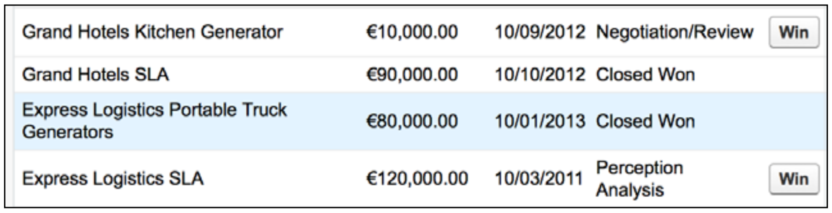
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ActionParameterController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionParameter` in the **Label** field.
8. Accept the default **ActionParameter** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionParameter.page` file from the code download into the Visualforce Markup area.
10. Click on the **Save** button to save the page.
11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **ActionParameter** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

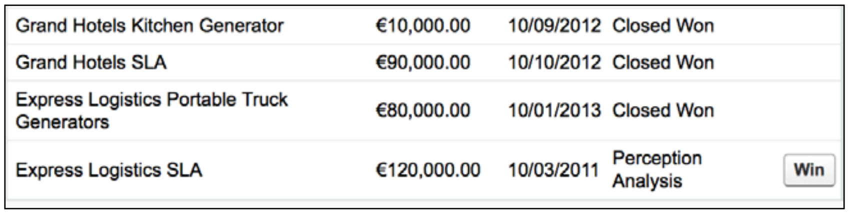
Opening the following URL in your browser shows the list of currently open opportunities: `https://<instance>/apex/ActionParameter`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Grand Hotels Kitchen Generator	€10,000.00	10/09/2012	Negotiation/Review	Win
Grand Hotels SLA	€90,000.00	10/10/2012	Closed Won	
Express Logistics Portable Truck Generators	€80,000.00	10/01/2013	Closed Won	
Express Logistics SLA	€120,000.00	10/03/2011	Perception Analysis	Win

Clicking on the **Win** button for the **Grand Hotels Kitchen Generator** opportunity updates the status to **Closed Won** and redraws the list of opportunities.



Grand Hotels Kitchen Generator	€10,000.00	10/09/2012	Closed Won	
Grand Hotels SLA	€90,000.00	10/10/2012	Closed Won	
Express Logistics Portable Truck Generators	€80,000.00	10/01/2013	Closed Won	
Express Logistics SLA	€120,000.00	10/03/2011	Perception Analysis	Win

The page markup to send the parameter to the controller is as follows:

```
<apex:commandButton value="Win" action="{!winOpp}" status="status"
        rerender="opps_pb"
        rendered="{!opp.StageName!='Closed Won'}">
   <apex:param name="oppIdToWin" value="{!opp.Id}"
assignTo="{!oppIdToWin}" />
</apex:commandButton>
```

The `<apex:param />` component defines the value of the parameter, in this case, the ID of the opportunity, and the controller property that the parameter will be assigned to – `oppIdToWin`.

Note that there is a `rerender` attribute on the command button. If this attribute is omitted, making the button a simple postback request, the parameter will not be passed to the controller. This is a known issue with Visualforce as documented in the following knowledge article: `http://help.salesforce.com/apex/HTViewSolution?id=000002664&language=en_US`.

The property is declared in the controller in a normal way.

```
public Id oppIdToWin {get; set;}
```

Finally, the action method is invoked when the button is pressed.

```
public PageReference winOpp()
{
   Opportunity opp=new Opportunity(Id=oppIdToWin,
                                   StageName='Closed Won');
   update opp;
   return null;
}
```

The ID of the opportunity to update is assigned to the `oppIdToWin` controller property *before the action method is invoked*; thus, the action method can simply access the property to get the parameter value.

Reacting to URL parameters

URL parameters are used to pass information to Visualforce pages that the page or controller can then react to. For example, setting a record ID parameter into the URL for a page that uses a standard controller causes the controller to retrieve the record from the database and make it available to the page.

In this recipe we will create a Visualforce search page to retrieve all accounts where the name contains a string entered by the user. If the parameter name is present in the page URL, a search will be run against the supplied value prior to the page being rendered for the first time.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

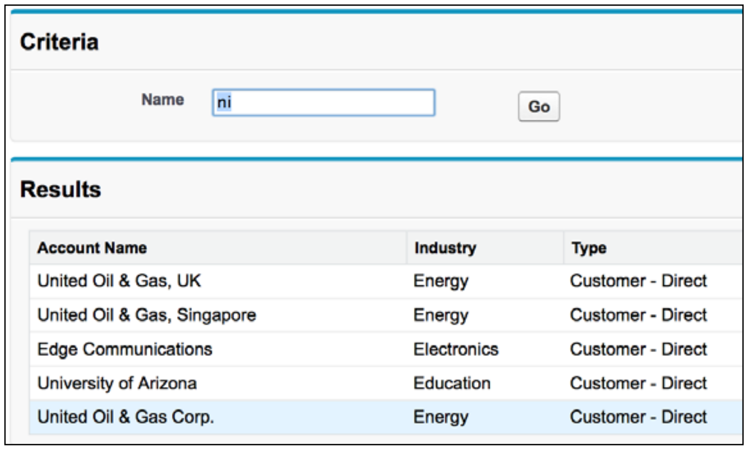
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SearchFromURLController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `SearchFromURL` in the **Label** field.
8. Accept the default **SearchFromURL** that is automatically generated for the **Name** field.
9. Paste the contents of the `SearchFromURL.page` file from the code download into the Visualforce Markup area.
10. Click on the **Save** button to save the page.
11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **SearchFromURL** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser retrieves all accounts where the name field contains the text `ni`: `https://<instance>/apex/SearchFromURL?name=ni`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Name: ni

Account Name	Industry	Type
United Oil & Gas, UK	Energy	Customer - Direct
United Oil & Gas, Singapore	Energy	Customer - Direct
Edge Communications	Electronics	Customer - Direct
University of Arizona	Education	Customer - Direct
United Oil & Gas Corp.	Energy	Customer - Direct

The constructor of the custom controller attempts to extract a value for the parameter **Name** from the page URL, and if one has been supplied, executes the search.

```
public SearchFromURLController()
{
  searched=false;
  String nameStr=
    ApexPages.currentPage().getParameters().get('name');
  if (null!=nameStr)
  {
      name=nameStr;
      executeSearch();
  }
}
```

Note that the constructor also sets the value retrieved from the URL into the **Name** property. This property is bound to the input field on the page, and causes the input field to be prepopulated with the value retrieved from the URL when the page is first rendered.

The action method that executes the search is as follows:

```
public PageReference executeSearch()
{
  searched=true;
  String searchStr='%' + name + '%';
  accounts=[select id, Name, Industry, Type from Account where name
LIKE :searchStr];

  return null;
}
```

Note that `searchStr` is constructed by concatenating the search term with the `%` wildcard characters; this allows the user to enter a fragment of text rather than full words. Also, note that the concatenation takes place outside the SOQL query and the resulting variable is included as a bind expression in the query. If the concatenation takes place directly in the SOQL query, no matches will be found.

See also

- The *Passing parameters between Visualforce pages* recipe in this chapter shows how URL parameters can be used to maintain the state across pages that do not share the same controller.

Passing parameters between Visualforce pages

If a user is redirected from one Visualforce page to another and they both share the same controller and extensions, the controller instance will be retained and re-used, allowing the second page to access any information captured by the first.

If the pages do not share the same controller and extensions, the controller instance will be discarded and the second page will have no access to any information captured by the first. If the state needs to be maintained across the pages in this case, it must be encapsulated in the parameters on the URL of the second page.

In this recipe, we will build on the example from the previous recipe to create a Visualforce search page to retrieve all accounts where the name contains a string entered by the user, and provide a way for the user to edit selected fields on all the accounts returned by the search. The record IDs of the accounts to edit will be passed as parameters on the URL to the edit page.

How to do it...

As the search page makes reference to the edit page, the edit page and associated custom controller must be created first.

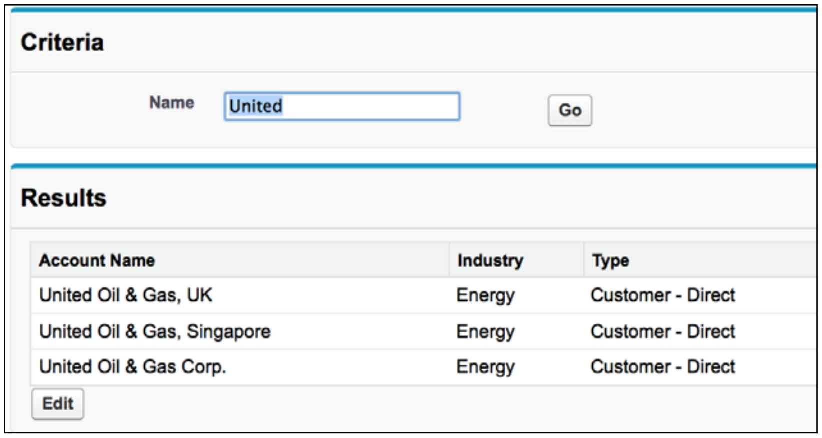
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `EditFromSearchController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the edit Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `EditFromSearch` in the **Label** field.
8. Accept the default **EditFromSearch** that is automatically generated for the **Name** field.

9. Paste the contents of the `EditFromSearch.page` file from the code download into the Visualforce Markup area.
10. Click on the **Save** button to save the page.
11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **EditFromSearch** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click the **Save** button.
14. Next, create the search page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
15. Click on the **New** button.
16. Paste the contents of the `SearchAndEditController.cls` Apex class from the code download into the Apex Class area.
17. Click on the **Save** button.
18. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
19. Click on the **New** button.
20. Enter `SearchAndEdit` in the **Label** field.
21. Accept the default **SearchAndEdit** that is automatically generated for the **Name** field.
22. Paste the contents of the `SearchAndEdit.page` page from the code download into the Visualforce Markup area.
23. Click on the **Save** button to save the page.
24. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
25. Locate the entry for the **SearchAndEdit** page and click on the **Security** link.
26. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser retrieves all accounts where the name field contains the string `United`: `https://<instance>/apex/SearchAndEdit?name=United`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Criteria

Name: United

Results

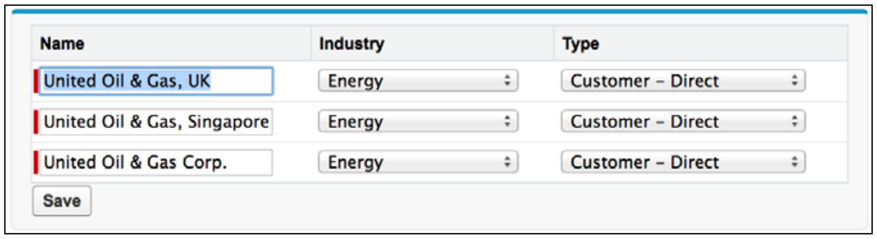
Account Name	Industry	Type
United Oil & Gas, UK	Energy	Customer - Direct
United Oil & Gas, Singapore	Energy	Customer - Direct
United Oil & Gas Corp.	Energy	Customer - Direct

Notice that the page contains an **Edit** button, and clicking on this executes the following action method:

```
public PageReference edit()
{
  PageReference pr=Page.EditFromSearch;
  Integer idx=1;
  for (Account acc : accounts)
  {
    pr.getParameters().put('account' + idx, acc.id);
    idx++;
  }

  return pr;
}
```

This method initially creates a page reference for the edit page—**EditFromSearch**. It then iterates the accounts in the search results and adds an entry to the page reference parameters for the account ID. Each parameter has the name account, concatenated with the index of the result, starting from `1`. This will result in a URL of the form `https://<instance>/apex/EditFromSearch?account1=001i0000006OVLIAA4&account2=001i0000006OVLJAA4`.

The **EditFromSearch** page then renders a form with an editable row per account.



Name	Industry	Type
United Oil & Gas, UK	Energy	Customer – Direct
United Oil & Gas, Singapore	Energy	Customer – Direct
United Oil & Gas Corp.	Energy	Customer – Direct

The constructor of `EditFromSearchController` that manages the data for the page extracts the IDs from the URL and adds them to a list, starting with `account1`, until it hits a parameter index that is not present in the URL.

```
Integer idx=1;
String accStr;
do
{
  accStr=ApexPages.currentPage().getParameters().
get('account' + idx);
  if (accStr!=null)
  {
    ids.add(accStr);
  }
  idx++;
}
while (null!=accStr);
```

The action method that saves the user's edits redirects them to the standard account tab once the save is complete.

```
return new PageReference('/001/o');
```

Note that accessing the standard tab via this URL is not supported by Salesforce, and if the URL scheme or three character prefix for account (`001`) were to change, this redirection would stop working.

See also

- The *Reacting to URL parameters* recipe in this chapter shows how a controller can process URL parameters prior to rendering a Visualforce page.

Opening a pop-up window

Pop-up browser windows have received mixed reviews in recent years. Originally created before tabbed browsers existed to display additional information without interfering with the page the user had navigated to, they were quickly hijacked and used to display advertisements and spam. Pop ups should be used sparingly in applications and wherever possible in response to an action by the user.

The target attribute can be specified as `_blank` on HTML hyperlink tags to open the link in a new window, but all modern browsers allow the user to specify that new windows should be opened as new tabs instead. Also, if the browser does open the URL in a new window, it will be of the same size as the existing window and block most of it. Opening a window in JavaScript allows for fine-grained control over many aspects of the pop-up window, for example, the size, and whether to display a toolbar.

In this recipe we will create a page that renders a list of accounts, displaying a very small subset of fields per row. A link will be provided on each row to allow the user to view full details of the account in a pop-up window.

> Note that there is no way to ensure that a browser will display a pop-up window. Pop-up blockers generally allow windows to be opened in response to an action by the user, such as clicking on a link, but it is possible for users to configure their browser to block all pop ups regardless of how they were triggered.

How to do it...

This recipe requires two Visualforce pages to be created: the main page containing the list of accounts and the pop-up window page. The pop-up page is referenced by the main page, so this will be created first.

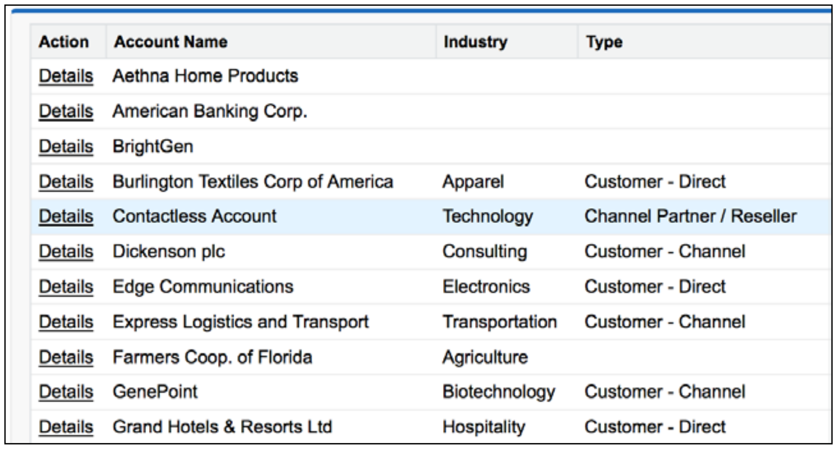
1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.

3. Enter `Popup` in the **Label** field.
4. Accept the default **Popup** that is automatically generated for the **Name** field.
5. Paste the contents of the `Popup.page` file from the code download into the Visualforce Markup area.
6. Click on the **Save** button to save the page.
7. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
8. Locate the entry for the **Setup** page and click on the **Security** link.
9. On the resulting page, select which profiles should have access and click on the **Save** button.
10. Next, create the main account list page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Click on the **New** button.
12. Enter `PopupMain` in the **Label** field.
13. Accept the default **PopupMain** that is automatically generated for the **Name** field.
14. Paste the contents of the `PopupMain.page` file from the code download into the Visualforce Markup area.
15. Click on the **Save** button to save the page.
16. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
17. Locate the entry for the **PopupMain** page and click on the **Security** link.
18. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays a list of accounts: `https://<instance>/apex/PopupMain`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Action	Account Name	Industry	Type
Details	Aethna Home Products		
Details	American Banking Corp.		
Details	BrightGen		
Details	Burlington Textiles Corp of America	Apparel	Customer - Direct
Details	Contactless Account	Technology	Channel Partner / Reseller
Details	Dickenson plc	Consulting	Customer - Channel
Details	Edge Communications	Electronics	Customer - Direct
Details	Express Logistics and Transport	Transportation	Customer - Channel
Details	Farmers Coop. of Florida	Agriculture	
Details	GenePoint	Biotechnology	Customer - Channel
Details	Grand Hotels & Resorts Ltd	Hospitality	Customer - Direct

Note that as this page uses a standard list controller, the list of accounts displayed will be that of the last list view that the user accessed.

The detail link markup is as follows:

```
<apex:outputLink title="View detail in a popup window"
    onclick="return openPopup('{!acc.Id}');">
    Details
</apex:outputLink>
```

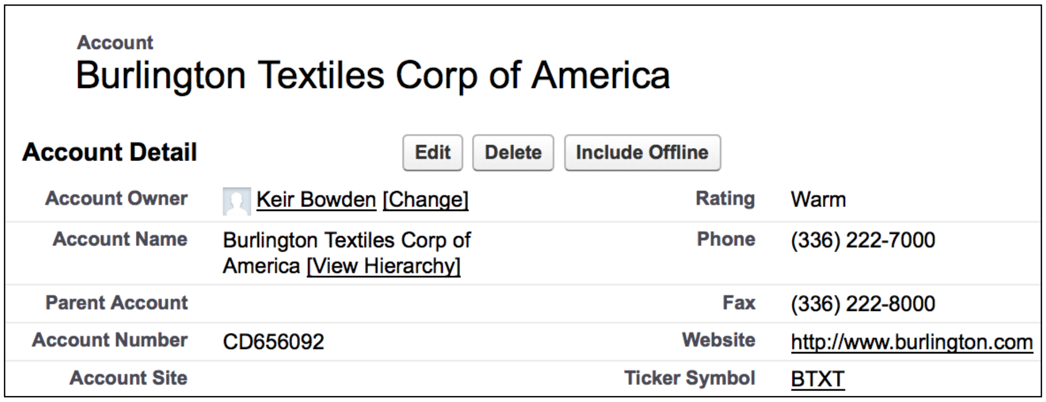
The `onclick` attribute defines the JavaScript function to be invoked when the link is clicked; note the `{!acc.id}` merge field, which passes the ID of the chosen account to the function.

The JavaScript function uses the `window.open` function to open the new window.

```
var newWin=window.open('{!$Page.Popup}?id=' + id, 'Popup',
'height=600,width=650,left=100,top=100,resizable=no,scrollbars=yes,too
lbar=no,status=no');
```

The final parameter details the features required for the new window as a comma separated list of `name=value` pairs.

Clicking on the **Details** link displays the full account details in a pop-up window.



Account

Burlington Textiles Corp of America

Account Detail

Account Owner	Keir Bowden [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

See also

- The *Adding a custom lookup to a form* recipe in *Chapter 3*, *Capturing Data Using Forms* shows how information can be captured in a pop-up window and passed back to the main window to populate input fields.

Adding a launch page

When a Visualforce page is deployed to production, only users whose profiles have been given access via the security settings will be able to access the page. Any user with a profile that does not have access will receive an **Insufficient Privileges** error, which is not a good experience and can lead users to think that the page is crashing.

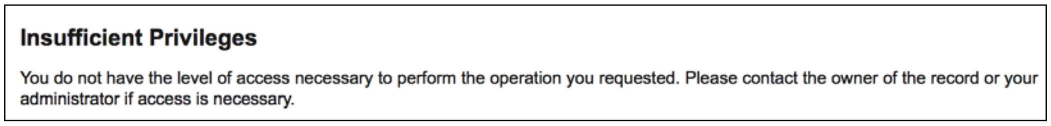
A better solution is to check whether the user has access to the page and if they do not, present a user-friendly message that explains the situation and directs them to where they can get more help.

In this recipe we will create a launch page accessible to all profiles that checks if the user has access to the protected page. If the user has access, they will be transferred to the protected page, while if they don't, they will receive an explanatory message.

How to do it...

This recipe requires a second user login. Ensure that this is not created with the System Administrator profile, as that profile has access to all Visualforce pages regardless of the security settings.

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `Protected` in the **Label** field.
4. Accept the default **Protected** that is automatically generated for the **Name** field.
5. Paste the contents of the `Protected.page` file from the code download into the Visualforce Markup area.
6. Click on the **Save** button to save the page.
7. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
8. Locate the entry for the **Protected** page and click on the **Security** link.
9. On the resulting page, ensure that the profile of your second user does not have access to the **Protected** page.
10. Log in using your second user credentials and attempt to access any account record. You will receive an error message as shown in the following screenshot:


Insufficient Privileges

You do not have the level of access necessary to perform the operation you requested. Please contact the owner of the record or your administrator if access is necessary.

11. Next, create the launch page controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
12. Click on the **New** button.
13. Paste the contents of the `LaunchController.cls` Apex class from the code download into the Apex Class area.
14. Click on the **Save** button.
15. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
16. Click on the **New** button.
17. Enter `Launch` in the **Label** field.
18. Accept the default **Launch** that is automatically generated for the **Name** field.

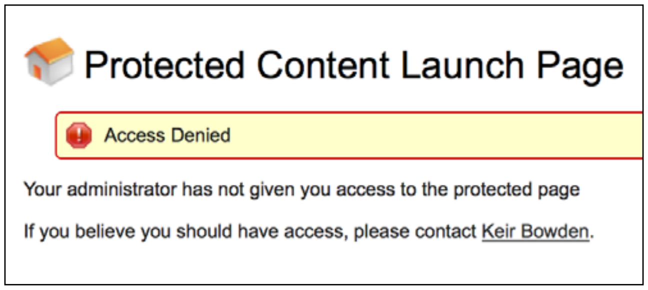
19. Paste the contents of the `Launch.page` file from the code download into the Visualforce Markup area.
20. Click on the **Save** button to save the page.
21. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
22. Locate the entry for the **Setup** page and click on the **Security** link.
23. On the resulting page, give access to all of the profiles and click on the **Save** button.

How it works...

Log in using your second user credentials and open the following URL in your browser: `https://<instance>/apex/Launch`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

The resulting page displays a friendly error message detailing that your user does not have access to the page, and renders a clickable link to request access.


Protected Content Launch Page

Access Denied

Your administrator has not given you access to the protected page

If you believe you should have access, please contact Keir Bowden.

The **Launch** page declaration contains an `action` attribute.

```
<apex:page controller="LaunchController" action="{!allowAccess}">
```

This invokes the `allowAccess` action method in the controller before the page is rendered.

```
public PageReference allowAccess()
{
PageReference pr=Page.Protected;
   try
   {
       pr.getContent();
    }
   catch (Exception e)
   {
```

```
            pr=null;
        }

             return pr;
         }
```

The `allowAccess` method attempts to retrieve the contents of the protected page programmatically. If the contents are retrieved successfully, it returns the page reference for the **Protected** page, which redirects the user to that page. If an exception occurs, the method returns `null`, which leaves the user on the **Launch** page and displays the friendly error message.

Testing a custom controller

Writing unit tests for Visualforce page controllers is often a source of confusion for developers new to the technology. A common mistake is to assume that the page must somehow be rendered and interacted with in the test context, whereas, in reality the page is very much a side issue. Instead, tests must instantiate the controller and set its internal state as though the user interaction had already taken place, and then execute one or more controller methods and confirm that the state has changed as expected.

In this recipe we will unit test `SearchFromURLController` from the *Reacting to URL parameters* recipe.

Getting ready

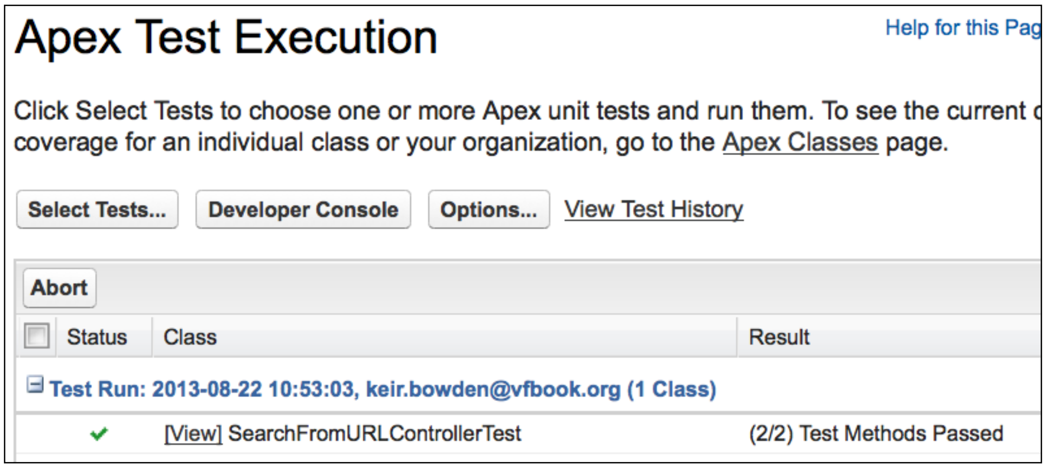
This recipe requires that you have already completed the *Reacting to URL parameters* recipe, as it relies on `SearchFromURLController` being present in your Salesforce instance.

How to do it...

1. Create the unit test class by navigating to the Apex Classes setup page and by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SearchFromURLControllerTest.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. On the resulting page, click on the **Run Tests** button.

How it works...

The tests successfully execute as shown in the following screenshot:



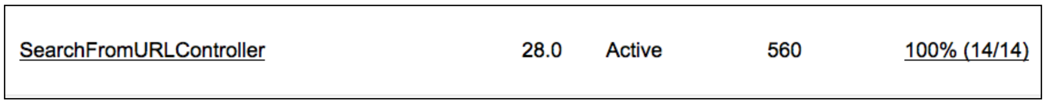
Apex Test Execution

Help for this Pag

Click Select Tests to choose one or more Apex unit tests and run them. To see the current c
coverage for an individual class or your organization, go to the Apex Classes page.

Status	Class	Result
Test Run: 2013-08-22 10:53:03, keir.bowden@vfbook.org (1 Class)		
✓	[View] SearchFromURLControllerTest	(2/2) Test Methods Passed

Navigating back to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes** shows that the tests have achieved 100 percent coverage of the controller.



SearchFromURLController	28.0	Active	560	100% (14/14)

Percentage coverage is important as at least 75 percent coverage across all code must be achieved before classes may be deployed to a production organization.

The test class contains two unit test methods. The first method tests that the search is correctly executed when the search term is passed on the page URL. As unit tests do not have access to organization data, the first task for the test is to set up three test accounts.

```
List<Account> accs=new List<Account>();
accs.add(new Account(Name='Unit Test'));
accs.add(new Account(Name='Unit Test 2'));
accs.add(new Account(Name='The Test Account'));
insert accs;
```

As the controller is reacting to parameters on the URL, the page reference must be set up and populated with the `name` parameter.

```
PageReference pr=Page.SearchFromURL;
pr.getParameters().put('name', 'Unit');
Test.setCurrentPage(pr);
```

Finally, the controller is instantiated, which causes the action method that executes the search to be invoked from the constructor. The test method then confirms that the search was executed and the actual number of matches equals the expected number.

```
SearchFromURLController controller=new
         SearchFromURLController();
System.assertEquals(true, controller.searched);
System.assertEquals(2, controller.accounts.size());
```

The second unit test method tests that the search is correctly executed when the user enters a search term. In this case, there is no interaction with the information on the page URL, so the test simply instantiates the controller and confirms that no search has been executed by the constructor.

```
SearchFromURLController controller=new SearchFromURLController();
System.assertEquals(false, controller.searched);
```

The test then sets the search term, executes the search method, and confirms the results.

```
controller.name='Unit';
System.assertEquals(null, controller.executeSearch());
System.assertEquals(2, controller.accounts.size());
```

See also

- The *Testing a controller extension* recipe in this chapter shows how to write unit tests for a controller that extends a standard or custom controller.

Testing a controller extension

Controller extensions provide additional functionality for standard or custom controllers. The contract for a controller extension is that it provides a constructor that takes a single argument of the standard or custom controller that it is extending. Testing a controller extension introduces an additional requirement that an instance of the standard or custom controller, with appropriate internal state, is constructed before the controller extension.

In this recipe we will create a controller extension to retrieve the contacts associated with an account managed by a standard controller and unit test the extension.

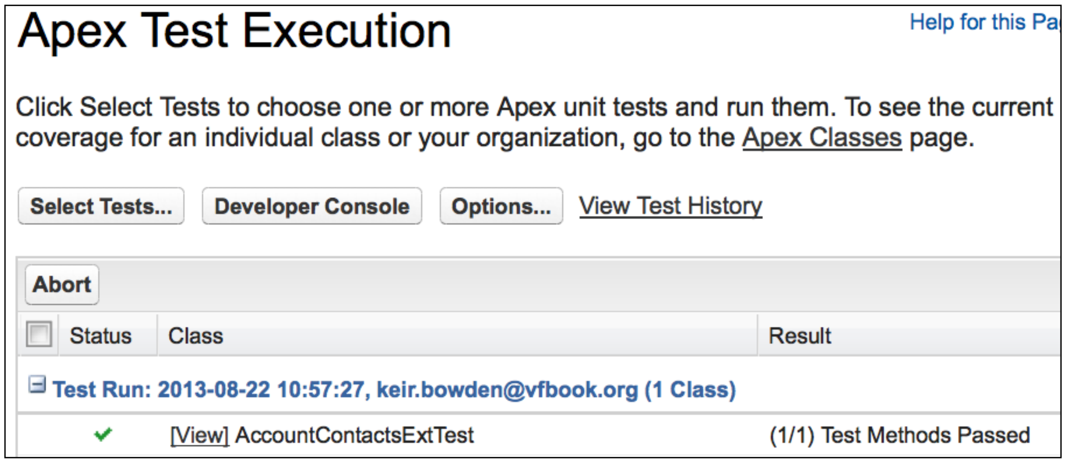
How to do it...

As the test class makes reference to the controller extension, this must be created first.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `AccountContactsExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Create the unit test class by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
6. Click on the **New** button.
7. Paste the contents of the `AccountContactsExtTest.cls` Apex class from the code download into the Apex Class area.
8. Click on the **Save** button.
9. On the resulting page, click on the **Run Tests** button.

How it works...

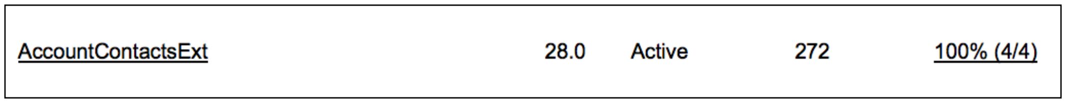
The tests successfully execute as shown in the following screenshot:



Click Select Tests to choose one or more Apex unit tests and run them. To see the current coverage for an individual class or your organization, go to the Apex Classes page.

Status	Class	Result
Test Run: 2013-08-22 10:57:27, keir.bowden@vfbook.org (1 Class)		
✓	[View] AccountContactsExtTest	(1/1) Test Methods Passed

Navigating back to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes** shows that the tests have achieved 100 percent coverage of the controller.



AccountContactsExt	28.0	Active	272	100% (4/4)

The test class contains one unit test method. As unit tests do not have access to the organization data, the first task for the test is to set up the account and contact information.

```
Account acc=new Account(Name='Unit Test');
insert acc;

List<Contact> contacts=new List<Contact>();
contacts.add(new Contact(FirstName='Unit',
    LastName='Test', Email='Unit.Test@Unit.Test',
    AccountId=acc.id));
contacts.add(new Contact(FirstName='Unit',
    LastName='Test 2', Email='Unit.Test2@Unit.Test',
    AccountId=acc.id));
insert contacts;
```

Next, the instance of the standard controller is instantiated.

```
ApexPages.StandardController std=
  new ApexPages.StandardController(acc);
```

Note that the standard controller requires the sObject record that it is managing as a parameter to the constructor. As this is the record that will be made available to the controller extension, it must have the fields populated that the extension relies upon. In this case, the only field used by the extension is the ID of the account, and this is automatically populated when the account is inserted.

In this recipe the records to be tested are created in the test classes. In the real world, this is likely to lead to a lot of repetition and a maintenance overhead. In that case, a utility class to handle the set up of test data would be a more robust solution.

Finally, the controller extension is instantiated, taking the standard controller as a constructor parameter, and the test verifies that the extension has successfully retrieved the associated contacts.

```
AccountContactsExt controller=new AccountContactsExt(std);
System.assertEquals(2, controller.contacts.size());
```

See also

- The *Testing a custom controller* recipe in this chapter shows how to write unit tests for a custom controller that does not extend or rely upon another controller.

2 Custom Components

In this chapter, we will cover the following recipes:

- Passing attributes to components
- Updating attributes in component controllers
- Passing action methods to components
- Data-driven decimal places
- The custom iterator component
- Setting a value into a controller property
- Multiselecting related objects
- Notifying the containing page controller

Introduction

Custom components allow custom Visualforce functionality to be encapsulated as discrete modules, which provides two main benefits:

- **Functional decomposition**, where a lengthy page is broken down into custom components to make it easier to develop and maintain
- **Code re-use**, where a custom component provides common functionality that can be re-used across a number of pages

A custom component may have a controller, but unlike Visualforce pages, only custom controllers may be used. A custom component can also take attributes, which can influence the generated markup or set property values in the component's controller.

Custom components do not have any associated security settings; a user with access to a Visualforce page has access to all custom components referenced by the page.

Passing attributes to components

Visualforce pages can pass parameters to components via attributes. A component declares the attributes that it is able to accept, including information about the type and whether the attribute is mandatory or optional. Attributes can be used directly in the component or assigned to properties in the component's controller.

In this recipe we will create a Visualforce page that provides contact edit capability. The page utilizes a custom component that allows the name fields of the contact, **Salutation**, **First Name**, and **Last Name**, to be edited in a three-column page block section. The contact record is passed from the page to the component as an attribute, allowing the component to be re-used in any page that allows editing of contacts.

How to do it...

This recipe does not require any Apex controllers, so we can start with the custom component.

1. Navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
2. Click on the **New** button.
3. Enter `ContactNameEdit` in the **Label** field.
4. Accept the default **ContactNameEdit** that is automatically generated for the **Name** field.
5. Paste the contents of the `ContactNameEdit.component` file from the code download into the Visualforce Markup area and click on the **Save** button.

Once a custom component is saved, it is available in your organization's component library, which can be accessed from the development footer of any Visualforce page. For more information visit `http://www.salesforce.com/us/developer/docs/pages/Content/pages_quick_start_component_library.htm`.

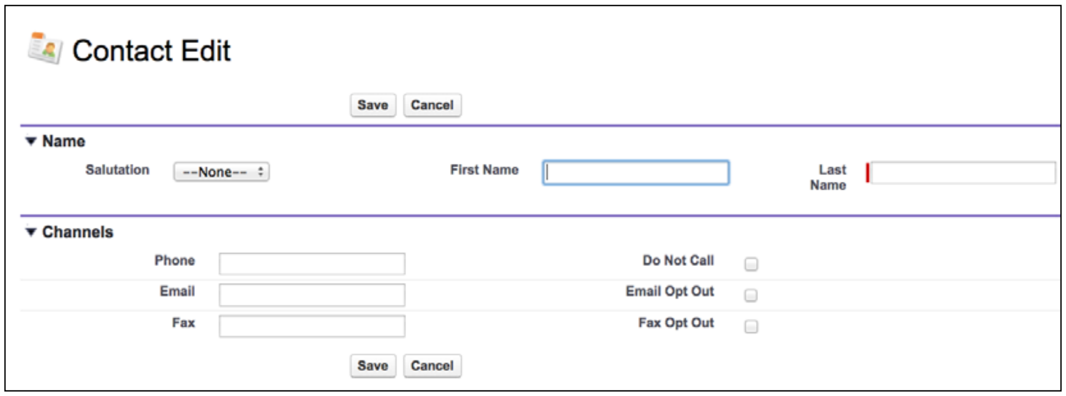
6. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `ContactEdit` in the **Label** field.
9. Accept the default **Contact Edit** that is automatically generated for the **Name** field.
10. Paste the contents of the `ContactEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **Contact Edit** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactEdit** page: `https://<instance>/apex/ContactEdit`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The custom component that renders the input fields in the **Name** section defines a single, required attribute of type `Contact`.

```
<apex:attribute name="Contact" type="Contact"
    description="The contact to edit" required="true" />
```

The description of the attribute must always be provided, as this is included in the component reference. The type of the attribute must be a primitive, sObject, one-dimensional list, map, or custom Apex class.

The `Contact` attribute can then be used in merge syntax inside the component.

```
<apex:inputField value="{!Contact.Salutation}"/>
<apex:inputField value="{!Contact.FirstName}"/>
<apex:inputField value="{!Contact.LastName}"/>
```

The page passes the contact record being managed by the standard controller to the component via the `Contact` attribute.

```
<c:ContactNameEdit contact="{!Contact}"/>
```

See also

- The *Updating attributes in component controllers* recipe in this chapter shows how a custom component can update an attribute that is a property of the enclosing page controller.

Updating attributes in component controllers

Updating fields of sObjects passed as attributes to custom components is straightforward, and can be achieved through simple merge syntax statements. This is not so simple when the attribute is a primitive and will be updated by the component controller, as parameters are passed by value, and thus, any changes are made to a copy of the primitive. For example, passing the name field of a contact sObject, rather than the contact sObject itself, would mean that any changes made in the component would not be visible to the containing page.

In this situation, the primitive must be encapsulated inside a containing class. The class instance attribute is still passed by value, so it cannot be updated to point to a different instance, but the properties of the instance can be updated.

In this recipe, we will create a containing class that encapsulates a Date primitive and a Visualforce component that allows the user to enter the date via day/month/year picklists. A simple Visualforce page and controller will also be created to demonstrate how this component can be used to enter a contact's date of birth.

Getting ready

This recipe requires a custom Apex class to encapsulate the Date primitive. To do so, perform the following steps:

1. First, create the class that encapsulates the Date primitive by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `DateContainer.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

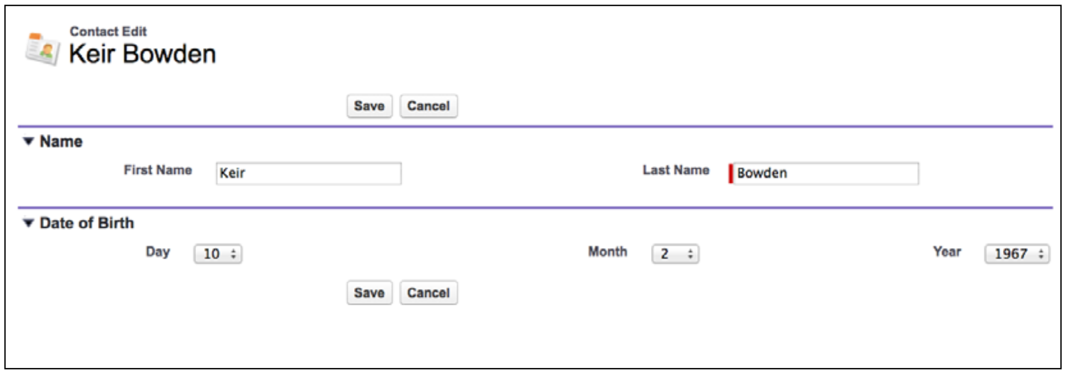
How to do it...

1. First, create the custom component controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `DateEditController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the custom component by navigating to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
6. Click on the **New** button.
7. Enter `DateEdit` in the **Label** field.
8. Accept the default **DateEdit** that is automatically generated for the **Name** field.
9. Paste the contents of the `DateEdit.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Next, create the Visualforce page controller extension by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
11. Click on the **New** button.
12. Paste the contents of the `ContactDateEditExt.cls` Apex class from the code download into the Apex Class area.
13. Click on the **Save** button.
14. Finally, create a Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Click on the **New** button.
16. Enter `ContactDateEdit` in the **Label** field.
17. Accept the default **ContactDateEdit** that is automatically generated for the **Name** field.
18. Paste the contents of the `ContactDateEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
19. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
20. Locate the entry for the `ContactDateEdit.page` file and click on the **Security** link.
21. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactDateEdit** page: `https://<instance>/apex/ContactDateEdit?id=<contact_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the ID of any contact in your Salesforce instance.



The Visualforce page controller declares a `DateContainer` property that will be used to capture the contact's date of birth.

```
public DateContainer dob {get; set;}
private Contact cont;
private ApexPages.StandardController stdCtrl {get; set;}

public ContactDateEditExt(ApexPages.StandardController std)
{
  stdCtrl=std;
  cont=(Contact) std.getRecord();
  dob=new DateContainer(cont.BirthDate);
}
```

Note that as `DateContainer` is a class, it must be instantiated when the controller is constructed.

The custom component that manages the **Date of Birth** section defines the following two attributes:

- A required attribute of type `DateContainer`, which is assigned to the `dateContainer` property of the controller

- The title of for the page block section that will house the picklists; as this is a reusable component, the page supplies an appropriate title

> Note that this component is not tightly coupled with a contact date of birth field; it may be used to manage a date field for any sObject.

```
<apex:attribute type="DateContainer" name="dateContainerAtt"
    description="The date" assignTo="{!dateContainer}"
    required="true" />
<apex:attribute type="String"
    description="Page block section title" name="title" />
```

The component controller defines properties for each of the day, month, and year elements of the date. Each setter for these properties attempts to construct the date if all of the other elements are present. This is required as there is no guarantee of the order in which the setters will be called when the **Save** button is clicked and the postback takes place.

```
public Integer year {get;
    set {
          year=value;
          updateContainer();
          }
    }

private void updateContainer()
{
  if ( (null!=year) && (null!=month) && (null!=day) )
  {
    Date theDate=Date.newInstance(year, month, day);
    dateContainer.value=theDate;
  }
}
```

When the contained date primitive is changed in the `updateContainer` method, this is reflected in the page controller property, which can then be used to update a field in the contact record.

```
public PageReference save()
{
  cont.BirthDate=dob.value;

  return stdCtrl.save();
}
```

See also

- The *Passing attributes to components* recipe in this chapter shows how an sObject may be passed as an attribute to a custom component.
- The *Adding a custom datepicker to a form* recipe in *Chapter 3*, *Capturing Data Using Forms* presents an alternative solution to capturing a date outside of the standard Salesforce range.

Passing action methods to components

A controller action method is usually invoked from the Visualforce page that it is providing the logic for. However, there are times when it is useful to be able to execute a page controller action method directly from a custom component contained within the page. One example is for styling reasons, in order to locate the command button that executes the action method inside the markup generated by the component.

In this recipe we will create a custom component that provides contact edit functionality, including command buttons to save or cancel the edit, and a Visualforce page to contain the component and supply the action methods that are executed when the buttons are clicked.

How to do it...

This recipe does not require any Apex controllers, so we can start with the custom component.

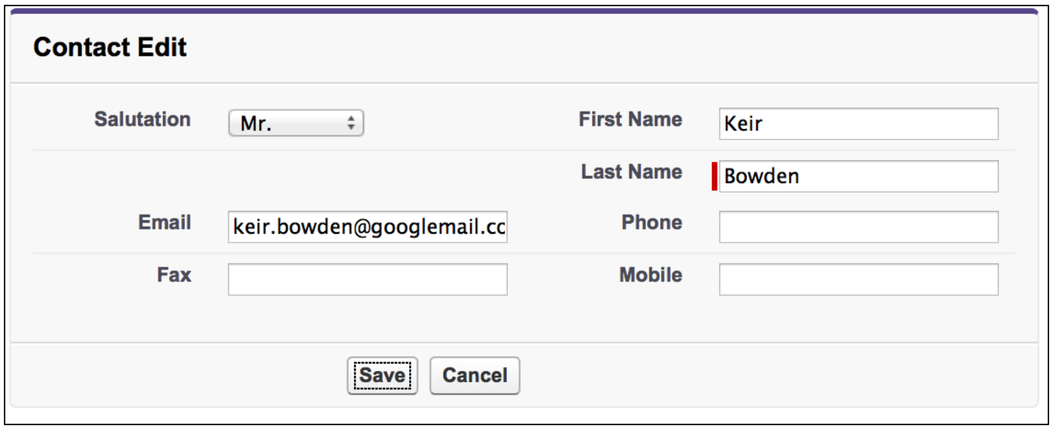
1. Navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
2. Click on the **New** button.
3. Enter `ContactEdit` in the **Label** field.
4. Accept the default **ContactEdit** that is automatically generated for the **Name** field.
5. Paste the contents of the `ContactEdit.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `ContactEditActions` in the **Label** field.
9. Accept the default **ContactEditActions** that is automatically generated for the **Name** field.
10. Paste the contents of the `ContactEditActions.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the `ContactEditActions` page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactEditActions** page: `https://<instance>/apex/ContactEditActions?id=<contact_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the ID of any contact in your Salesforce instance.


Contact Edit

Salutation	Mr.	First Name	Keir
		Last Name	Bowden
Email	keir.bowden@googlemail.cc	Phone	
Fax		Mobile	

The Visualforce page simply includes the custom component, and passes the `Save` and `Cancel` methods from the standard controller as attributes.

```
<apex:page standardController="Contact">
  <apex:pageMessages />
  <apex:form >
    <c:ContactEdit contact="{!contact}" saveAction="{!save}"
    cancelAction="{!cancel}" />
  </apex:form>
</apex:page>
```

The `ContactEdit` custom component declares attributes for the action methods of type `ApexPages.Action`.

```
<apex:attribute name="SaveAction"
    description="The save action method from the page controller"
    type="ApexPages.Action" required="true"/>
<apex:attribute name="CancelAction"
  description="The cancel action method from the page controller"
  type="ApexPages.Action" required="true"/>
```

These attributes can then be bound to the command buttons in the component in the same way as if they were supplied by the component's controller.

```
<apex:commandButton value="Save" action="{!SaveAction}" />
<apex:commandButton value="Cancel" action="{!CancelAction}"
    immediate="true" />
```

There's more...

While this example has used action methods from a standard controller, any action method can be passed to a component using this mechanism, including methods from a custom controller or controller extension.

See also

- The *Updating attributes in component controller*s recipe in this chapter shows how a custom component can update an attribute that is a property of the enclosing page controller.

Data-driven decimal places

Attributes passed to custom components from Visualforce pages can be used wherever the merge syntax is legal. The `<apex:outputText />` standard component can be used to format numeric and date values, but the formatting is limited to literal values rather than merge fields. In this scenario, an attribute indicating the number of decimal places to display for a numeric value cannot be used directly in the `<apex:outputText />` component.

In this recipe we will create a custom component that accepts attributes for a numeric value and the number of decimal places to display for the value. The decimal places attribute determines which optional component is rendered to ensure that the correct number of decimal places is displayed, and the component will also bracket negative values. A Visualforce page will also be created to demonstrate how the component can be used.

How to do it...

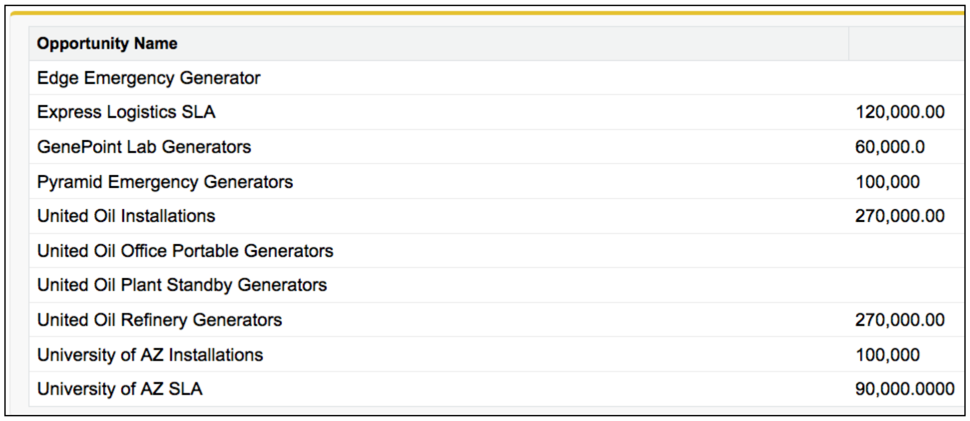
This recipe does not require any Apex controllers, so we can start with the custom component.

1. Navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
2. Click on the **New** button.
3. Enter `DecimalPlaces` in the **Label** field.
4. Accept the default **DecimalPlaces** that is automatically generated for the **Name** field.
5. Paste the contents of the `DecimalPlaces.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `DecimalPlacesDemo` in the **Label** field.
9. Accept the default **DecimalPlacesDemo** that is automatically generated for the **Name** field.
10. Paste the contents of the `DecimalPlacesDemo.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
11. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Locate the entry for the **DecimalPlacesDemo** page and click on the **Security** link.
13. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **DecimalPlacesDemo** page: `https://<instance>/apex/DecimalPlacesDemo`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Opportunity Name	
Edge Emergency Generator	
Express Logistics SLA	120,000.00
GenePoint Lab Generators	60,000.0
Pyramid Emergency Generators	100,000
United Oil Installations	270,000.00
United Oil Office Portable Generators	
United Oil Plant Standby Generators	
United Oil Refinery Generators	270,000.00
University of AZ Installations	100,000
University of AZ SLA	90,000.0000

The Visualforce page iterates a number of opportunity records and delegates to the component to output the opportunity amount, deriving the number of decimal places from the amount.

```
<c:DecimalPlaces dp="{!TEXT(MOD(opp.Amount/10000, 5))}"
  value="{!opp.Amount}" />
```

The component conditionally renders the appropriate output panel, which contains two conditionally rendered `<apex:outputText />` components, one to display a positive value to the correct number of decimal places and another to display a bracketed negative value.

```
<apex:outputPanel rendered="{!dp=='1'}">
  <apex:outputText rendered="{!AND(NOT(ISNULL(VALUE)), value>=0)}"
                value="{0, number, #,##0.0}">
    <apex:param value="{!value}"/>
  </apex:outputText>
  <apex:outputText rendered="{!AND(NOT(ISNULL(VALUE)), value<0)}"
                value="({0, number, #,##0.0})">
    <apex:param value="{!ABS(value)}"/>
  </apex:outputText>
</apex:outputPanel>
```

See also

- The *Data-driven styling* recipe in *Chapter 1*, *General Utilities* shows how to conditionally color a numeric value based on whether it is positive or negative.

The custom iterator component

The Visualforce standard component `<apex:repeat />` iterates a collection of data and outputs the contained markup once for each element in the collection. In the scenario where this is being used to display a table of data, the markup for the table headings appears before the `<apex:repeat />` component and is rendered regardless of whether the collection contains any records or not.

Custom iterator components may contain additional markup to be rendered outside the collection, for example, the headings markup in the scenario mentioned earlier. This allows the component to avoid rendering any markup if the collection is empty through the logic implemented in a custom controller.

In this recipe, we will create a custom component that takes a collection of data and renders a containing page block and a page block section for each element in the collection. If the collection is empty, no markup will be rendered. We will also create a containing page that displays the details of an sObject account and utilizes the custom component to display contact and opportunity information, if present.

Getting ready

This recipe makes use of a custom controller, so this will need to be present before the custom component can be created.

How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `AllOrNothingController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
6. Click on the **New** button.
7. Enter `AllOrNothingPageBlock` in the **Label** field.

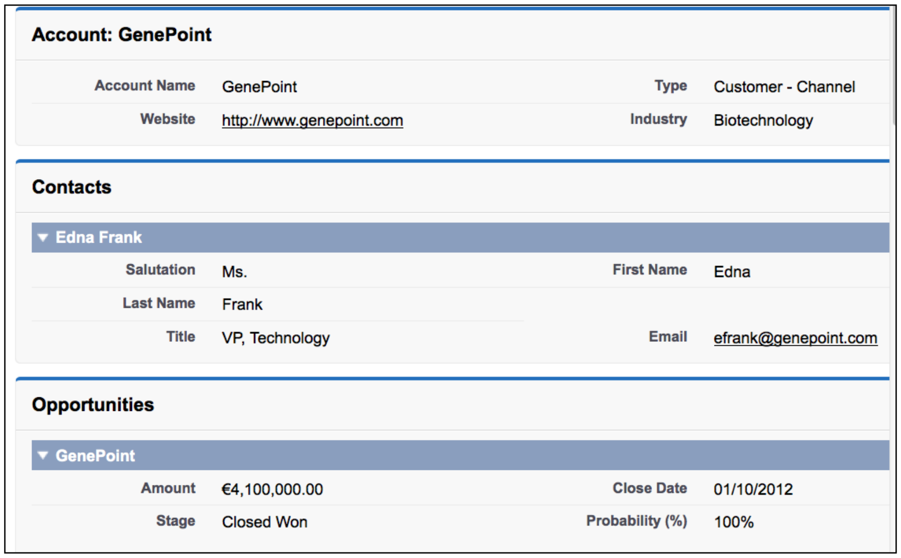
8. Accept the default **AllOrNothingPageBlock** that is automatically generated for the **Name** field.
9. Paste the contents of the `AllOrNothingPageBlock.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Next, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
11. Click on the **New** button.
12. Paste the contents of the `AllOrNothingListsExt.cls` Apex class from the code download into the Apex Class area.
13. Click on the **Save** button.
14. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Click on the **New** button.
16. Enter `AllOrNothingLists` in the **Label** field.
17. Accept the default **AllOrNothingLists** that is automatically generated for the **Name** field.
18. Paste the contents of the `AllOrNothingLists.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
19. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
20. Locate the entry for the **AllOrNothingLists** page and click on the **Security** link.
21. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **AllOrNothingLists** page: `https://<instance>/apex/AllOrNothingLists?id=<account_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<account_id>` is the ID of an account from your Salesforce instance.

The page displays brief details of the account and page blocks for the account's contacts and opportunities.



Account: GenePoint

Account Name	GenePoint	Type	Customer - Channel
Website	http://www.genepoint.com	Industry	Biotechnology

Contacts

▼ Edna Frank

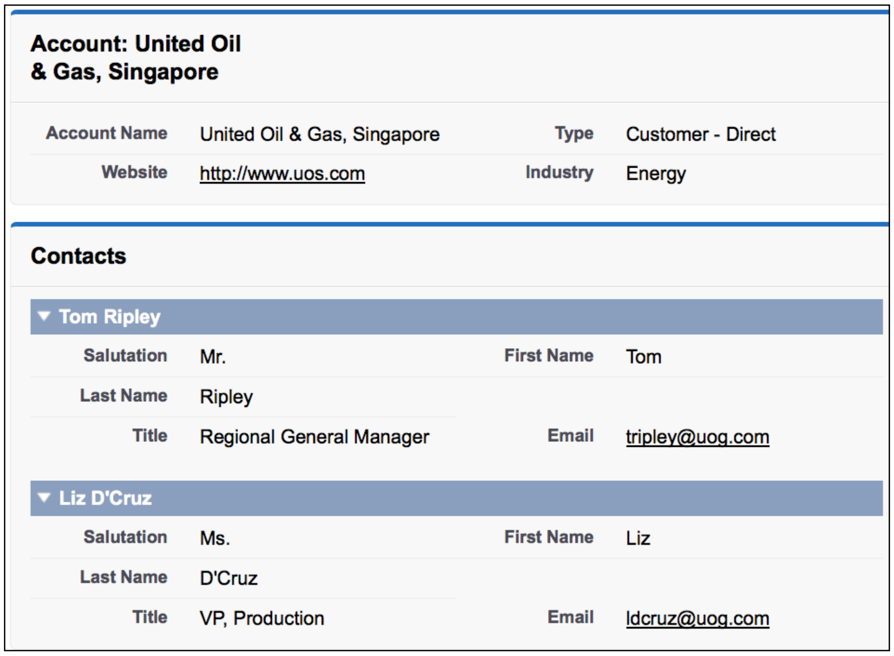
Salutation	Ms.	First Name	Edna
Last Name	Frank		
Title	VP, Technology	Email	efrank@genepoint.com

Opportunities

▼ GenePoint

Amount	€4,100,000.00	Close Date	01/10/2012
Stage	Closed Won	Probability (%)	100%

If the account does not have any opportunities or contacts, the appropriate page block is omitted.



Account: United Oil & Gas, Singapore

Account Name	United Oil & Gas, Singapore	Type	Customer - Direct
Website	http://www.uos.com	Industry	Energy

Contacts

▼ Tom Ripley

Salutation	Mr.	First Name	Tom
Last Name	Ripley		
Title	Regional General Manager	Email	tripley@uog.com

▼ Liz D'Cruz

Salutation	Ms.	First Name	Liz
Last Name	D'Cruz		
Title	VP, Production	Email	ldcruz@uog.com

The `AllOrNothingPageBlock` custom component accepts a list of generic sObjects and wraps its content in an output panel that is only rendered if the list is not empty.

```
<apex:outputPanel rendered="{!render}">
      <apex:pageBlock title="{!title}">
        <apex:repeat value="{!list}" var="ele">
          <apex:componentBody >
            <apex:variable var="{!var}" value="{!ele}"/>
          </apex:componentBody>
        </apex:repeat>
      </apex:pageBlock>
  </apex:outputPanel>
```

The `<apex:componentBody />` defines where the content from the Visualforce page will be inserted. As this is inside the `<apex:repeat />` component, this insertion will take place for each element in the collection.

The Visualforce page passes attributes to the component of the list to iterate the title of the page block, and the variable name representing an element in the list. The contained markup can reference the variable name to access fields or properties of the element.

```
<c:AllOrNothingPageBlock list="{!opportunities}" var="opp"
       title="Opportunities">
<apex:pageBlockSection title="{!opp.Name}">
      <apex:outputField value="{!opp.Amount}" />
      <apex:outputField value="{!opp.CloseDate}" />
      <apex:outputField value="{!opp.StageName}" />
      <apex:outputField value="{!opp.Probability}" />
</apex:pageBlockSection>
</c:AllOrNothingPageBlock>
```

Setting a value into a controller property

Visualforce controllers are often re-used across pages with minor variations in behavior specific to the page, for example, displaying accounts of a particular type. While the controller can detect the page that it is being used by and alter its behavior accordingly, this is not a particularly maintainable solution, as use of the controller in any new page would require changes to the Apex code and renaming a page would break the functionality.

A better mechanism is for the page to set the values of properties in the controller to indicate the desired behavior. In this recipe we will create a custom component that takes two attributes: a value and the controller property to set the value into. Two Visualforce pages with a common controller will also be created to demonstrate how the component can be used to change the behavior of the controller to suit the page.

Getting ready

This recipe does not require any Apex controllers, so we can start with the custom component.

How to do it...

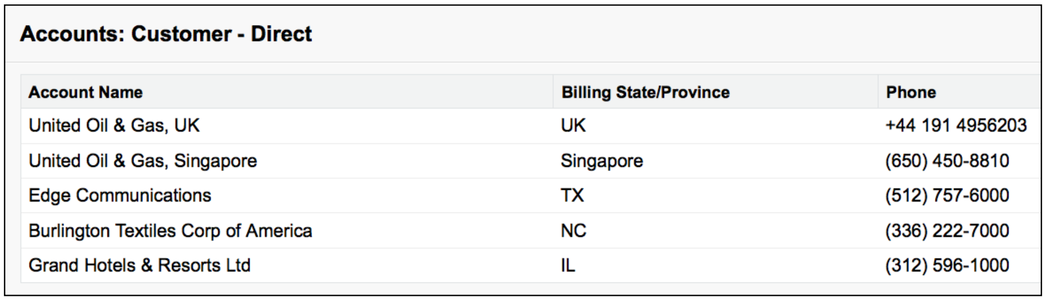
1. Navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
2. Click on the **New** button.
3. Enter `SetControllerProperty` in the **Label** field.
4. Accept the default **SetControllerProperty** that is automatically generated for the **Name** field.
5. Paste the contents of the `SetControllerProperty.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the custom controller for the Visualforce pages by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
7. Click on the **New** button.
8. Paste the contents of the `AccountsTypeController.cls` Apex class from the code download into the Apex Class area.
9. Click on the **Save** button.
10. Next, create the first Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Click on the **New** button.
12. Enter `AccountsType1` in the **Label** field.
13. Accept the default **AccountsType1** that is automatically generated for the **Name** field.
14. Paste the contents of the `AccountsType1.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
15. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
16. Locate the entry for the **AccountsType1** page and click on the **Security** link.
17. On the resulting page, select which profiles should have access and click on the **Save** button.
18. Finally, create the second Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
19. Click on the **New** button.

20. Enter `AccountsType2` in the **Label** field.
21. Accept the default **AccountsType2** that is automatically generated for the **Name** field.
22. Paste the contents of the `AccountsType2.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
23. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
24. Locate the entry for the **AccountsType2** page and click on the **Security** link.
25. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the first Visualforce page in your browser displays a list of accounts whose type is **Customer - Direct**: `https://<instance>/apex/AccountsType1`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

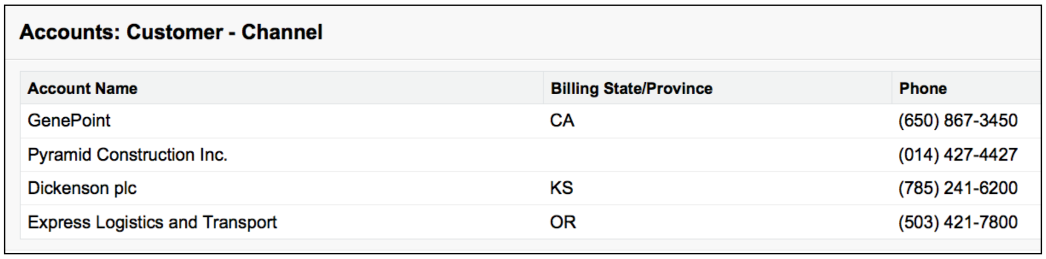


Accounts: Customer - Direct

Account Name	Billing State/Province	Phone
United Oil & Gas, UK	UK	+44 191 4956203
United Oil & Gas, Singapore	Singapore	(650) 450-8810
Edge Communications	TX	(512) 757-6000
Burlington Textiles Corp of America	NC	(336) 222-7000
Grand Hotels & Resorts Ltd	IL	(312) 596-1000

While, opening the second Visualforce page displays a list of accounts whose type is **Customer - Channel**: `https://<instance>/apex/AccountsType2`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Accounts: Customer - Channel

Account Name	Billing State/Province	Phone
GenePoint	CA	(650) 867-3450
Pyramid Construction Inc.		(014) 427-4427
Dickenson plc	KS	(785) 241-6200
Express Logistics and Transport	OR	(503) 421-7800

The `SetControllerProperty` custom component assigns the value attribute to the controller property attribute:

```
<apex:component >
    <apex:attribute name="from" type="String" assignTo="{!to}"
      description="The value to set"/>
    <apex:attribute name="to" type="String"
      description="The controller property to set the value into"/>
</apex:component>
```

The Visualforce pages set the type of account to be retrieved into the controller property via the custom component.

```
<c:SetControllerProperty from="Customer - Direct"
to="{!accType}" />
```

See also

- The *Updating attributes in component controllers* recipe in this chapter shows how a custom component can update an attribute that is a property of the enclosing page controller.
- The *Passing attributes to components* recipe in this chapter shows how an sObject may be passed as an attribute to a custom component.

Multiselecting related objects

One task that users often find unwieldy when implementing Salesforce is setting up the sObjects to represent many-to-many relationships. A junction object allows a single instance of one sObject type to be related to multiple instances of another sObject type and vice versa.

This requires the user to create a new instance of the junction object and populate master-detail fields to associate two sObjects with each other, resulting in a large number of clicks and page transitions.

In this recipe, we will create a custom object – account group – that acts as a container for multiple accounts. We will then create a page that allows a number of accounts to be associated with a single custom sObject. We will use junction objects for the relationship to allow a single account to be related to multiple account groups, and a single account group to be associated with multiple accounts. A custom Visualforce component will manage the action of presenting the available accounts and allowing the user to choose which to relate with the account group. The component will use the mechanism described in the *Updating attributes in component controllers* recipe to make the selected values available to the page controller via a custom string container class.

Getting ready

This recipe requires two custom sObjects: the account group, and the junction object between an account group and an account.

1. First, create the account group custom sObject by navigating to **Your Name** | **Setup** | **Develop** | **Objects**.
2. Click on the **New Custom Object** button.
3. Enter `Account Group` in the **Label** field.
4. Enter `Account Groups` in the **Plural Label** field.
5. Select the **Starts with vowel sound** box.
6. Leave all other input values at their defaults and click on the **Save** button.
7. Next, create the junction object to associate an account group with an account by navigating to **Your Name** | **Setup** | **Develop** | **Objects**.
8. Click on the **New Custom Object** button.
9. Enter `Account Group JO` in the **Label** field.
10. Enter `Account Group JOs` in the **Plural Label** field.
11. Select the **Starts with vowel sound** box.
12. Leave all other input values at their defaults and click on the **Save** button.
13. On the resulting page, create the master-detail relationship for the account group by scrolling down to the **Custom Fields and Relationships** section and clicking on the **New** button.
14. On the next page, **Step 1. Choose the field type**, select the **Master-Detail Relationship** from the **Data Type** radio buttons and click on the **Next** button.
15. On the next page, **Step 2. Choose the related object**, choose **Account Group** from the **Related To** picklist and click on the **Next** button.
16. On the next page, **Step 3. Enter the label and name for the lookup field**, leave all the fields at their default values and click on the **Next** button.
17. On the next page, **Step 4. Establish field-level security for reference field**, leave all the fields at their default values and click on the **Next** button.
18. On the next page, **Step 5. Add reference field to page layouts**, leave all the fields at their default values and click on the **Next** button.
19. On the final page, **Step 6. Add Custom Related Lists**, leave all the fields at their default values and click on the **Save** button.
20. Next, create the master-detail relationship for the account by scrolling down to the **Custom Fields and Relationships** section and click on the **New** button.
21. On the next page, **Step 1. Choose the field type**, select the **Master-Detail Relationship** from the **Data Type** radio buttons and click on the **Next** button.

22. On the next page, **Step 2. Choose the related object**, choose **Account** from the **Related To** picklist and click on the **Next** button.
23. On the next page, **Step 3. Enter the label and name for the lookup field**, leave all the fields at their default values and click on the **Next** button.
24. On the next page, **Step 4. Establish field-level security for reference field**, leave all the fields at their default values and click on the **Next** button.
25. On the next page, **Step 5. Add reference field to page layouts**, leave all the fields at their default values and click on the **Next** button.
26. On the final page, **Step 6. Add Custom Related Lists**, leave all the fields at their default values and click on the **Save** button.

How to do it...

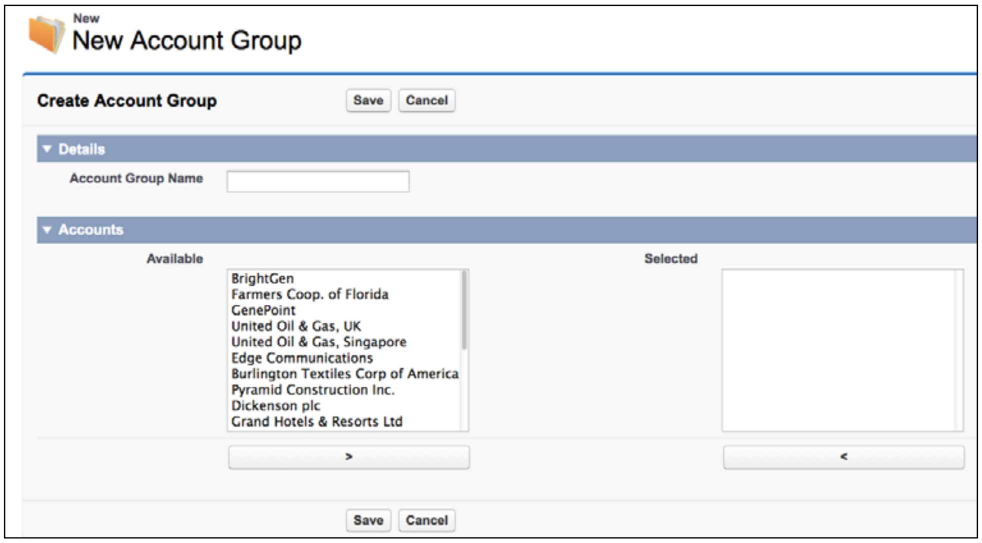
1. Create the custom string container class by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `StringContainer.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the custom controller for the Visualforce component by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
6. Click on the **New** button.
7. Paste the contents of the `MultiSelectRelatedController.cls` Apex class from the code download into the Apex Class area.
8. Click on the **Save** button.
9. Next, navigate to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
10. Click on the **New** button.
11. Enter `MultiSelectRelated` in the **Label** field.
12. Accept the default **MultiSelectRelated** that is automatically generated for the **Name** field.
13. Paste the contents of the `MultiSelectRelated.component` file from the code download into the Visualforce Markup area and click on the **Save** button.
14. Next, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
15. Click on the **New** button.

16. Paste the contents of the `AccountGroupController.cls` Apex class from the code download into the Apex Class area.
17. Click on the **Save** button.
18. Finally, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
19. Click on the **New** button.
20. Enter `AccountGroup` in the **Label** field.
21. Accept the default **AccountGroup** that is automatically generated for the **Name** field.
22. Paste the contents of the `AccountGroup.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
23. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
24. Locate the entry for the **AccountGroup** page and click on the **Security** link.
25. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the Visualforce page in your browser displays the account group create page: `https://<instance>/apex/AccountGroup`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



BrightGen
Farmers Coop. of Florida
GenePoint
United Oil & Gas, UK
United Oil & Gas, Singapore
Edge Communications
Burlington Textiles Corp of America
Pyramid Construction Inc.
Dickenson plc
Grand Hotels & Resorts Ltd

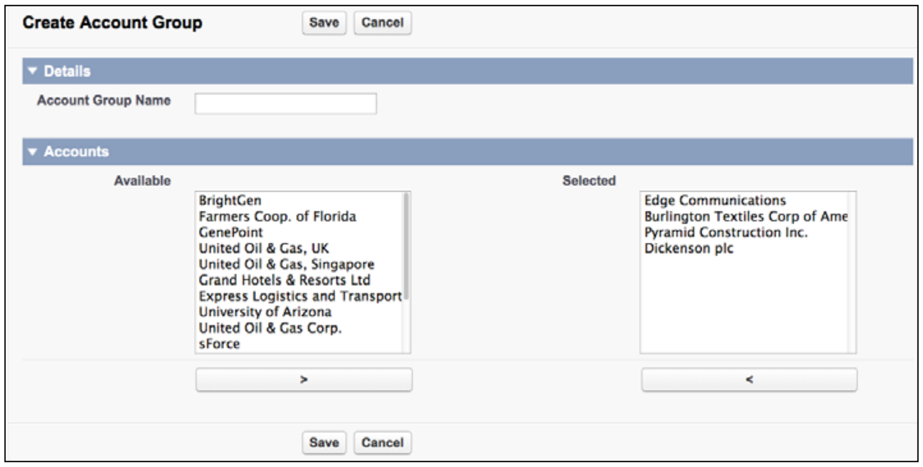
The chosen account IDs are stored as semi-colon separated values in a property of the page controller. The page controller constructor extracts all accounts and creates a standard `SelectOption` class for each one.

```
accountOptions=new List<SelectOption>();
for (Account acc : [select id, Name from Account])
{
  accountOptions.add(new SelectOption(acc.id, acc.name));
}
```

The custom component controller contains two collections of options: **available** and **chosen**. The component iterates the full set of account options and adds the option to available or chosen depending on whether the account ID is present in the semi-colon separated string of chosen IDs.

```
availableItems=new List<SelectOption>();
chosenItems=new List<SelectOption>();

for (SelectOption sel : allOptions)
{
  String selId=sel.getValue();
  if (selected.value.contains(selId+';'))
  {
    chosenItems.add(sel);
  }
  else
  {
    availableItems.add(sel);
  }
}
```

Accounts can be moved between the **Available** and **Selected** lists by highlighting the options and clicking on the **>** or **<** buttons.



Available	Selected
BrightGen	Edge Communications
Farmers Coop. of Florida	Burlington Textiles Corp of Ame
GenePoint	Pyramid Construction Inc.
United Oil & Gas, UK	Dickenson plc
United Oil & Gas, Singapore	
Grand Hotels & Resorts Ltd	
Express Logistics and Transport	
University of Arizona	
United Oil & Gas Corp.	
sForce	

Each button invokes an action method in the component controller that adds or removes the account IDs from the semi-colon separated chosen string, and then rebuilds the available and chosen lists.

Clicking on the **Save** button creates the account group record and a junction object for each account ID in the chosen string.

```
public PageReference save()
{
  insert accountGroup;
  List<Account_Group_JO__c> agJOs=
        new List<Account_Group_JO__c>();

  for (String accId : chosenAccounts.value.split(';'))
  {
    Account_Group_JO__c agJO=
      new Account_Group_JO__c(
          Account_Group__c=accountGroup.id,
          Account__c=accId);
    agJOs.add(agJO);
  }

  insert agJOs;

  return new PageReference('/' + accountGroup.id);
}
```

Notifying the containing page controller

In the earlier recipes, we have seen how components can accept an attribute that is a property from the containing page controller and update the value of the property in response to a user action. If the containing page controller needs to determine if the property has changed, it must capture the previous value of the property and compare that with the current value. The same applies if the attribute passed to the component is a field from an sObject managed by the parent page controller.

In this recipe we will create a custom component that can notify its containing page controller when an attribute value is changed. In order to avoid tying the component to a particular page controller class, we will create an interface that defines the method to be used to notify the page controller. This will allow the component controller to notify any page controller that implements the interface.

Interfaces define a "contract" between the calling code and the implementing code. The calling code is able to rely on the method(s) defined in the interface being available without having to know the details of the underlying code that is implementing the interface. This allows the implementing code to be swapped in and out without affecting the calling code.

Getting ready

This recipe requires that you have already completed the *Multiselecting related objects* recipe, as it relies on the custom sObjects created in that recipe.

How to do it...

1. First, create the interface by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `Notifiable.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the component controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
6. Click on the **New** button.
7. Paste the contents of the `NotifyingMultiSelectRelatedController.cls` Apex class from the code download into the Apex Class area.
8. Click on the **Save** button.
9. Next, create the custom component by navigating to the Visualforce Components setup page by clicking on **Your Name** | **Setup** | **Develop** | **Components**.
10. Click on the **New** button.
11. Enter `NotifyingMultiSelectRelated` in the **Label** field.
12. Accept the default **NotifyingMultiSelectRelated** that is automatically generated for the **Name** field.
13. Paste the contents of the `NotifyingMultiSelectRelated.component` file from the code download into the Visualforce Markup area and click on the **Save** button.

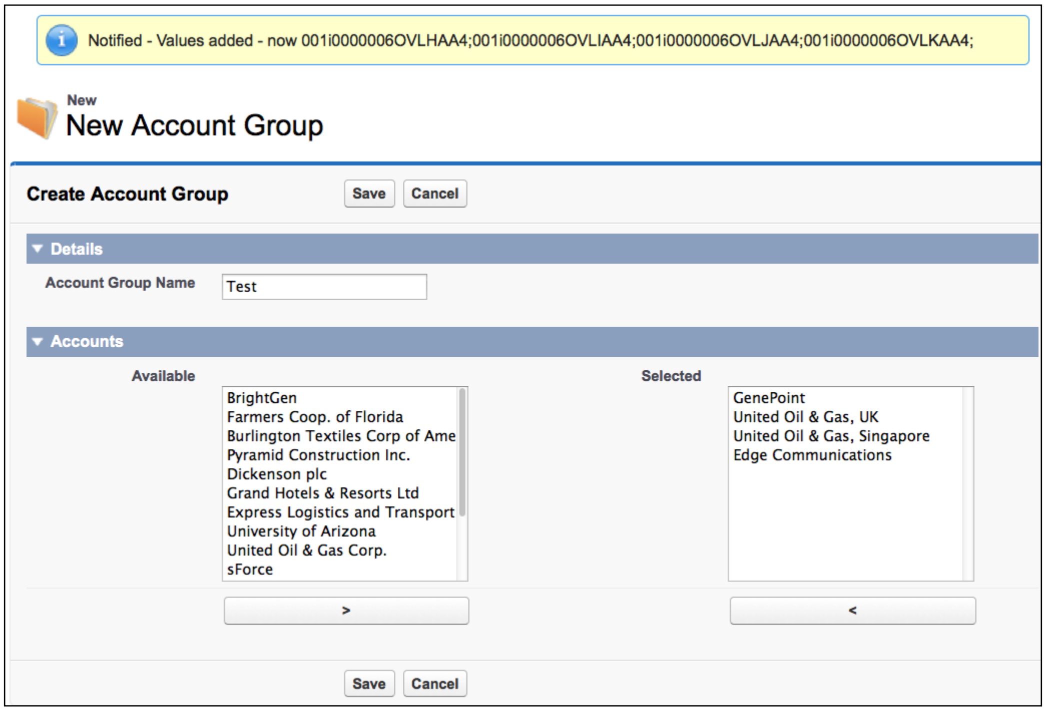
14. Next, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
15. Click on the **New** button.
16. Paste the contents of the `NotifiableAccountGroupController.cls` Apex class from the code download into the Apex Class area.
17. Click on the **Save** button.
18. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
19. Click on the **New** button.
20. Enter `NotifiableAccountGroup` in the **Label** field.
21. Accept the default **NotifiableAccountGroup** that is automatically generated for the **Name** field.
22. Paste the contents of the `NotifiableAccountGroup.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
23. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
24. Locate the entry for the **NotifiableAccountGroup** page and click on the **Security** link.
25. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **NotifiableAccountGroup** page: `https://<instance>/apex/NotifiableAccountGroup`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<account_id>` is the ID of an account from your Salesforce instance.

When accounts are moved between the **Available** and **Selected** lists, the component controller notifies the parent page controller, which writes a message into the page with the notification details.



Notified - Values added - now 001i0000006OVLHAA4;001i0000006OVLIAA4;001i0000006OVLJAA4;001i0000006OVLKAA4;

New
New Account Group

Create Account Group

Details

Account Group Name: Test

Accounts

Available:
BrightGen
Farmers Coop. of Florida
Burlington Textiles Corp of Ame
Pyramid Construction Inc.
Dickenson plc
Grand Hotels & Resorts Ltd
Express Logistics and Transport
University of Arizona
United Oil & Gas Corp.
sForce

Selected:
GenePoint
United Oil & Gas, UK
United Oil & Gas, Singapore
Edge Communications

The parent page controller, `NotifiableAccountGroupController`, implements the `Notifiable` interface.

```
public with sharing class NotifiableAccountGroupController
          implements Notifiable
```

The implementation of the `notify` method wraps the notification text in an `ApexPages.Message` class and adds this to the messages for the page.

```
public void notify(String detail)
{
  ApexPages.addMessage(new ApexPages.Message(
    ApexPages.Severity.INFO, 'Notified - ' + detail));
}
```

The custom component takes a parameter of type `Notifiable`, which is assigned to a property in its controller.

```
<apex:attribute name="notify"
   description="The entity to notify when the selection changes"
   type="Notifiable" assignTo="{!notifiable}" />
```

The custom component also takes a `rerender` parameter, which is used to update part of the containing page when the selection changes.

```
<apex:attribute name="rerender"
description="The component to rerender when the selection changes"
   type="String" />
```

In this case, the page passes through the ID of its `<apex:pageMessages />` component to display the message added in the page controller's `notify` method as described earlier.

When the action methods to move accounts between the **Available** and **Selected** lists are executed, these invoke the `notify` method with details of the change.

```
if (null!=notifiable)
{
  notifiable.notify('Values deleted - now ' + selected.value);
}
```

See also

- The *Updating attributes in component controllers* recipe in this chapter shows how a custom component can update an attribute that is a property of the enclosing page controller.
- The *Passing attributes to components* recipe in this chapter shows how an sObject may be passed as an attribute to a custom component.
- The *Multiselecting related objects* in this chapter shows how to create a custom multiselect picklist style component.

3
Capturing Data Using Forms

In this chapter, we will cover the following recipes:

- Editing a record in Visualforce
- Adding error messages to field inputs
- Adding error messages to nonfield inputs
- Using field sets
- Adding a custom lookup to a form
- Adding a custom datepicker to a form
- Retrieving fields when a lookup is populated
- Breaking up forms with action regions
- The "Please wait" spinner
- Avoiding validation errors with action regions
- Action chaining
- Errors – harmful if swallowed

Introduction

Forms are a key feature of any application that makes use of Visualforce. They provide a mechanism to capture data entered by the user and send this to the page controller for processing, for example, to create, edit, or delete sObject records, or to send the user to a specific page.

Users enter data through input components. Visualforce provides a specific standard component, `<apex:inputField />`, for entering sObject field data. This component renders the appropriate device for entering data based on the field type, such as a JavaScript date picker for a field of type `Date`. Input components are bound to sObject fields or controller properties via the merge syntax. Controller properties that are public and have a public getter and setter may be bound to input components without writing any further code.

Processing of the submitted form is carried out via Action methods. These may be provided automatically by the platform in the case of standard controllers, or coded using Apex in the case of extension or custom controllers. Action methods can rely on all sObject fields and controller properties bound to input components containing the latest user input when they execute.

Editing a record in Visualforce

The standard record edit page does not allow customization outside the layout of fields. Editing records with Visualforce pages allows customization of all aspects of the page, including styling, content, and displayed buttons.

In this recipe, we will create a Visualforce page that provides contact edit capability, but does not allow a contact to be reparented to a different account. The account lookup field is editable until the record is saved with the lookup populated, after which it becomes read-only. This page will also render a different section heading depending on whether the contact is being created or edited.

Getting ready

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

How to do it...

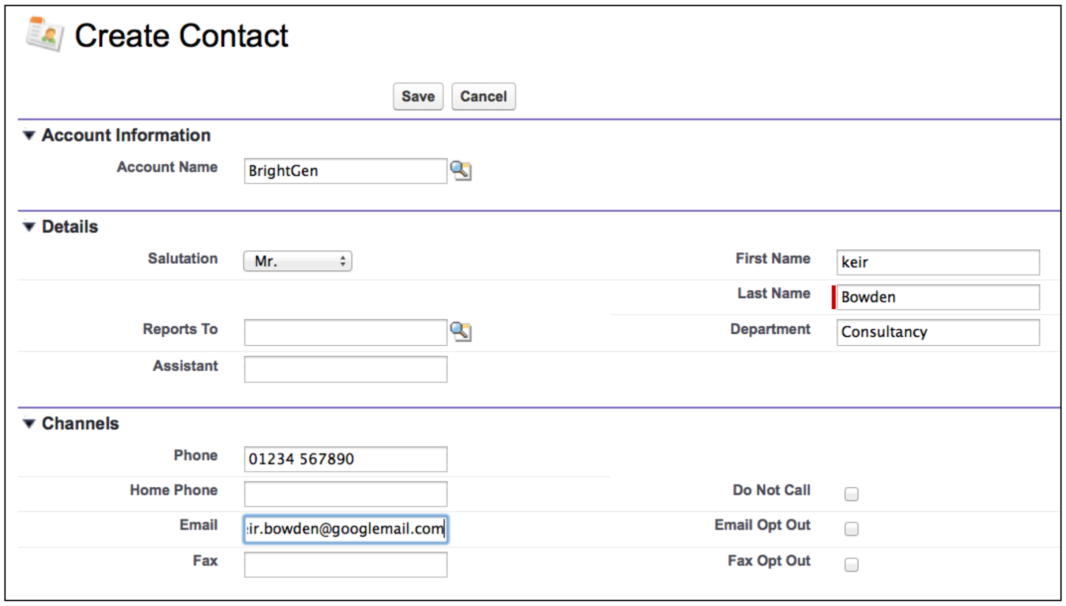
1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `ContactCreateEdit` in the **Label** field.
4. Accept the default **ContactCreateEdit** that is automatically generated for the **Name** field.
5. Paste the contents of the `ContactCreateEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

7. Locate the entry for the **ContactCreateEdit** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactCreateEdit** page to create a new record: `https://<instance>/apex/ContactCreateEdit`.

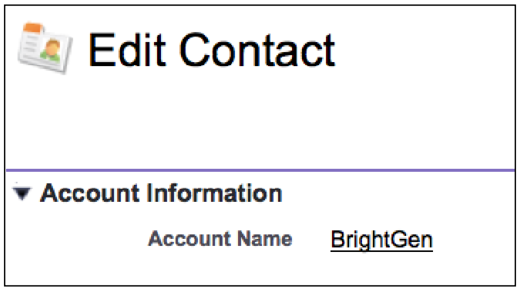
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Account Name	BrightGen		
Salutation	Mr.	First Name	keir
		Last Name	Bowden
Reports To		Department	Consultancy
Assistant			
Phone	01234 567890		
Home Phone		Do Not Call	☐
Email	ir.bowden@googlemail.com	Email Opt Out	☐
Fax		Fax Opt Out	☐

Note that the section heading for the page is **Create Contact**, and the **Account Name** field is editable. Save this record and edit it again via the same page using the following URL: `https://<instance>/apex/ContactCreateEdit?id=<contact_id>`.

Here, `<contact_id>` is the record ID of the newly created contact, and it displays the edit form of the page with a different section heading, and the **Account Name** field is changed to read-only.



The Visualforce page utilizes a standard controller and conditionally renders the section heading, and input/output variants of the Account Name field based on the ID of the record.

```
<apex:sectionHeader
    title="{!IF (Contact.id==null, 'Create', 'Edit')} Contact" />

<apex:inputField value="{!Contact.AccountId}"
        rendered="{!null==Contact.AccountId}" />
<apex:outputField value="{!Contact.AccountId}"
        rendered="{!null!=Contact.AccountId}" />
```

See also

- The *Using field sets* recipe in this chapter shows how an administrator can control the editable fields on a Visualforce page.

Adding error messages to field inputs

When users are editing or creating a record via a Visualforce page, they will often make mistakes or enter invalid data. The required fields will present an error message underneath the field itself, but validation rules or exceptions will simply send the user to a new page with a large error message, telling them that the insert or update failed.

In this recipe we will create a Visualforce page to allow a user to create or edit a contact record. The contact standard controller and a controller extension manage the page. The extension controller checks whether the e-mail address or phone number field has been populated. If either of the fields is populated, the record will be saved, but if both are missing, an error message will be added to both fields asking the user to populate at least one of them.

Getting ready

This recipe makes use of a controller extension, so this will need to be created before the Visualforce page.

How to do it...

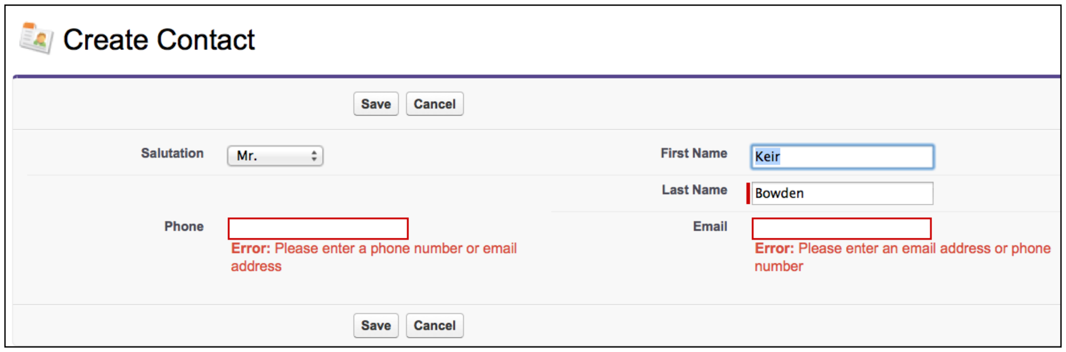
1. First, create the controller extension by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `InputFieldErrorExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `InputFieldError` in the **Label** field.
8. Accept the default **InputFieldError** that is automatically generated for the **Name** field.
9. Paste the contents of the `InputFieldError.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **InputFieldError** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **InputFieldError** page: `https://<instance>/apex/InputFieldError`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Leaving both the **Phone** and **Email** fields blank causes a tailored error message to be displayed underneath the fields.



Salutation	Mr.	First Name	Keir
		Last Name	Bowden
Phone	**Error:** Please enter a phone number or email address	Email	**Error:** Please enter an email address or phone number

The Visualforce page controller extension declares a `Save` method, which overrides the standard controller's `Save` method. If the contact's Phone and Email fields are both empty, the controller uses the `addError` method of the sObject field to associate an error message with the field. When the page is rendered, Visualforce will automatically display the error message with the field. If either of the **Phone** or **Email** fields are populated, the controller extension delegates it to the standard controller `Save` method and returns the result.

```
public PageReference save()
{
  PageReference result=null;
  Contact cont=(Contact) stdCtrl.getRecord();

  if ( (String.IsBlank(cont.Email)) &&
        (String.IsBlank(cont.Phone)) )
{
    cont.email.addError
      ('Please enter an email address or phone number');
    cont.phone.addError
      ('Please enter a phone number or email address');
  }
  else
  {
    result=stdCtrl.save();
  }

  return result;
}
```

See also

- The *Adding error messages to nonfield inputs* recipe in this chapter shows how error messages can be displayed against input elements associated with controller properties.

Adding error messages to nonfield inputs

In the previous recipe, *Adding error messages to field inputs*, the platform took care of positioning of the error message based on whether the field had any errors associated with it. Visualforce automatically provides this functionality for `<apex:inputField />` components, but if a different input component is used, such as `<apex:inputText />` or `<apex:selectList />`, there is no equivalent functionality.

In this recipe we will create a Visualforce page to allow a user to create or edit a contact record. The contact standard controller and a controller extension manage the page. The ID of the account that the contact is associated with is entered via an `<apex:selectList />` component, which is bound to a controller property rather than an sObject field. If the user does not select an account to associate the contact with, an error message is displayed under the `<apex:selectList />` component.

Getting ready

This recipe makes use of a controller extension, so this will need to be created before the Visualforce page.

How to do it...

1. First, create the Visualforce page controller extension by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `InputSelectErrorExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `InputSelectError` in the **Label** field.
8. Accept the default **InputSelectError** that is automatically generated for the **Name** field.

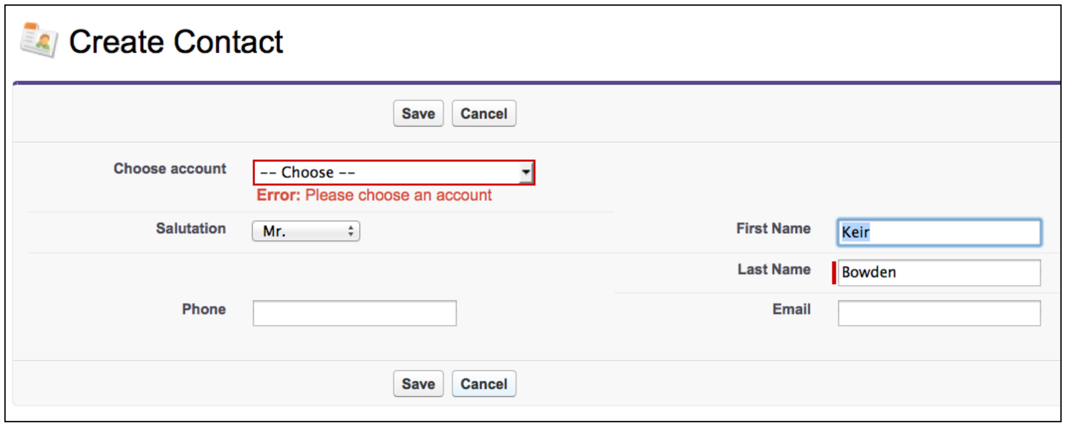
9. Paste the contents of the `InputSelectError.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **InputSelectError** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **InputSelectError** page: `https://<instance>/apex/InputSelectError`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Leaving the selected account value as **-- Choose --** causes the save to fail and an error message to be displayed under the select component.



Choose account	-- Choose -- Error: Please choose an account		
Salutation	Mr.	First Name	Keir
		Last Name	Bowden
Phone		Email	

In order to display the error message in the correct place, it must be coupled with the property that the input component is bound to. The controller extension defines an inner class that contains two `String` properties; one to hold the input value and another to hold the error message. The class also exposes a method to determine if the input value is currently in error.

```
public class ValueAndError
{
  public String value {get; set;}
  public String error {get; set;}

  public Boolean getHasError()
```

```
    {
      return (!String.IsBlank(error));
    }
  }
```

An instance of this class is used by the controller extension to contain the ID of the selected account.

```
public ValueAndError accountIdVal {get; set;}
```

The controller extension defines a `Save` method, which overrides the standard controller save method. If the `value` property of the `ValueAndError` instance remains at the default, an error message is added to the instance.

```
  if (String.IsBlank(accountIdVal.value))
  {
    accountIdVal.error='Please choose an account';
  }
```

The Visualforce page conditionally adds the `error` style class to the input component if the `ValueAndError` instance indicates there is an error. Additionally, the actual error message is displayed using a style class of `errorMsg`.

```
<apex:selectList value="{!accountIdVal.value}" size="1"
  styleClass="{!IF(accountIdVal.hasError,'error','')}">
  <apex:selectOptions value="{!accountOptions}" />
</apex:selectList>
<div class="errorMsg"
  style="display:{!IF(accountIdVal.hasError,'block','none')}">
  <strong>Error:</strong>&nbsp;{!accountIdVal.error}
</div>
```

The `error` and `errorMsg` style classes are from the standard Salesforce stylesheets. Using these classes entails the risk that if Salesforce update their styling, these classes may be changed or removed entirely, which would break the error message styling. In order to avoid this, clone the Salesforce styles into your own stylesheet.

See also

- The *Adding error messages to field inputs* recipe in this chapter shows how error messages can be displayed against input elements associated with sObject fields.

Using field sets

A **field set** defines a group of sObject fields. A Visualforce page can iterate the fields contained in the set and access the values and other information, such as label or type, through the merge syntax. This decouples maintenance of the page from the skill set required to author Visualforce pages, and allows administrators to add or remove fields through point and click.

In this recipe we will create two field sets for the contact sObject: one to display the address information and another to display information about the contact. We will then create a Visualforce page that uses these field sets to render input components inside a page block section, which allow a contact record to be created or edited.

[Field sets are in beta as of the Summer 13 release of Salesforce; this means that the functionality is of production quality but contains known limitations.]

Getting ready

This recipe relies on two field sets, which must be created before the Visualforce page can be created.

1. First, create the contact detail field set. Navigate to the Contact Field Sets setup page by clicking on **Your Name** | **Setup** | **Customize** | **Contacts** | **Field Sets**.
2. Click on the **New** button.
3. Enter `Detail` in the **Field Set Label** field.
4. Accept the default **Detail** that is automatically generated for the **Field Set Name** field.

   [Ensure that the name is correctly set, as the Visualforce page uses this to retrieve the field set.]

5. Enter `In the cookbook field sets example page` in the **Where is this used?** field and click on the **Save** button.
6. On the resulting page, drag the following fields onto the **In the Field Set** pane: **Salutation**, **First Name**, **Last Name**, **Contact Description**, **Business Phone**, and **Email**.

   [If you wish to allow administrators to add additional fields to the field set, these must be dragged onto the **Available for the Field Set** pane.]

7. Click on the **Save** button to commit the changes.
8. Next, create the contact address field set. Navigate to the Contact Field Sets setup page by clicking on **Your Name** | **Setup** | **Customize** | **Contacts** | **Field Sets**.
9. Click on the **New** button.
10. Enter `Address Information` in the **Field Set Label** field.
11. Accept the default **Address_Information** that is automatically generated for the **Field Set Name** field.

> Ensure that the name is correctly set, as the Visualforce page uses this to retrieve the field set.

12. Enter `In the cookbook field sets example page` in the **Where is this used?** field and click on the **Save** button.
13. On the resulting page, drag the following fields onto the **In the Field Set** pane: **Mailing Street**, **Other Street**, **Mailing City**, **Other City**, **Mailing State/Province**, **Other State/Province**, **Mailing Zip/Postal Code**, **Other Zip/Postal Code**, **Mailing Country**, and **Other Country**.
14. Click on the **Save** button to commit the changes.

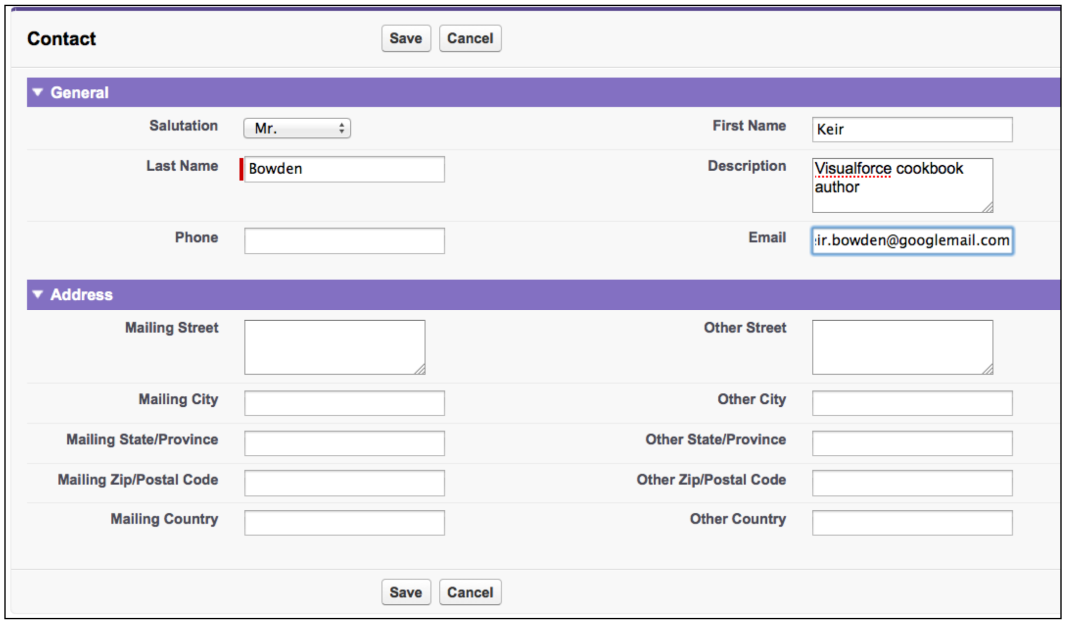
How to do it...

1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `FieldSets` in the **Label** field.
4. Accept the default **FieldSets** that is automatically generated for the **Name** field.
5. Paste the contents of the `FieldSets.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **FieldSets** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **FieldSets** page: `https://<instance>/apex/FieldSets`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Salutation	Mr.	First Name	Keir
Last Name	Bowden	Description	Visualforce cookbook author
Phone		Email	ir.bowden@googlemail.com
Mailing Street		Other Street	
Mailing City		Other City	
Mailing State/Province		Other State/Province	
Mailing Zip/Postal Code		Other Zip/Postal Code	
Mailing Country		Other Country	

The Visualforce page uses the `$ObjectType` global variable to access a named field set and iterate the contained fields, which are bound to input components.

```
<apex:pageBlockSection title="General">
  <apex:repeat value="{!$ObjectType.Contact.FieldSets.Detail}"
        var="field">
    <apex:inputField value="{!Contact[field]}" />
  </apex:repeat>
</apex:pageBlockSection>
```

Adding a custom lookup to a form

The Salesforce standard lookup functionality renders a dialog that supports a small amount of customization. The fields displayed may be configured, and if **Enhanced Lookups** have been enabled, the results can be filtered and ordered, and large result sets may be paged through. The lookup dialog's layout can neither be altered, nor can it be branded or contain additional help text.

In this recipe we will create a Visualforce lookup page that replaces the lookup dialog and provides additional instructions to the user. An additional Visualforce page will demonstrate how this can be be integrated into a custom opportunity create page.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

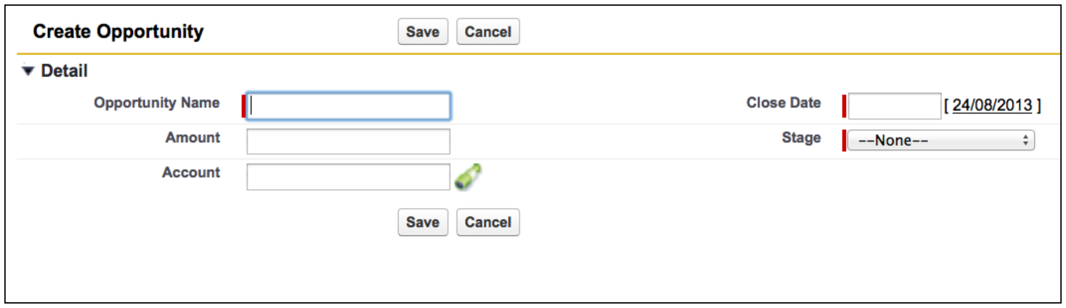
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `LookupController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the custom lookup Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `LookupPopup` in the **Label** field.
8. Accept the default **LookupPopup** that is automatically generated for the **Name** field.
9. Paste the contents of the `LookupPopup.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **LookupPopup** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Finally, create the custom create opportunity Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

14. Click on the **New** button.
15. Enter `Lookup` in the **Label** field.
16. Accept the default **Lookup** that is automatically generated for the **Name** field.
17. Paste the contents of the `Lookup.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
18. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
19. Locate the entry for the **Lookup** page and click on the **Security** link.
20. On the resulting page, select which profiles should have access and click the **Save** button.

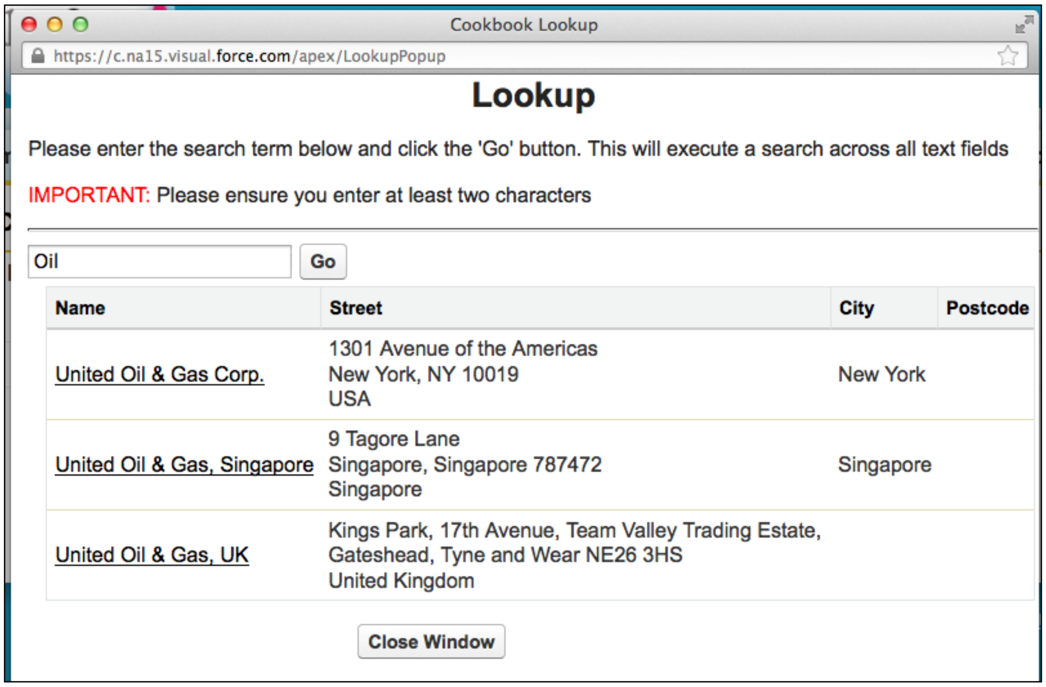
How it works...

Opening the following URL in your browser displays the custom opportunity create **Lookup** page: `https://<instance>/apex/Lookup`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Clicking on the telescope icon to the right of the **Account** field opens the custom lookup dialog.


Cookbook Lookup

https://c.na15.visual.force.com/apex/LookupPopup

Lookup

Please enter the search term below and click the 'Go' button. This will execute a search across all text fields

IMPORTANT: Please ensure you enter at least two characters

Oil Go

Name	Street	City	Postcode
United Oil & Gas Corp.	1301 Avenue of the Americas New York, NY 10019 USA	New York	
United Oil & Gas, Singapore	9 Tagore Lane Singapore, Singapore 787472 Singapore	Singapore	
United Oil & Gas, UK	Kings Park, 17th Avenue, Team Valley Trading Estate, Gateshead, Tyne and Wear NE26 3HS United Kingdom		

Close Window

The **Lookup** page contains a hidden input component that is populated with the ID of the chosen account, while the text input component is used to display the name of the chosen account. JavaScript is used to execute the function to open the lookup dialog when the telescope icon is clicked.

```
<apex:inputHidden value="{!Opportunity.AccountId}"
    id="targetId" />
<apex:inputText size="20" id="targetName" onFocus="this.blur "/>
<a href="#" onclick="openLookupPopup('{!$Component.targetName}',
'{!$Component.targetId}'); return false">
<apex:image style="vertical-align:middle;width:21px; height:21px"
    value="/img/icon/telescope16.png" /></a>
```

Note that the input text component defines an `onFocus` handler that simply removes the focus from itself. This stops the user entering an account name directly, as there is no controller logic to retrieve the ID based on the name.

The ID of the input fields for the chosen account ID and name are passed to the dialog as URL parameters. This allows the dialog to populate the fields once the user has chosen an account.

In addition to opening the dialog, the **Lookup** page also handles closing it. This is because many browsers do not allow a page to close a dialog if it was not responsible for opening it.

```
var newWin=null;
function openLookupPopup(name, id)
{
  var url="/apex/LookupPopup?namefield=" + name +
    "&idfield=" + id;
  newWin=window.open(url, 'Popup',
    'height=500,width=600,left=100,top=100,resizable=no,
    scrollbars=yes,toolbar=no,status=no');
  if (window.focus)
  {
    newWin.focus();
  }

  return false;
}

function closeLookupPopup()
{
if (null!=newWin)
  {
    newWin.close();
  }
}
```

The **Lookup** dialog attaches an `onclick` handler to each of the account names in the search results, to execute the JavaScript function that populates the name and ID of the chosen account.

```
<apex:column headerValue="Name">
  <apex:outputLink value="#"
    onclick="fillIn('{!account.Name}',
      '{!account.id}')">{!account.Name}
  </apex:outputLink>
</apex:column>
```

The `fillIn` function populates the fields in the parent window and then executes the JavaScript in the parent window to close the pop up.

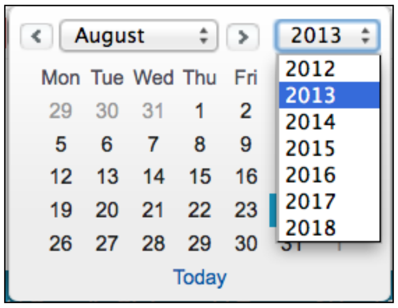
```
function fillIn(name, id)
{
var winMain=window.opener;
if (null==winMain)
{
    winMain=window.parent.opener;
}
var ele=winMain.document.getElementById
    ('{!$CurrentPage.parameters.namefield}');
ele.value=name;
ele=winMain.document.getElementById
    ('{!$CurrentPage.parameters.idfield}');
ele.value=id;
    winMain.closeLookupPopup();
}
```

See also

- The *Opening a pop-up window* recipe in *Chapter 1*, *General Utilities* shows how to create a pop-up window when a user clicks on a link in a Visualforce page.

Adding a custom datepicker to a form

The standard datepicker that is rendered when an `<apex:inputField />` component is bound to an sObject field of type `Date` or `DateTime` has a limited range of years available, as shown in the following screenshot:



While this range of years may be suitable for opportunity close dates, it is unsuitable for capturing a contact's date of birth. One option to improve this is to add some JavaScript to the page that alters the datepicker year range, but this entails a risk as it relies on the standard datepicker code to remain the same.

In this recipe, we will integrate a third-party JavaScript datepicker with a Visualforce input field bound to a date. The datepicker used is from Design2Develop and can be downloaded from `http://www.design2develop.com/calendar/`. This has been chosen as the style class names; do not conflict with any standard Salesforce style classes as of the Summer 13 release of Salesforce.

Getting ready

This recipe requires the Design2Develop calendar ZIP file to be present as a static resource.

1. Download the custom datepicker ZIP file from `http://www.design2develop.com/calendar/#download`.
2. Navigate to the Static Resource setup page by clicking on **Your Name** | **Setup** | **Develop** | **Static Resources**.
3. Click on the **New** button.
4. Enter `D2DCalendar` in the **Name** field.
5. Enter `Design2Develop JavaScript Date Picker` in the **Description** field.
6. Click on the **Browse** button and select the `calendar.zip` file downloaded in step 1.
7. Accept the default **Private** value for the **Cache Control** field and click on the **Save** button.

How to do it...

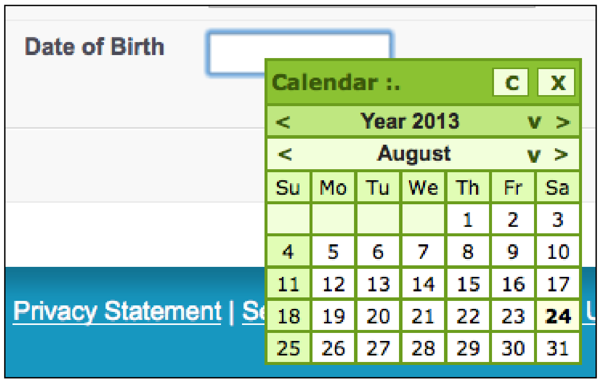
1. First, create the Visualforce page that the datepicker will be used in by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `DatePicker` in the **Label** field.
4. Accept the default **DatePicker** that is automatically generated for the **Name** field.
5. Paste the contents of the `DatePicker.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **DatePicker** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **DatePicker** Visualforce page: `https://<instance>/apex/DatePicker`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Clicking on the **Date of Birth** input field renders the custom datepicker with a wider range of years as shown in the following screenshot:



Note that a side effect of this is that the styling of the datepicker may be customized. The sample code uses the `calendar_green.css` stylesheet supplied as part of the `calendar.zip` file to produce a datepicker that is colored green.

The **DatePicker** page provides an `onFocus` handler for the **Date of Birth** input field.

```
<apex:inputText id="birthdate" size="10"
  value="{!Contact.BirthDate}"
    onfocus="initialiseCalendar(this,
      '{!$Component.birthdate}')"/>
```

The `initialiseCalendar` JavaScript function extracts the existing date from the input component, if present, and passes this to the custom datepicker initialization code, which renders the datepicker with the existing date preselected, if defined.

```
function initialiseCalendar(obj, eleId)
{
  var element=document.getElementById(eleId);
  var params='close=true';
  if (null!=element)
```

```
    {
      if (element.value.length>0)
      {
        // date is formatted dd/mm/yyyy - pull out the month and year
        var month=element.value.substr(3,2);
        var year=element.value.substr(6,4);
        params+=',month='+month;
        params+=',year='+year;
      }
    }

    fnInitCalendar(obj, eleId, params);
  }
```

See also

- The *Adding a custom lookup to a form* recipe in this chapter shows how to replace the standard lookup component with a custom version.

Retrieving fields when a lookup is populated

When viewing an sObject that has a lookup relationship to another sObject, additional fields from the related sObject can be displayed on the page using formula fields. When creating a new record, or editing an existing record and changing the lookup value, formula fields cannot be used, as the lookup field has only been populated with a record ID and the related record has not been retrieved.

In this recipe, we will create a Visualforce page that allows a user to create a case sObject record. The case standard controller and a controller extension manage the new case record. When the lookup to the account that the case is related to is populated, additional fields are retrieved from the account record and displayed.

Getting ready

This recipe makes use of a controller extension, so this will need to be present before the Visualforce page can be created.

How to do it...

1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.

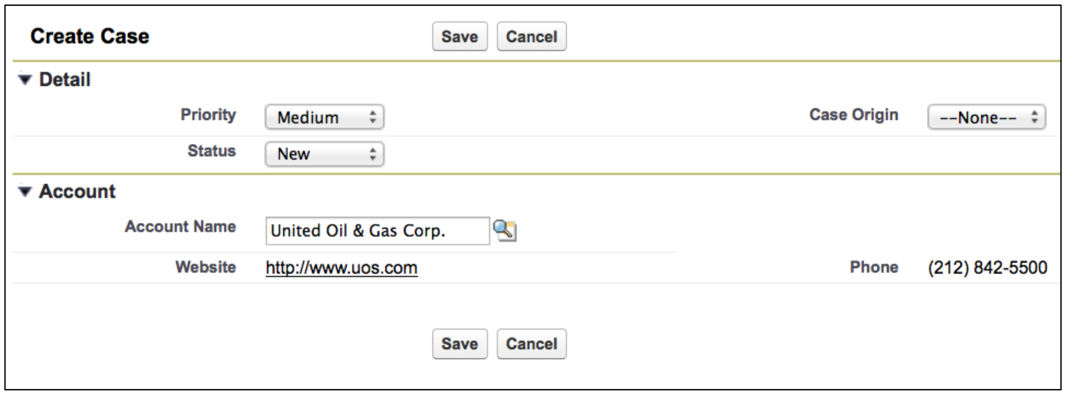
3. Paste the contents of the `PullLookupFieldsExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `PullLookupFields` in the **Label** field.
8. Accept the default **PullLookupFields** that is automatically generated for the **Name** field.
9. Paste the contents of the `PullLookupFields.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page, by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **PullLookupFields** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following page in your browser displays the **PullLookupFields** page: `https://<instance>/apex/PullLookupFields`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Populating the account lookup automatically populates the **Website** and **Phone** output elements with the details from the account.


Create Case

Detail

Priority	Medium	Case Origin	--None--
Status	New		

Account

Account Name	United Oil & Gas Corp.		
Website	http://www.uos.com	Phone	(212) 842-5500

The **PullLookupFields** page executes an `action` method when the contents of the case account lookup field changes.

```
<apex:inputField value="{!Case.AccountId}">
  <apex:actionSupport event="onchange" action="{!accountSelected}"
    rerender="account, msgs" status="stat"/>
</apex:inputField>
```

When the `accountSelected` method is executed, it confirms that the field has not been cleared and retrieves the related account record, attaching this to the case record from the standard controller.

```
Case cs=(Case) stdCtrl.getRecord();

// handle the situation where the account field has been cleared
if (!String.isBlank(cs.AccountId))
{
    cs.Account=[select Website, Phone from Account
        where id=:cs.AccountId];
}
```

When the account section of the page is rerendered, the related fields are retrieved using the standard dot notation.

```
<apex:outputField value="{!Case.Account.Website}"/>
<apex:outputField value="{!Case.Account.Phone}"/>
```

Breaking up forms with action regions

The submission of a form in a Visualforce page causes the view state and all user inputs to be processed by the controller. In the event that the form is being submitted back, purely to introduce some additional information based on a single user input, this can be inefficient, especially if there are a large number of field inputs on the page. The `<apex:actionRegion />` component can be used to break the form up into discrete sections, reducing the amount of data processed by the controller and improving performance of the page.

In this recipe we will create a Visualforce page that allows a user to create a case record. The case subject is automatically generated by a controller extension from a base subject entered by the user and the name of the account that the case is associated with. A change to either the base subject or the account lookup causes the form to be submitted in order to update the generated subject. Each of these fields is contained in an action region, ensuring that only the controller only processes the updated value.

Getting ready

This recipe makes use of a controller extension, so this will need to be present before the Visualforce page can be created.

How to do it...

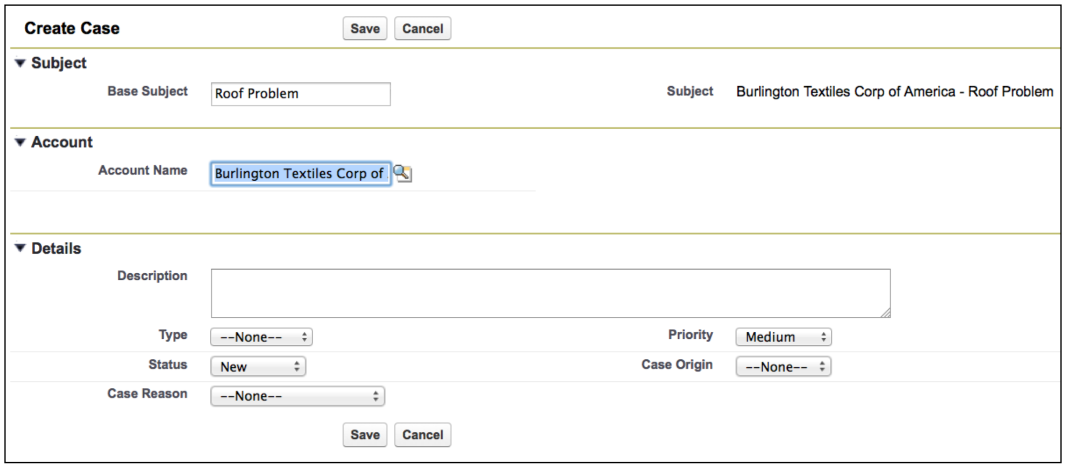
1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ActionRegionExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionRegion` in the **Label** field.
8. Accept the default **ActionRegion** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionRegion.page` page from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ActionRegion** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionRegion** page: `https://<instance>/apex/ActionRegion`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Entering a value in the **Base Subject** or **Account Name** field and losing focus on the field (by tabbing out or clicking into another field) updates the **Subject** field with the latest values from these fields.



Base Subject	Roof Problem	Subject	Burlington Textiles Corp of America - Roof Problem
Account Name	Burlington Textiles Corp of		
Description			
Type	--None--	Priority	Medium
Status	New	Case Origin	--None--
Case Reason	--None--		

The **ActionRegion** page defines an action region for each of the fields that contribute to the **Subject** field value.

```
<apex:actionRegion>
  <apex:pageBlockSection title="Subject" id="subject">
    <apex:pageBlockSectionItem >
      <apex:outputLabel value="Base Subject"/>
      <apex:inputText value="{!baseSubject}">
        <apex:actionSupport event="onchange"
          action="{!setupSubject}" rerender="subject, msgs"
          status="stat"/>
      </apex:inputText>
    </apex:pageBlockSectionItem>
    <apex:outputField value="{!Case.Subject}" />
  </apex:pageBlockSection>
</apex:actionRegion>
```

The `setupSubject` action method defined in the page controller extension concatenates the name of the selected account with the base subject and assigns this to the **Subject** field.

```
String subject='';
Case cs=(Case) stdCtrl.getRecord();

// handle the situation where the account field has been cleared
if (!String.isBlank(cs.AccountId))
{
    Account acc=[select Name from Account where id=:cs.AccountId];
    subject+=acc.Name + ' - ';
  }

  if (null!=baseSubject)
  {
    subject+=baseSubject;
  }

  cs.Subject=subject;
```

See also

- The *Avoiding validation errors with action regions* recipe in this chapter shows how action regions may be used to submit part of a form for server-side processing that would otherwise be blocked, due to validation errors.

The "Please wait" spinner

When a user carries out an action that results in a Visualforce form submission, for example, clicking a button, it can be useful to render a visual indication that the submit is in progress. Without this a user may click on the button again, or assume there is a problem and navigate away from the page. The standard Visualforce `<apex:actionStatus />` component can display messages when starting and stopping a request, but these messages are easily missed, especially if the user is looking at a different part of the page.

In this recipe, we will create a Visualforce page that allows a user to create a case sObject record utilizing the case standard controller. When the user clicks on the button to create the new record, a spinner GIF will be displayed. In order to ensure that we have the user's full attention, the page will be grayed out while the submit takes place.

How to do it…

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `Working` in the **Label** field.
4. Accept the default **Working** that is automatically generated for the **Name** field.
5. Paste the contents of the `Working.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

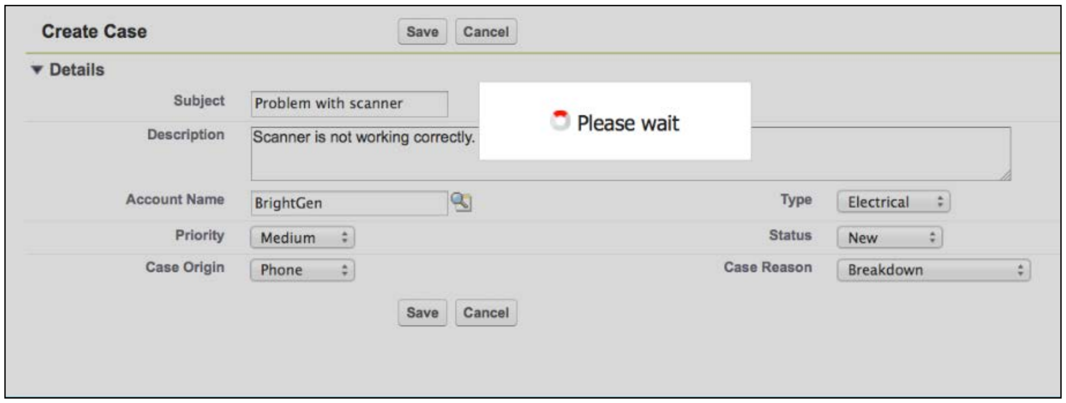
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **Working** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Working** page: `https://<instance>/apex/Working`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example `na6.salesforce.com`.

Populating the fields on the page and clicking on the Save button greys out the page contents and displays a message with a spinning icon.



The **Working** page defines two `div` elements that are initially hidden as follows:

```
<div id="opaque"/>
<div id="spinner">
  <p align="center"
    style='{font-family:"Arial", Helvetica, sans-serif; font-
size:20px;}'>
    <apex:image value="/img/loading.gif"/>&nbsp;Please wait
</p>
</div>
```

The `opaque` div greys out the entire page when made visible—the style for this, and the `spinner` div are defined at the top of the Visualforce page.

The **Save** and **Cancel** buttons have `onclick` handlers that execute a JavaScript function that makes the `opaque` and `spinner` div elements visible.

```
<script>
function showSpinner()
{
    document.getElementById('opaque').style.display='block';
    var popUp = document.getElementById('spinner');

    popUp.style.display = 'block';
}
</script>

<apex:commandButton value="Save" action="{!save}"
      onclick="showSpinner()" />
<apex:commandButton value="Cancel" action="{!cancel}"
      onclick="showSpinner()" />
```

Avoiding validation errors with action regions

Submitting a Visualforce form without populating a required field causes an error message to be returned to the user. When the user has triggered the submission by clicking on a button, a message of this nature will not come as a surprise. If the submission is automatically triggered, for example, to retrieve fields once a lookup is populated, the sudden and unexpected appearance of an error message is a poor user experience.

In this recipe we will create a Visualforce page to create an opportunity with a number of required fields. When the user selects the account to associate the opportunity with, the form will be submitted, and related fields from the account record populated regardless of whether the required fields have been populated.

Getting ready

This recipe makes use of a controller extension, so this will need to be created before the Visualforce page.

How to do it...

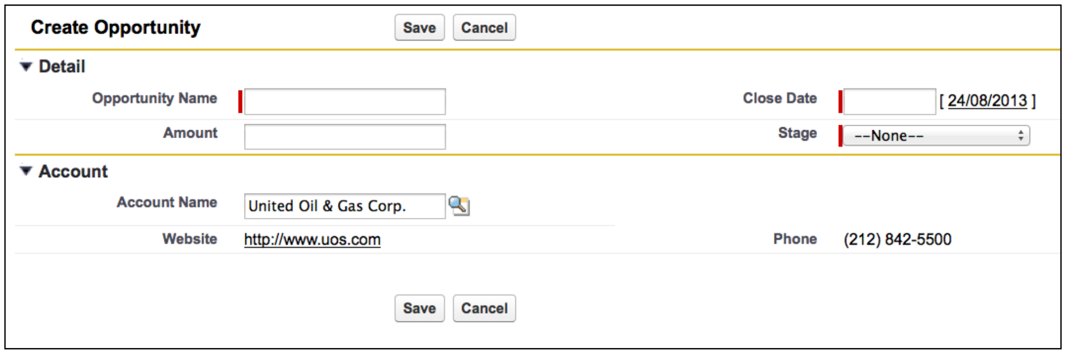
1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ActionRegionAvoidValidationExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionRegionAvoidValidation` in the **Label** field.
8. Accept the default **ActionRegionAvoidValidation** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionRegionAvoidValidation.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ActionRegionAvoidValidation** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionRegionAvoidValidation** page: `https://<instance>/apex/ActionRegionAvoidValidation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Entering a value in the **Account Name** field and losing focus on the field (by tabbing out or clicking into another field) updates the **Website** and **Phone** fields with the details from the chosen account, even though the required fields of **Opportunity Name**, **Close Date**, and **Stage** are not populated.



Opportunity Name		Close Date	[24/08/2013]
Amount		Stage	--None--
Account Name	United Oil & Gas Corp.		
Website	http://www.uos.com	Phone	(212) 842-5500

The **ActionRegionAvoidValidation** page defines an action region for the **Account** section, which stops the controllers from processing other fields on the page when the automatic submission takes place. The automatic submission is handled by the `<apex:actionSupport />` component nested in the **Account Name** input field.

```
<apex:actionRegion >
  <apex:pageBlockSection title="Account" id="account">
    <apex:inputField value="{!Opportunity.AccountId}">
      <apex:actionSupport event="onchange"
          action="{!accountSelected}" rerender="account, msgs"
          status="stat"/>
      </apex:inputField>
    <apex:pageBlockSectionItem />
    <apex:outputField value="{!Opportunity.Account.Website}"/>
    <apex:outputField value="{!Opportunity.Account.Phone}"/>
    <apex:actionStatus startText="Getting detail" id="stat" />
  </apex:pageBlockSection>
</apex:actionRegion>
```

The accountSelected method in the controller extension retrieves the related account record and associates it with the opportunity record that the standard controller is managing.

```
Opportunity opp=(Opportunity) stdCtrl.getRecord();

// handle the situation where the account field has been cleared
if (!String.isBlank(opp.AccountId))
{
  opp.Account=[select Website, Phone from Account
    where id=:opp.AccountId];
}
else
{
  opp.Account=null;
}
```

See also

- The *Breaking up forms with action regions* recipe in this chapter shows how areas of a large and complex form may be submitted independent of each other using action regions.

Action chaining

Action chaining allows multiple controller action methods to be executed in a series from a Visualforce page, each in a separate transaction. This technique is rarely used, but does solve the following problems:

- Working around governor limits; for example, repeatedly polling an external system to determine if processing triggered through a web service call has completed without breaching the limit for callouts per transaction. In this case, the same action would be chained to poll the external system and then update the Visualforce page to indicate the user whether the action had completed.
- Avoiding the MIXED_DML_OPERATION error when the controller must modify setup and nonsetup records; for example, changing a user and an opportunity record. In this case, the first action in the chain would modify the user record, while the second would modify the opportunity record.

In this recipe, we will create a Visualforce page to create an opportunity and move it through a number of stages, with each stage transition taking place in a separate transaction. This may be used to avoid governor limits in a situation where an opportunity must progress through each stage individually, but the act of changing stage causes a significant amount of processing to take place, or external systems must be updated as the opportunity progresses.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

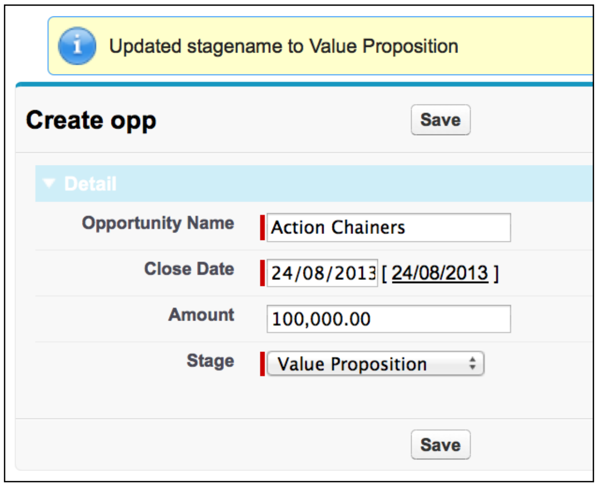
1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ActionChainController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionChain` in the **Label** field.
8. Accept the default **ActionChain** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionChain.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ActionChain** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionChain** page: `https://<instance>/apex/ActionChain`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Filling in the opportunity record fields, setting the **Stage Name** field to **Prospecting**, and clicking on the **Save** button causes the chain of actions to execute, with each action displaying a message to indicate that the opportunity has progressed to a new stage.


Updated stagename to Value Proposition

Create opp

Opportunity Name	Action Chainers
Close Date	24/08/2013 [24/08/2013]
Amount	100,000.00
Stage	Value Proposition

The **ActionChain** page conditionally renders one of a number of action function components and JavaScript that executes the function based on the opportunity stage name.

```
<apex:outputPanel rendered="{!opp.StageName='Prospecting'}">
  <apex:actionFunction name="qualificationJS"
    action="{!qualification}" rerender="js,msgs,detail"
    status="stat"/>
  <script>
    qualificationJS();
  </script>
</apex:outputPanel>

<apex:outputPanel rendered="{!opp.StageName='Qualification'}">
  <apex:actionFunction name="needsAnalysisJS"
    action="{!needsAnalysis}" rerender="js,msgs,detail"
    status="stat"/>
  <script>
    needsAnalysisJS();
  </script>
</apex:outputPanel>
```

Each action function executes a controller action method to progress the opportunity to the next stage, and rerenders the JavaScript section and any messages from the controller.

Errors – harmful if swallowed

When a form submission results in a rerender of a section rather than refreshing the entire Visualforce page, it is very easy to cause error messages from the controller to be swallowed rather than displayed to the user. In this situation, as far as the user is concerned, the form submission is broken; they click on a button and nothing on the page changes.

In this recipe we will create a Visualforce page to create an opportunity. When the account the opportunity is associated with is selected, the form will be submitted and the account record retrieved so that additional fields on the page may populated. If the opportunity name is not defined, the form submission will fail and an error message will be displayed to the user.

Getting ready

This recipe makes use of a controller extension, so this will need to be created before the Visualforce page.

How to do it...

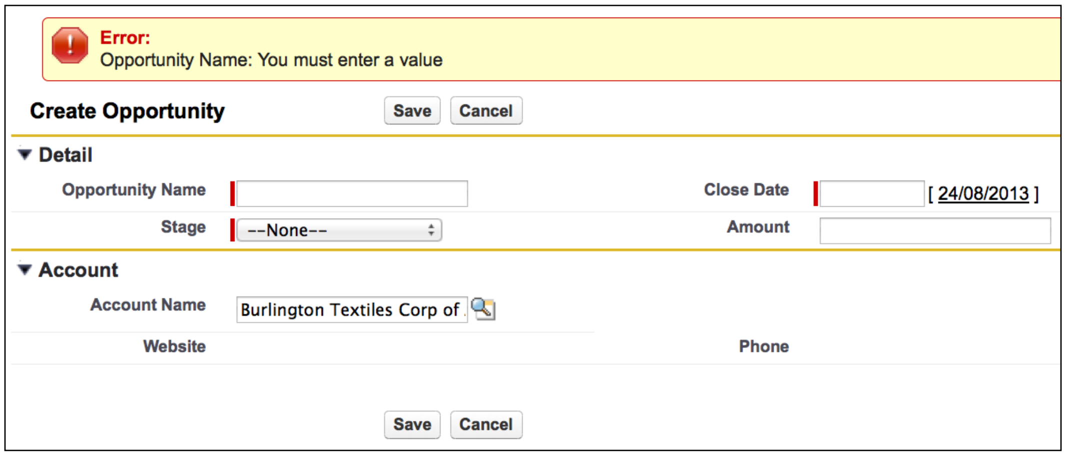
1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `RerenderValidationExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `RerenderValidation` in the **Label** field.
8. Accept the default **RerenderValidation** that is automatically generated for the **Name** field.
9. Paste the contents of the `RerenderValidation.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **RerenderValidation** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **RerenderValidation** page: `https://<instance>/apex/RerenderValidation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Choosing the account before supplying the opportunity name causes the postback to fail with an error message.



Error:
Opportunity Name: You must enter a value

Create Opportunity

Opportunity Name		Close Date	[24/08/2013]
Stage	--None--	Amount	
Account Name	Burlington Textiles Corp of		
Website		Phone	

The Account Name input field has a nested `<apex:actionSupport />` component to submit the form and execute a controller method when the value changes.

```
<apex:inputField value="{!Opportunity.AccountId}">
  <apex:actionSupport event="onchange" action="{!accountSelected}"
    rerender="account, msgs" status="stat"/>
</apex:inputField>
```

This component defines the `rerender` attribute, which will result in only the listed standard components being redrawn when the controller method has completed. This list includes an `<apex:pageMessages />` component.

```
<apex:pageMessages id="msgs" />
```

In the event that an error occurs while submitting or processing the form, the account section will not change. Rerendering the `msgs` component ensures that any errors that occur are communicated back to the user.

See also

- The *Adding error messages to nonfield inputs* recipe in this chapter shows how error messages can be displayed against input elements associated with controller properties.
- The *Adding error messages to field inputs* recipe in this chapter shows how error messages can be displayed against input elements associated with sObject fields.

4
Managing Records

In this chapter, we will cover the following recipes:

- Styling fields as required
- Styling table columns as required
- Attaching an image to a record
- Managing attachments
- Maintaining custom settings
- Refreshing record details from embedded Visualforce
- Using wrapper classes
- Changing options based on the user input
- Changing page layout based on the user input
- Form-based searching

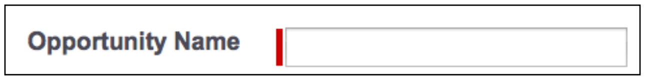
Introduction

One of the common use cases for Visualforce pages is to simplify, streamline, or enhance the management of sObject records. In the earlier chapters we have covered how Visualforce can be used to provide a custom or user interface to create and edit records.

In this chapter, we will use Visualforce to carry out some more advanced customization of the user interface—redrawing the form to change available picklist options, or capturing different information based on the user's selections. We will also see how Visualforce can be used to manage non-sObject information by providing custom user interfaces to allow custom settings and attachments to be maintained, and searching for records based on the value of specific fields.

Styling fields as required

Standard Visualforce input components, such as `<apex:inputText />`, can take an optional required attribute. If set to true, the component will be decorated with a red bar to indicate that it is required, and form submission will fail if a value has not been supplied as shown in the following screenshot:



In the scenario where one or more inputs are required and there are additional validation rules, for example, when one of either the **Email** or **Phone** fields is defined for a contact, this can lead to a drip feed of error messages to the user. This is because the inputs make repeated unsuccessful attempts to submit the form, each time getting slightly further in the process.

In this recipe we will create a Visualforce page that allows a user to create a contact record. The **Last Name** field is captured through a nonrequired input decorated with a red bar identical to that created for required inputs. When the user submits the form, the controller validates that the **Last Name** field is populated and that one of the **Email** or **Phone** fields is populated. If any of the validations fail, details of all errors are returned to the user.

Getting ready

This recipe makes use of a controller extension so this must be created before the Visualforce page.

How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `RequiredStylingExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `RequiredStyling` in the **Label** field.
8. Accept the default **RequiredStyling** that is automatically generated for the **Name** field.

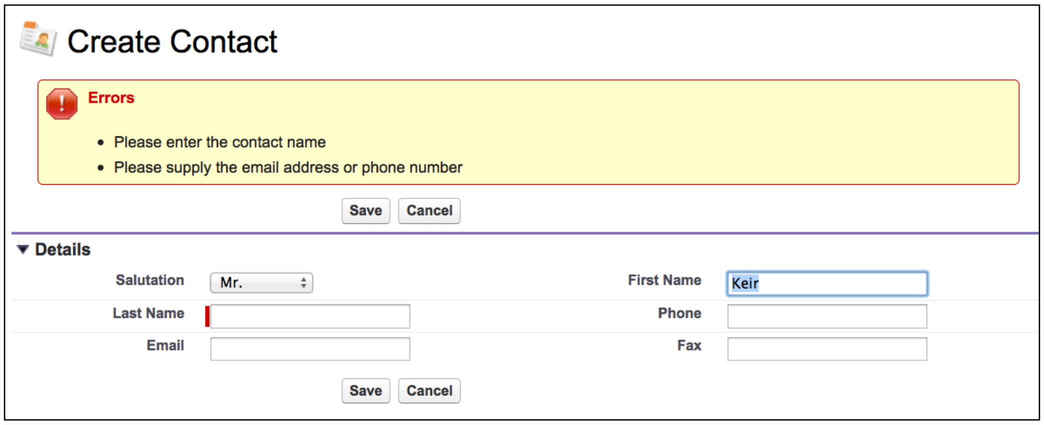
9. Paste the contents of the `RequiredStyling.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **RequiredStyling** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **RequiredStyling** page to create a new contact record: `https://<instance>/apex/RequiredStyling`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

Clicking on the **Save** button without populating any of the fields results in the save failing with a number of errors.


Create Contact

Errors

- Please enter the contact name
- Please supply the email address or phone number

Salutation	Mr.	First Name	Keir
Last Name		Phone	
Email		Fax	

The **Last Name** field is constructed from a label and text input component rather than a standard input field, as an input field would enforce the required nature of the field and stop the submission of the form.

```
<apex:pageBlockSectionItem >
  <apex:outputLabel value="Last Name"/>
  <apex:outputPanel id="detailrequiredpanel" layout="block"
      styleClass="requiredInput">
    <apex:outputPanel layout="block" styleClass="requiredBlock" />
    <apex:inputText value="{!Contact.LastName}"/>
  </apex:outputPanel>
</apex:pageBlockSectionItem>
```

[The required styles are defined in the Visualforce page rather than relying on any existing Salesforce style classes to ensure that if Salesforce changes the names of its style classes, this does not break the page.]

The controller extension `save` action method carries out validation of all fields and attaches error messages to the page for all validation failures.

```
if (String.IsBlank(cont.name))
{
  ApexPages.addMessage(new ApexPages.Message(
    ApexPages.Severity.ERROR,
    'Please enter the contact name'));
  error=true;
}

if ( (String.IsBlank(cont.Email)) &&
              (String.IsBlank(cont.Phone)) )
{
  ApexPages.addMessage(new ApexPages.Message(
    ApexPages.Severity.ERROR,
    'Please supply the email address or phone number'));
  error=true;
}
```

See also

- The *Styling table columns as required* recipe in this chapter shows how to style a column header to indicate that the contents of the column are required.

Styling table columns as required

When maintaining records that have required fields through a table, using regular input fields can end up with an unsightly collection of red bars striped across the table.

In this recipe we will create a Visualforce page to allow a user to create a number of contact records via a table. The contact **Last Name** column header will be marked as required, rather than the individual inputs.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

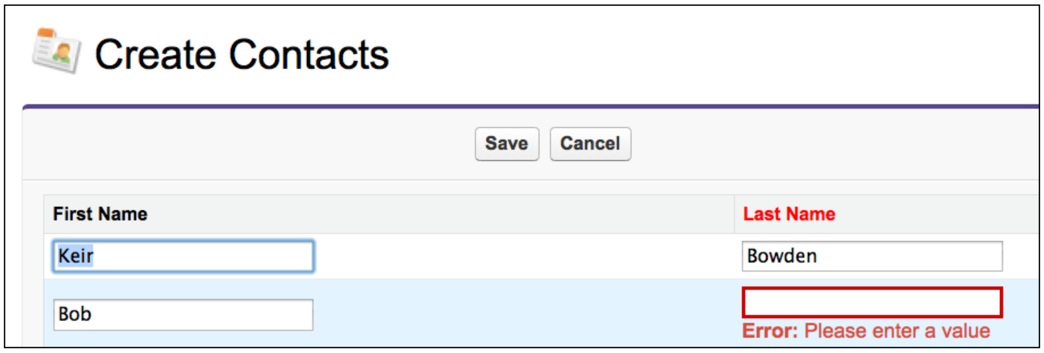
1. First, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `RequiredColumnController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `RequiredColumn` in the **Label** field.
8. Accept the default **RequiredColumn** that is automatically generated for the **Name** field.
9. Paste the contents of the `RequiredColumn.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **RequiredColumn** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **RequiredColumn** page: `https://<instance>/apex/RequiredColumn`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

The **Last Name** column header is styled red, indicating that this is a required field. Attempting to create a record where only **First Name** is specified results in an error message being displayed against the Last Name input for the particular row.



First Name	Last Name
Keir	Bowden
Bob	Error: Please enter a value

The Visualforce page sets the `required` attribute on the `inputField` components in the **Last Name** column to false, which removes the red bar from the component.

```
<apex:column >
  <apex:facet name="header">
    <apex:outputText styleclass="requiredHeader"
        value="{!$ObjectType.Contact.fields.LastName.label}" />
  </apex:facet>
  <apex:inputField value="{!contact.LastName}" required="false"/>
</apex:column>
```

The Visualforce page custom controller `Save` method checks if any of the fields in the row are populated and if this is the case, it checks that the last name is present. If the last name is missing from any record, an error is added. If an error is added to any record, the save does not complete.

```
if ( (!String.IsBlank(cont.FirstName)) ||
   (!String.IsBlank(cont.LastName)) )
{
  // a field is defined - check for last name
  if (String.IsBlank(cont.LastName))
  {
    error=true;
    cont.LastName.addError('Please enter a value');
}
```

`String.IsBlank()` is used as this carries out three checks at once: to check that the supplied string is not null, it is not empty, and it does not only contain whitespace.

See also

- The *Styling fields as required* recipes in this chapter shows how to style an input field to indicate it is required without using the `required` attribute.

Attaching an image to a record

Associating an image with a record is a common requirement while implementing Salesforce, for example, adding a photo to a contact or a custom news story sObject. Using the standard attachments functionality creates a disconnect between the record and the image, requiring additional clicks to view the image, and often relies on the user following a naming convention when uploading the file.

In this recipe, we will create a Visualforce page to allow a user to attach an image to a contact record. The page also displays the image if one has been uploaded. This page will be embedded into the standard contact page layout.

> While the size limit for a record attachment in Salesforce is 5 MB, as the attachment in this recipe is rendered on a Visualforce page, it is important to keep the size of the file as small as possible to avoid lengthy download times and excessive bandwidth usage.

Getting ready

This recipe makes use of a controller extension, so this will need to be created before the Visualforce page.

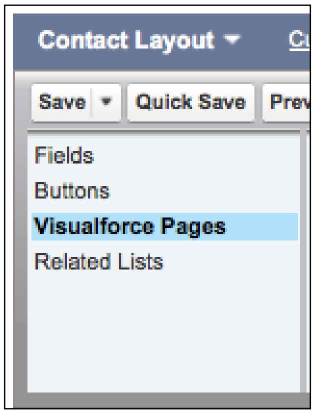
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `AddImageExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `AddImage` in the **Label** field.
8. Accept the default **AddImage** that is automatically generated for the **Name** field.

9. Paste the contents of the `AddImage.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **AddImage** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Finally, add the page to the standard contact page layout. Navigate to the contact Page Layouts page by clicking on **Your Name** | **Setup** | **Customize** | **Contacts** | **Page Layouts**.
14. Locate the page layout you wish to add the Visualforce page to and click on the **Edit** link in the **Action** column.

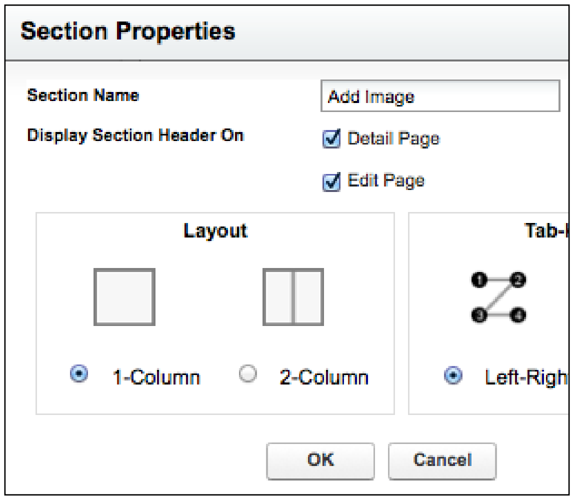
If there are multiple page layouts defined, choose the page layout assigned to your profile. You can view the assignments by clicking on the **Page Layout Assignment** button.

15. On the resulting page layout editor page, click on the **Visualforce Pages** link in the left-hand column of the palette as shown in the following screenshot:



16. Drag the **+Section** option from the right-hand side of the palette and drop this beneath the **Address Information** section.

17. In the **Section Properties** popup, set **Section Name** to **Add Image**, select the **1-Column** radio button in the **Layout** section, and click on the **OK** button.


Section Properties

Section Name: Add Image

Display Section Header On: [✓ Detail Page] [✓ Edit Page]

Layout: [✓ 1-Column] 2-Column

Tab-

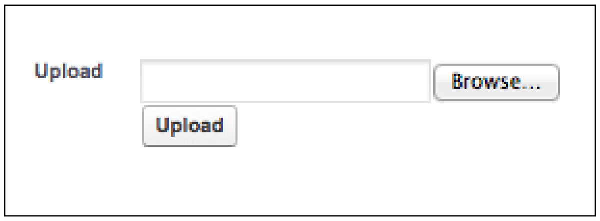
[✓ Left-Righ]

18. Drag the **AddImage** page from the right-hand side of the palette and drop this beneath the **Add Image** section.
19. Click on the **Save** button to commit the page layout changes.
20. Repeat steps 14 to 19 to add the page to any additional page layouts as required.

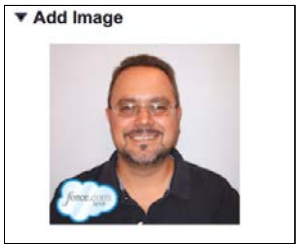
[Only pages that make use of a standard controller may be embedded in a standard record view page.]

How it works...

Navigating to the detail view of any contact record displays the new **Add Image** section. Only the **Upload** section is populated, as no image has been added to the record.



Once an image has been uploaded, this is displayed in the **Add Image** section.



The controller extension extracts the ID of the contact record from the standard controller and stores it in `parentId`.

```
parentId=std.getId();
```

When the file is uploaded, it is stored as an attachment on the record with the name of `image`.

```
public void uploadImage()
{
  att.parentId = parentId;
  att.Name='image';
  insert att;

  att=new Attachment();
}
```

The attached image is displayed using an `<apex:image />` standard component.

```
<apex:pageBlockSectionItem >
  <apex:image
        value="/servlet/servlet.FileDownload?file={!ImageId}"
        rendered="{!NOT(ISBLANK(ImageId))}" />
</apex:pageBlockSectionItem>
```

The controller retrieves the ID of the attached image by querying the record's attachments for one named `image`.

```
public Id getImageId()
{
  Id result=null;
  List<Attachment> images=[select id from Attachment
        where Name='image' and parentId=:parentId
        order by CreatedDate DESC];
  if (images.size()>0)
{
    result=images[0].id;
    }

    return result;
  }
```

Note that the resulting attachments are ordered by the `CreatedDate` field and the ID of the first result is used. This allows the user to upload a new image without having to remember to delete any existing images.

There's more...

As the controller extension deals with the ID of the record from the standard controller, rather than a specific sObject type, this can be used to extend any standard controller to provide this functionality.

See also

- The *Managing attachments* recipe in this chapter shows how to manage a record and its attachments from a single page.

Managing attachments

The standard mechanism of attaching files to Salesforce records navigates the user away from the record to a dedicated upload page. This leaves the user unable to see if they are duplicating an existing attachment, or see the exact details of any fields that may be required to name the attachment correctly.

In this recipe, we will create a Visualforce page to allow a user to attach files directly to a contact record, displaying fields from the record and details of any existing attachments.

Getting ready

This recipe makes use of a custom controller, so this must be present before the Visualforce page can be created.

How to do it...

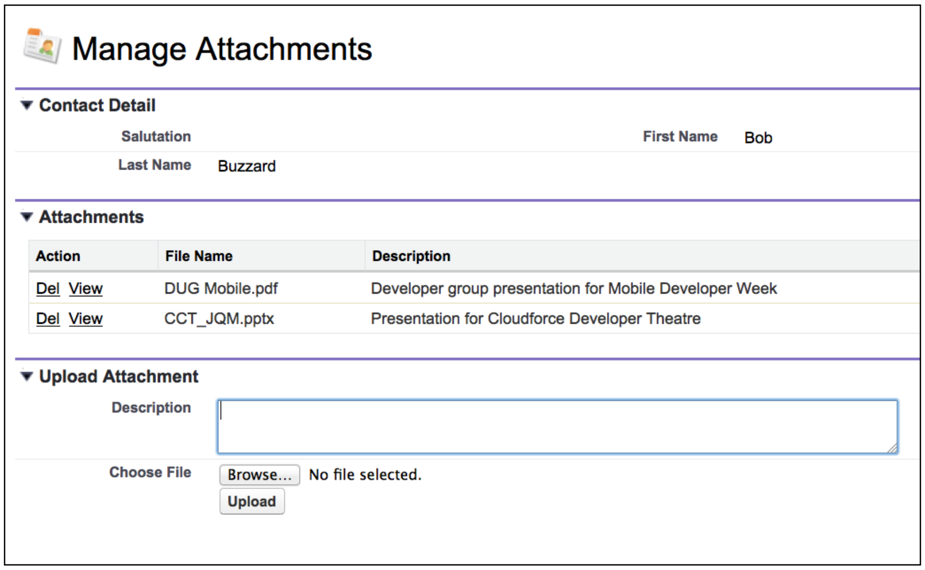
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.

3. Paste the contents of the `AttachmentsExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `Attachments` in the **Label** field.
8. Accept the default **Attachments** that is automatically generated for the **Name** field.
9. Paste the contents of the `Attachments.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **Attachments** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Attachments** page: `https://<instance>/apex/Attachments?id=<contact_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the ID of any contact record in your organization.


Manage Attachments

Contact Detail

Salutation		First Name	Bob
Last Name	Buzzard		

Attachments

Action	File Name	Description
Del View	DUG Mobile.pdf	Developer group presentation for Mobile Developer Week
Del View	CCT_JQM.pptx	Presentation for Cloudforce Developer Theatre

Upload Attachment

Description

Choose File

The Visualforce page retrieves the attachments associated with the contact record through the standard related list.

```
<apex:pageBlockTable value="{!Contact.attachments}"
            var="attachment" >
  <apex:column headerValue="Action">
    <apex:commandLink action="{!deleteAttachment}" value="Del">
      <apex:param name="deleteId"
        assignTo="{!selectedAttachmentId}"
        value="{!attachment.id}"/>
    </apex:commandLink>
    &nbsp;
    <apex:outputLink
      value="/servlet/servlet.FileDownload?file={!attachment.id}"
      target="_blank">View</apex:outputLink>
  </apex:column>
  <apex:column value="{!attachment.Name}" />
  <apex:column value="{!attachment.Description}" />
</apex:pageBlockTable>
```

Note that the **Del** link in the **Action** column passes the ID of the attachment to delete using the `<apex:param />` tag.

The controller extension sets `parentId` of the uploaded attachment to the ID of the contact record being managed by the standard controller, prior to inserting into the database.

```
att.ParentId = recordId;
insert att;

att=new Attachment();

PageReference result=ApexPages.CurrentPage();
result.setRedirect(true);
return result;
```

Note that the `Redirect` attribute of the returned page reference is set to `true` to force a client-side redirect back to the same page. This causes the standard controller to be constructed from scratch, querying the latest attachments from the database.

There's more...

As the controller extension deals with the ID of the record from the standard controller rather than a specific sObject type, this can be used to extend any standard controller to provide this functionality.

See also

- The *Attaching an image to a record* recipe in this chapter shows how to upload an image as an attachment and display it in a record.

Maintaining custom settings

Custom settings are a natural fit for data that controls application behavior. They are similar to custom sObjects but are cached, and so do not have to be retrieved from the Salesforce database each time they are accessed. For more information, refer to the *Custom Settings Overview* page in the Salesforce online help. Unlike custom sObjects, custom settings do not have a configurable user interface provided by the platform, which can make maintenance a challenge for inexperienced administrators.

In this recipe, we will create a Visualforce frontend to an existing custom setting that allows an administrator to take an application in and out of maintenance, with an associated message to display to users.

Getting ready

This recipe makes use of a custom setting, so this will need to be created and populated before the Visualforce page can be created.

1. Navigate to the Custom Settings setup page by clicking on **Your Name** | **Setup** | **Develop** | **Custom Settings**.
2. Click on the **New** button.
3. Enter `VF Cookbook Settings` in the **Label** field.
4. Enter `VF_Cookbook_Settings` in the **Object Name** field.
5. Select the **List** option from the in the **Setting Type** picklist.
6. Select the **Protected** option from the in the **Visibility** picklist.
7. Enter `Maintaining Custom Settings Recipe` into the **Description** field and click on **Save**.
8. On the resulting page, click on the **New** button in the **Custom Fields** section.
9. Select the **Checkbox** radio button on the **Step 1. Choose the field type** page and click on the **Next** button.

10. Enter `In Maintenance` in the **Field Label** field on the **Step 2. Enter the Details** page.
11. Accept the default **In_Maintenance** that is automatically generated for the **Field Name** field.
12. Select **Unchecked** for the **Default Value** radio button and click on the **Next** button.
13. Click on the **Save & New button** on the **Step 3. Confirm information** page.
14. Select the **Text Area** radio button on the **Step 1. Choose the field type** page and click on the **Next** button.
15. Enter `Message` in the **Field Label** field on the **Step 2. Enter the Details** page.
16. Accept the default **Message** that is automatically generated for the **Field Name** field.
17. Click on the **Save** button on the **Step 3. Confirm information** page.
18. Next, create the instance of the custom setting that will be managed by the Visualforce page. Navigate to the Custom Settings setup page by clicking on **Your Name** | **Setup** | **Develop** | **Custom Settings**.
19. Locate the entry the **VF Cookbook Settings** and click on the **Manage** link.
20. Click on the **New** button.
21. Enter `VF Cookbook App` in the **Name** field and leave the other fields empty.

Ensure the **Name** field is set correctly as the Visualforce page custom controller relies on this to retrieve the custom setting.

How to do it...

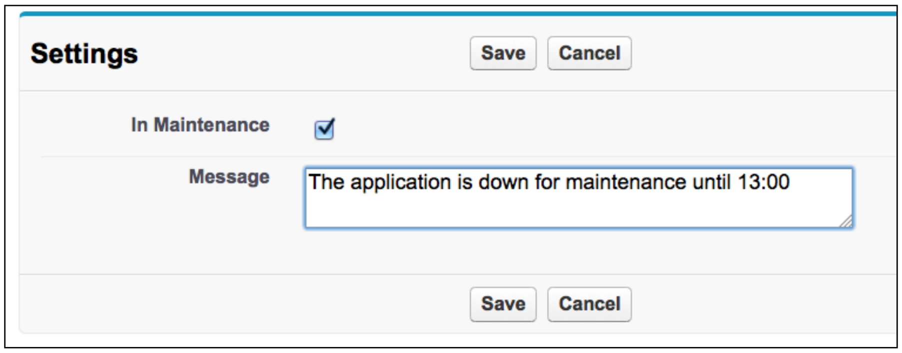
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SettingsController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `Settings` in the **Label** field.
8. Accept the default **Settings** that is automatically generated for the **Name** field.
9. Paste the contents of the `Settings.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **Settings** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the custom setting maintenance **Settings** page: `https://<instance>/apex/Settings`.

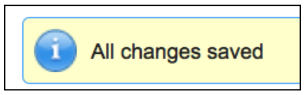
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



In Maintenance: ✓

Message: The application is down for maintenance until 13:00

Populating the fields and clicking on the **Save** button displays a message that the changes have been saved.



All changes saved

While clicking on the **Cancel** button takes the user back to their home page.

The custom controller defines the name of the custom setting as a `public final static` property (effectively a constant) to avoid hardcoding values in methods,

```
public final static String VF_COOKBOOK_APP='VF Cookbook App';
```

The custom controller retrieves the custom setting without consuming a SOQL query.

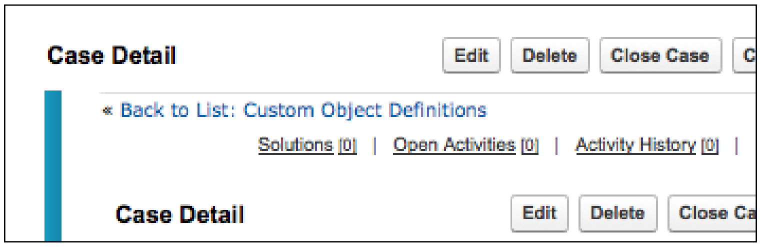
```
public VF_Cookbook_Settings__c settings {get; set;}

public SettingsController()
{
  settings=VF_Cookbook_Settings__c.
                        getInstance(VF_COOKBOOK_APP);
}
```

The action method executed by clicking on the **Save** button stores the custom setting through a DML `update` call.

```
public PageReference Save()
{
  update settings;
  ApexPages.addMessage(new ApexPages.Message(
    ApexPages.Severity.INFO, 'All changes saved'));

  return null;
}
```

Refreshing record details from embedded Visualforce

A Visualforce page can be embedded into a standard or custom sObject record view page, providing the standard controller for the sObject type manages it. This technique is often used to allow information to be added to a record or its related lists without leaving the view page. The Visualforce page is embedded in the record view page using an iframe. This means that returning a page reference to send the user to the record view page after an update results in the entire record view page being displayed inside the iframe, as shown in the following screenshot:


Case Detail
Edit
Delete
Close Case
« Back to List: Custom Object Definitions
Solutions [0] | Open Activities [0] | Activity History [0] |
Case Detail
Edit
Delete

In this recipe, we will create a Visualforce page that is embedded inside the standard case sObject record view page and allows a case comment to be added directly from the page. Upon saving a case comment, the entire record view page will be refreshed to display the updated case comments related list.

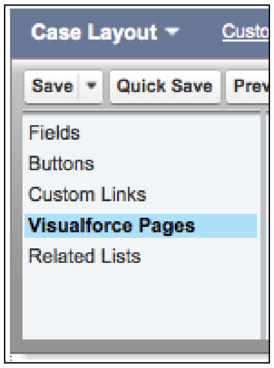
Getting ready

This recipe makes use of a controller extension, so this must be present before the Visualforce page can be created.

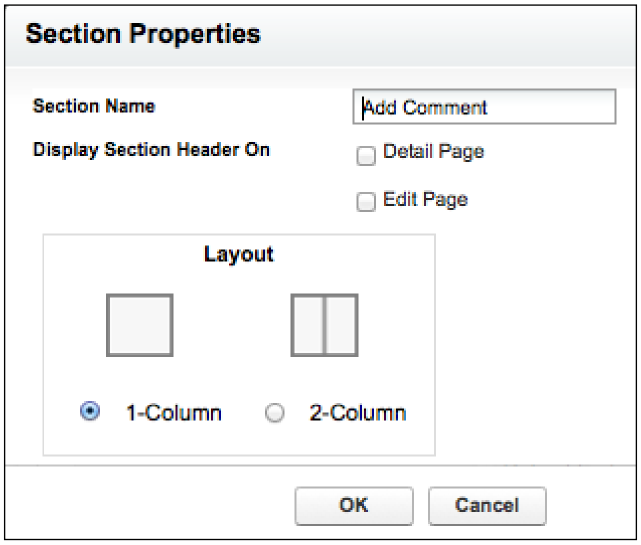
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `RefreshEmbeddedExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `RefreshEmbedded` in the **Label** field.
8. Accept the default **RefreshEmbedded** that is automatically generated for the **Name** field.
9. Paste the contents of the `RefreshEmbedded.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **RefreshEmbedded** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Finally, add the page to the standard case page layout. Navigate to the case Page Layouts page by clicking on **Your Name** | **Setup** | **Customize** | **Case** | **Page Layouts**.
14. Locate the first page layout to add the page to and click on the **Edit** link in the **Action** column.

15. On the resulting page layout editor page, click on the **Visualforce Pages** link in the left-hand column of the palette.



16. Drag the **+Section** option from the right-hand side of the palette and drop this beneath the standard and custom buttons.
17. In the **Section Properties** popup, set the **Section Name** to **RefreshEmbedded**, select the **1-column** radio button in the **Layout** section, and click on the **OK** button.


Section Properties

Section Name: Add Comment

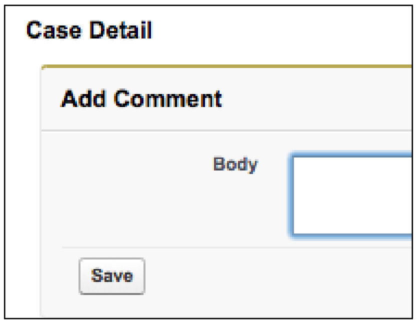
Display Section Header On: [ ] Detail Page [ ] Edit Page

Layout: [✓ 1-Column] 2-Column

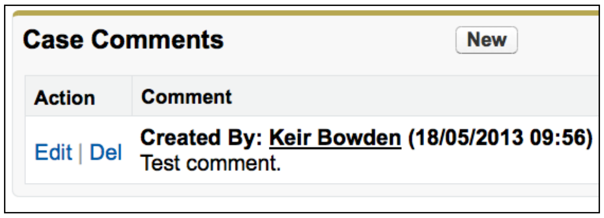
18. Drag the **RefreshEmbedded** page from the right-hand side of the palette and drop this beneath the **Add RefreshEmbedded** section.
19. Click on the **Save** button to commit the page layout changes.
20. Repeat steps 14 to 19 to add the page to additional page layouts.

How it works...

Navigating to the detail view of any case record displays the new **Add Comment** section,



Entering a comment into the **Body input** field and clicking on the **Save** button refreshes the entire page and displays the new comment in the **Case Comments** related list.



Action	Comment
Edit | Del	Created By: Keir Bowden (18/05/2013 09:56) Test comment.

The action method invoked by the **Save** button sets a `Boolean` property to indicate that the page must be refreshed, and defines the target URL reference.

```
public PageReference save()
{
  cc.parentId=stdCtrl.getId();
  insert cc;

  refreshPage=true;
  pageRef=stdCtrl.view().getUrl();

  return null;
}
```

When the Visualforce page is refreshed, JavaScript is conditionally rendered based on the value of the `refreshPage` property to redirect the main page to the target URL.

```
<apex:outputPanel rendered="{!refreshPage}">
  <script>
    window.top.location='{!pageRef}';
  </script>
</apex:outputPanel>
```

> Note that this technique will not work when the record view is displayed in the Salesforce Service Cloud Console, as the record view is not the top-level page and thus, cannot be accessed or refreshed through JavaScript.

Using wrapper classes

A common use case for Visualforce is to present a list of sObjects, allowing a user to select a number of these, and then choose an action to apply to the selected entries. Marking an sObject entry as selected presents a challenge, as it is associating transient information, the selected status, with a record persisted in the Salesforce database.

The solution is to use a **wrapper** class to encapsulate or wrap an sObject instance and some additional information associated with the sObject instance.

In this recipe, we will create a Visualforce page that presents a list of opportunity sObjects, and allows the user to select a number of records to remove from the displayed list.

Getting ready

This recipe makes use of a wrapper class which associates a checkbox with an opportunity sObject record.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SelectOpportunityWrapper.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

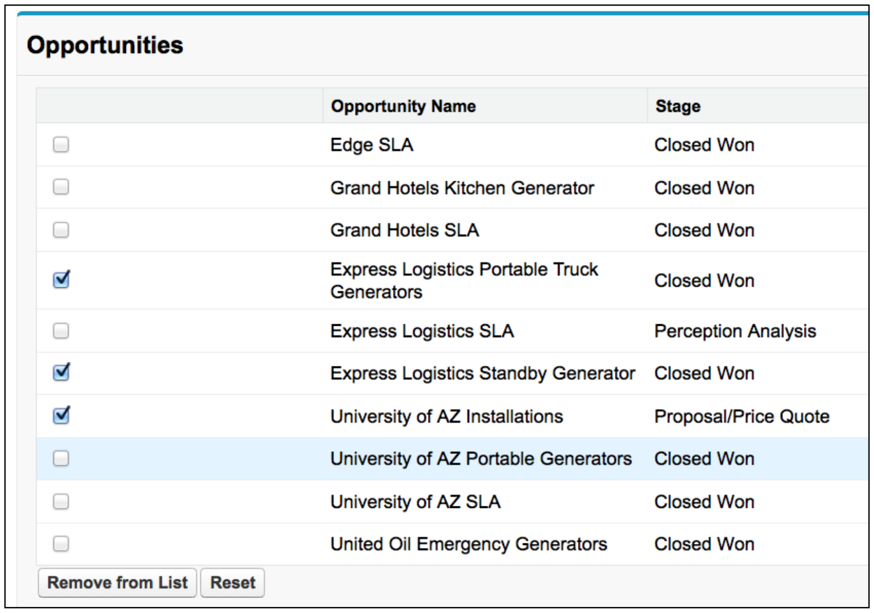
How to do it...

1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SelectOpportunitiesController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `SelectOpportunities` in the **Label** field.
8. Accept the default **SelectOpportunities** that is automatically generated for the **Name** field.
9. Paste the contents of the `SelectOpportunities.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **SelectOpportunities** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following page in your browser displays the **SelectOpportunities** page: `https://<instance>/apex/SelectOpportunities`.

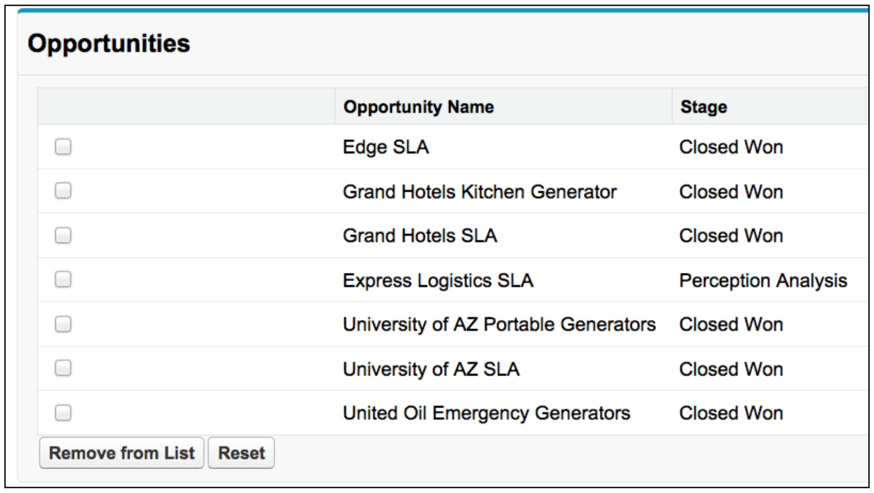
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Opportunities

	Opportunity Name	Stage
☐	Edge SLA	Closed Won
☐	Grand Hotels Kitchen Generator	Closed Won
☐	Grand Hotels SLA	Closed Won
✓	Express Logistics Portable Truck Generators	Closed Won
☐	Express Logistics SLA	Perception Analysis
✓	Express Logistics Standby Generator	Closed Won
✓	University of AZ Installations	Proposal/Price Quote
☐	University of AZ Portable Generators	Closed Won
☐	University of AZ SLA	Closed Won
☐	United Oil Emergency Generators	Closed Won

Selecting a number of checkboxes and clicking on the **Remove from List** button removes the selected rows from the list.



Opportunities

	Opportunity Name	Stage
☐	Edge SLA	Closed Won
☐	Grand Hotels Kitchen Generator	Closed Won
☐	Grand Hotels SLA	Closed Won
☐	Express Logistics SLA	Perception Analysis
☐	University of AZ Portable Generators	Closed Won
☐	University of AZ SLA	Closed Won
☐	United Oil Emergency Generators	Closed Won

The custom controller extracts the opportunities for the page and encapsulates each of these in a `SelectOpportunityWrapper` class.

```
opps=new List<SelectOpportunityWrapper>();
for (Opportunity opp : [select id, Name, StageName
           from Opportunity order by CreatedDate limit 10])
{
   opps.add(new SelectOpportunityWrapper(opp));
}
```

The action method executed when the **Save** button is clicked iterates the list and removes any wrapper class instances where the selected checkbox property is set to `true`.

```
List<SelectOpportunityWrapper> keep=new
      List<SelectOpportunityWrapper>();
for (SelectOpportunityWrapper wrap : opps)
{
  if (!wrap.selected)
  {
    keep.add(wrap);
  }
}

opps=keep;
```

See also

- The *Changing options based on the user input* recipe in this chapter shows how a wrapper class can be used to choose options that will be displayed in a picklist.

Changing options based on the user input

In the earlier recipes, for example, the *Retrieving fields when a lookup is populated* recipe in *Chapter 3*, *Capturing Data Using Forms*, we have seen how to populate the contents of fields in response to user actions. This technique can also be used to change the characteristics of other input fields on the page in response to user selections.

In this recipe we will create a Visualforce page that allows a user to create five task sObject records at once. Each task is associated with a contact selected from a picklist. The picklist options are configured through a series of checkboxes; clearing a checkbox removes the contact from the picklist.

Getting ready

This recipe makes use of a custom controller, so this will need to be present before the Visualforce page can be created.

How to do it...

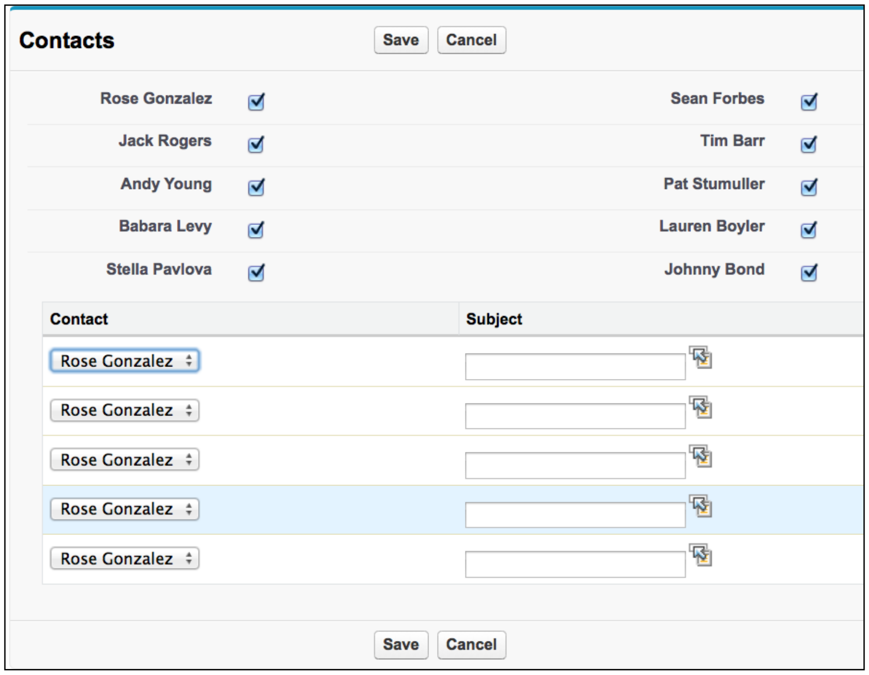
1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SelectContactsController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `SelectContacts` in the **Label** field.
8. Accept the default **SelectContacts** that is automatically generated for the **Name** field.
9. Paste the contents of the `SelectContacts.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **SelectContacts** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **SelectContacts** page: `https://<instance>/apex/SelectContacts`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

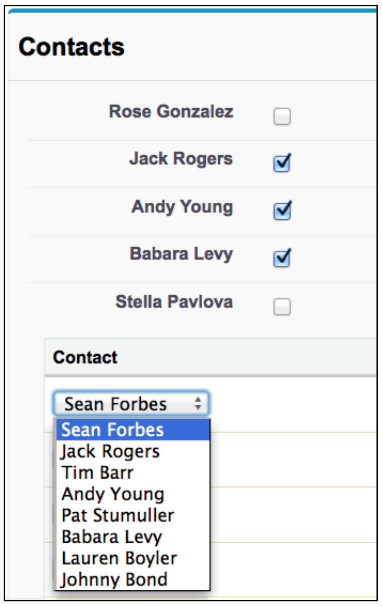
The contact picklist for each task initially contains all 10 contact options.



Rose Gonzalez	✓	Sean Forbes	✓
Jack Rogers	✓	Tim Barr	✓
Andy Young	✓	Pat Stumuller	✓
Babara Levy	✓	Lauren Boyler	✓
Stella Pavlova	✓	Johnny Bond	✓

Contact	Subject
Rose Gonzalez	
Rose Gonzalez	
Rose Gonzalez	
Rose Gonzalez	
Rose Gonzalez	

Clearing the checkboxes of a number of the contacts removes those contacts from the picklist.



Rose Gonzalez	
Jack Rogers	✓
Andy Young	✓
Babara Levy	✓
Stella Pavlova	

Contact

Sean Forbes

Sean Forbes
Jack Rogers
Tim Barr
Andy Young
Pat Stumuller
Babara Levy
Lauren Boyler
Johnny Bond

The displayed contacts are associated with their checkbox through an inner wrapper class in the custom controller.

```
public class ContactWrapper
{
  public Contact cont {get; set;}
  public Boolean available {get; set;}

  public ContactWrapper(Contact inContact)
  {
    cont=inContact;
    available=true;
  }
}
```

The custom controller creates the picklist options based on the value of each contact's checkbox.

```
available=new List<SelectOption>();
for (ContactWrapper cw : contacts)
{
  if (cw.available)
  {
    available.add(new SelectOption(cw.cont.id,
        cw.cont.FirstName + ' ' +
        cw.cont.LastName));
  }
}
```

The **SelectContacts** page iterates the contact wrappers and renders a checkbox next to the contact name. An `<apex:actionsupport />` component is nested in each checkbox to automatically submit the form and rebuild the contact picklists when a contact checkbox is selected or cleared.

```
<apex:repeat value="{!contacts}" var="wrap">
  <apex:pageBlockSectionItem >
    <apex:outputLabel
          value="{!wrap.cont.FirstName} {!wrap.cont.LastName}" />
    <apex:inputCheckbox value="{!wrap.available}">
      <apex:actionSupport event="onchange"
          action="{!availableChanged}" />
    </apex:inputCheckbox>
  </apex:pageBlockSectionItem>
</apex:repeat>
```

See also

- The *Using wrapper classes* recipe in this chapter shows how to encapsulate an sObject record and a checkbox via a wrapper class.
- The *Changing page layout based on the user input* recipe in this chapter shows how the format of a page can change in response to user selections.

Changing page layout based on the user input

When a Visualforce form submission component specifies a `rerender` attribute, this causes a partial refresh of the page to take place, redrawing the specified components based on the result of the submission. This can be used to change the elements on the page based on the user input; for example, to guide the user through creating a record a few fields at a time.

In this recipe, we will create a Visualforce page that allows the user to create an account record. If the user chooses an account type containing the word **customer**, additional fields will be rendered to capture additional customer-specific information.

Getting ready

This recipe makes use of a controller extension, so this needs to be present before the Visualforce page can be created.

How to do it...

1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ChangeContentExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ChangeContent` in the **Label** field.
8. Accept the default **ChangeContent** that is automatically generated for the **Name** field.

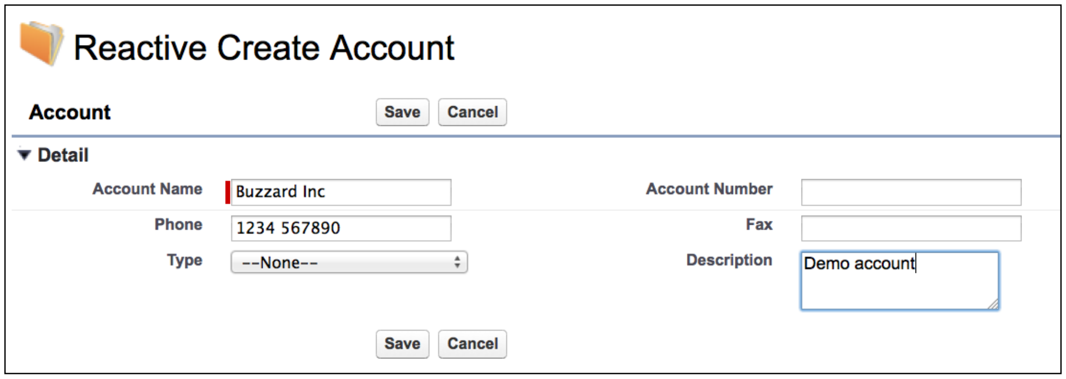
9. Paste the contents of the `ChangeContent.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ChangeContent** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ChangeContent** page: `https://<instance>/apex/ChangeContent`.

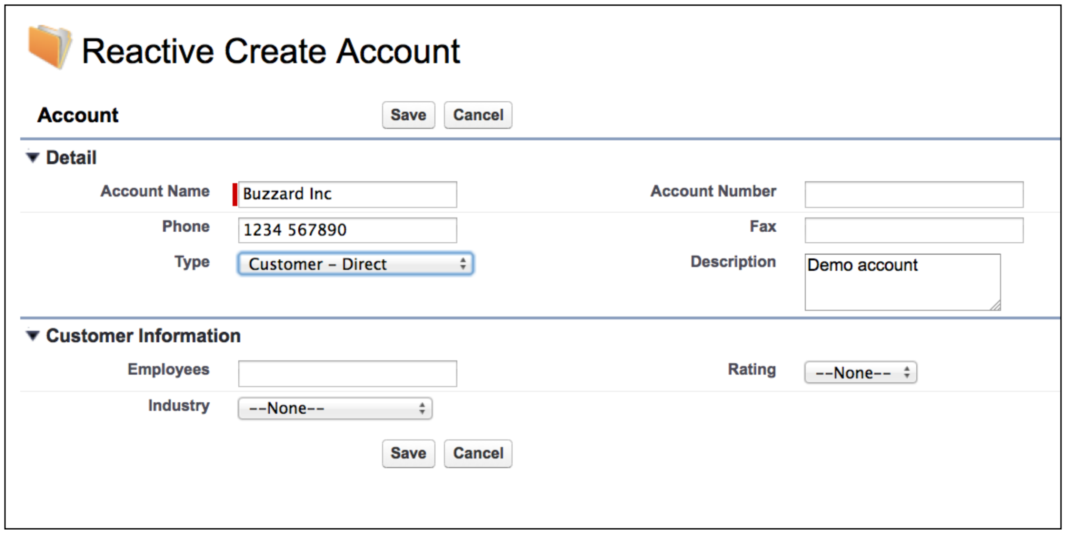
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

The page initially displays the **Detail** section for the user to fill in as shown in the following screenshot:



Account Name	Buzzard Inc	Account Number	
Phone	1234 567890	Fax	
Type	--None--	Description	Demo account

Selecting any customer option from the **Type** picklist causes the **Customer Information** section to be rendered to capture additional customer-specific information.



▼ Detail

Account Name	Buzzard Inc	Account Number	
Phone	1234 567890	Fax	
Type	Customer - Direct	Description	Demo account

▼ Customer Information

Employees		Rating	--None--
Industry	--None--		

The **ChangeContent** page defines a conditionally rendered **Customer Information** section.

```
<apex:pageBlockSection title="Customer Information"
        rendered="{!showCustomerSection}">
  <apex:inputField value="{!Account.NumberOfEmployees}"/>
  <apex:inputField value="{!Account.Rating}"/>
  <apex:inputField value="{!Account.Industry}"/>
</apex:pageBlockSection>
```

The **Type** picklist contains a nested `<apex:actionSupport />` component to automatically submit the form when a value is selected.

```
<apex:actionRegion >
  <apex:pageBlockSection >
    <apex:inputField value="{!Account.Type}">
      <apex:actionSupport event="onchange" action="{!typeChanged}"
        rerender="customersection" />
    </apex:inputField>
    <apex:inputField value="{!Account.Description}"/>
  </apex:pageBlockSection>
</apex:actionRegion>
```

Note that the **Type** field is nested inside an action region. This allows the form to be submitted even if the required **Account Name** field has not been populated.

The action method invoked when the `Type` value changes sets the `showCustomerSection` property based on the selected value.

```
Account acc=(Account) stdCtrl.getRecord();
if (acc.Type.toLowerCase().contains('customer'))
{
  showCustomerSection=true;
}
else
{
  showCustomerSection=false;
}
```

See also

- The *Changing options based on the user input* recipe in this chapter shows how to alter picklist options based on user selections.

Form-based searching

Standard Salesforce searching looks for any occurrence of a supplied text value in all searchable fields of one or more sObject types. In the scenario where a user is interested in the occurrence of the text value in a particular field, this can lead to a number of unwanted results. For example, searching for an account whose name contains the text **United** will also retrieve all accounts with a mailing or billing address in the United Kingdom.

Form-based searching allows a user to specify the text that should be present in particular fields in order to be considered a match.

In this recipe, we will create a Visualforce page to allow a user to search for accounts that contain specified text in the **Account Name** or **Website** fields, or where the name entered in the **Industry** field matches one of a number of options.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

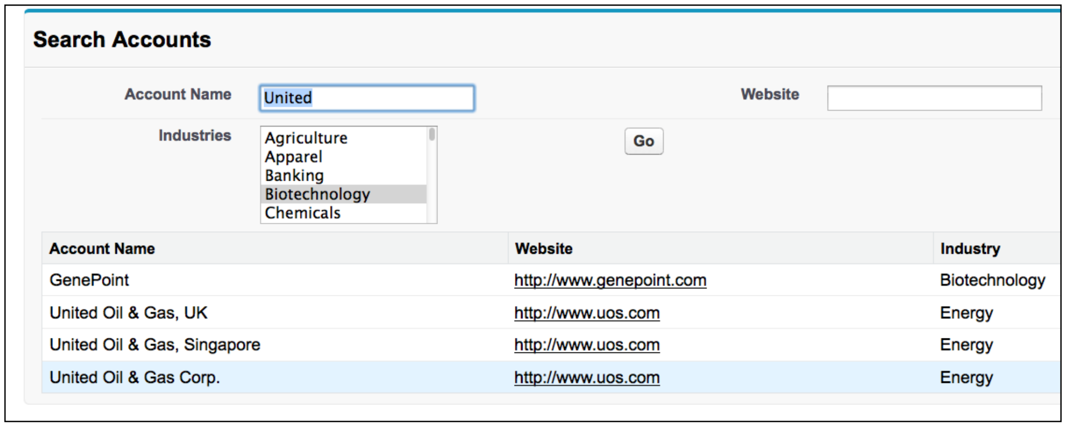
How to do it...

1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SearchAccountsController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `SearchAccounts` in the **Label** field.
8. Accept the default **SearchAccounts** that is automatically generated for the **Name** field.
9. Paste the contents of the `SearchAccounts.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **SearchAccounts** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **SearchAccounts** page: `https://<instance>/apex/SearchAccounts`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Search Accounts

Account Name	United
Website	
Industries	Agriculture, Apparel, Banking, Biotechnology, Chemicals

Account Name	Website	Industry
GenePoint	http://www.genepoint.com	Biotechnology
United Oil & Gas, UK	http://www.uos.com	Energy
United Oil & Gas, Singapore	http://www.uos.com	Energy
United Oil & Gas Corp.	http://www.uos.com	Energy

The **SearchAccountsController** page utilizes an account record to capture the **Account Name** and **Website** fields for searching. As the **Industries** picklist allows multiple values to be selected, the standard `Industry` field cannot be used to capture this. Instead, the controller interrogates the database schema to find the available options.

```
Schema.DescribeFieldResult fieldDesc =
  Account.Industry.getDescribe();
List<Schema.PicklistEntry> plEntries =
  fieldDesc.getPicklistValues();

for (Schema.PickListEntry plEntry : plEntries)
{
  SelectOption option=new
    SelectOption(plEntry.getValue(),
      plEntry.getLabel());
  industryOptions.add(option);
}
```

The action method executed by the **Go** button determines which of the search criteria fields have been populated and constructs a dynamic SOQL query to retrieve the matches.

```
if ( (null!=industries) && (industries.size()>0) )
{
  for (Integer idx=0; idx<industries.size(); idx++)
  {
    whereStr+=' OR Industry = \'' + industries[idx] + '\'';
  }
}

if (''!=whereStr)
{
  String queryStr='select Id, Name, Website, Industry from
    Account where ' + whereStr.substring(4);
  results=Database.query(queryStr);
}
```

Note that a substring of the `whereStr` variable is used to generate the query, starting at position 4. This eliminates the initial `' OR '` added with the first selected industry.

See also

- The *Reacting to URL parameters* recipe in *Chapter 1*, *General Utilities* shows how to execute a text search based on a URL parameter.

5
Managing Multiple Records

In this chapter, we will cover the following recipes:

- Preventing duplicates by searching before creating
- Editing a record and its parent
- Managing a list of records
- Converting a lead
- Managing a hierarchy of records
- Inline-editing a record from a list
- Creating a Visualforce report
- Loading records asynchronously

Introduction

When Visualforce pages utilize a controller extension or custom controller, they can retrieve additional records via SOQL queries. This allows pages to manage more than one record, regardless of the record sObject types or whether there is any relationship between the records.

> The **Salesforce Object Query Language** (**SOQL**) allows information to be retrieved from the Salesforce database based on supplied criteria. It has an SQL-like syntax, but does not support advanced operations such as wildcard field lists.

In this chapter, we will explore a number of scenarios to manage multiple records on a single page, ranging from a single record and its parent to a deep and wide hierarchy.

We will also see how Visualforce can be used to present details of a collection of records in response to user-specified criteria, in order to search for existing matches before creating a new record or to produce a custom report page.

Preventing duplicates by searching before creating

A very common problem in Salesforce implementations is duplicate data. While training users to search for existing records before creating can help, this relies on users remembering to follow the process, which can be especially problematic if some users create records on an infrequent basis.

In this recipe, we will create a Visualforce page that overrides the lead sObject create button and requires a user to search for existing matching records before they are allowed to create a new record. In order to avoid the user having to rekey data in the event that no matches are found, the search criteria is carried through to the create page.

Getting ready

This recipe makes use of a controller extension, so this must be created before the Visualforce page.

> This page does not make use of the standard controller, but the page is required to use the lead standard controller in order to be able to override a standard button.

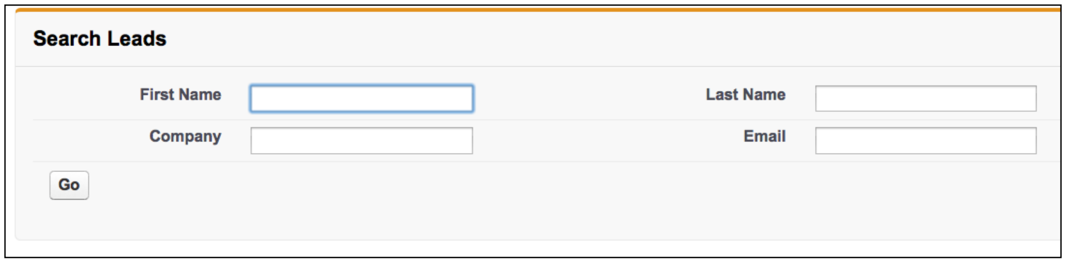
How to do it...

1. Navigate to the Apex Classes setup page, by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SearchLeadsExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.

7. Enter `SearchLeads` in the **Label** field.
8. Accept the default **SearchLeads** that is automatically generated for the **Name** field.
9. Paste the contents of the `SearchLeads.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **SearchLeadspage** and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Finally, configure the New Lead override by navigating to **Your Name** | **Setup** | **Customize** | **Leads** | **Buttons, Links, and Actions**.
14. Locate the **New** entry on the resulting page and click on the **Edit** link.
15. On the following page, locate the **Override With** entry, check the **Visualforce Page** radio button, and choose **SearchLeads** from the list of available pages.
16. Click on the **Save** button.

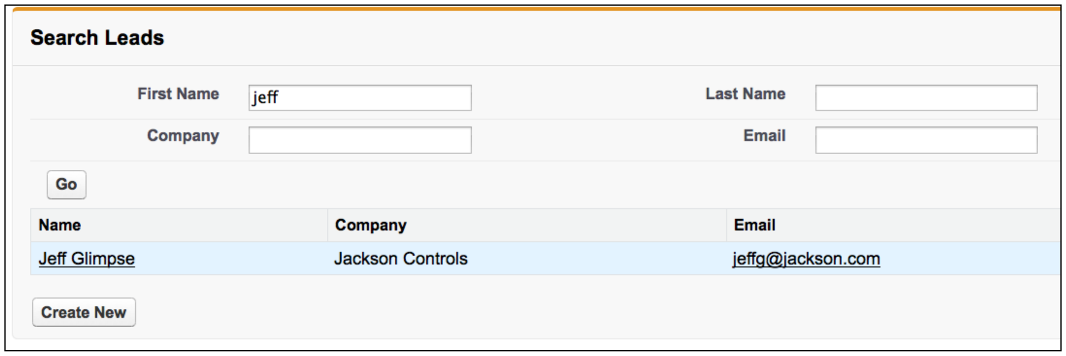
How it works...

Creating a new lead through any mechanism, for example, via the **New** button on the **Leads** tab or the **Create New** menu on the sidebar, displays the **Search Leads** page. Initially, only the search criteria fields and the **Go** button are displayed.


Search Leads

First Name		Last Name	
Company		Email	

Once the user has searched for records, the **Create New** button is displayed along with any matches.

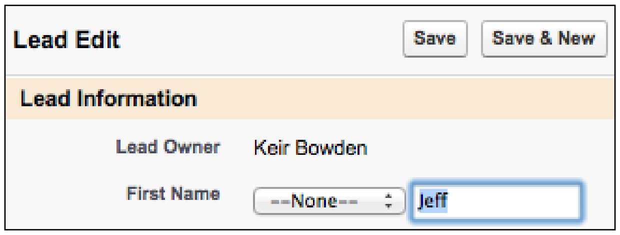


Search Leads

First Name	jeff	Last Name	
Company		Email	

Name	Company	Email
Jeff Glimpse	Jackson Controls	jeffg@jackson.com

Clicking on the **Create New** button takes the user to the standard New Lead page with the search criteria prefilled into the appropriate fields.



Lead Edit

Lead Information

Lead Owner	Keir Bowden
First Name	--None-- Jeff

The **Create New** button is conditionally rendered based on a controller property set when the user searches.

```
<apex:commandButton value="Create New" action="{!createNew}"
    rendered="{!executed}" />
```

The action method executed by the **Go** button determines which of the search criteria fields have been populated, constructs a dynamic SOQL query to retrieve the matches, and sets the property to indicate that a search has been executed. If no search criteria has been entered, an error message is displayed.

```
String whereStr='';
if (!String.IsBlank(searchLead.FirstName))
{
  String wcFName='%' + searchLead.FirstName + '%';
    whereStr+=' OR FirstName LIKE \'' + wcFName + '\'';
}

if (''!=whereStr)
```

```
{
  String queryStr='select Id, FirstName, LastName, Company,
    Email from Leadwhere ' + whereStr.substring(4);
  results=Database.query(queryStr);

  executed=true;
}
else
{
  ApexPages.addMessage(
    newApexPages.Message(
      ApexPages.Severity.ERROR,
      'Please enter the search criteria'));
}
```

The action method executed by the **Create New** button redirects the user to the standard **New Lead** page and includes parameters to prefill the lead fields with the search criteria entered by the user.

```
PageReferencepr=newPageReference('/00Q/e');
pr.getParameters().put('nooverride', '1');

if (!String.IsBlank(searchLead.FirstName))
{
  pr.getParameters().put('name_firstlea2',
        searchLead.FirstName);
}
```

This recipe relies on the HTML element IDs for standard lead fields not changing in the future. Salesforce does not support this mechanism of passing parameters to prefill fields, although it has been successfully used for a number of years.

See also

- The *Reacting to URL parameters* recipe in *Chapter 1*, *General Utilities* shows how to execute a text search based on a URL parameter.

Editing a record and its parent

A Visualforce page managed by a standard controller can provide the edit capability for a record and its parent. However, when the standard controller **Save** method is invoked, the object graph is not traversed and only the record being managed by the controller is saved.

In this recipe we will create a Visualforce page to allow a user to edit fields from a contact and its parent account. Saving the record will also apply any changes made to the parent account record.

Getting ready

This recipe makes use of an extension controller, so this will need to be created before the Visualforce page.

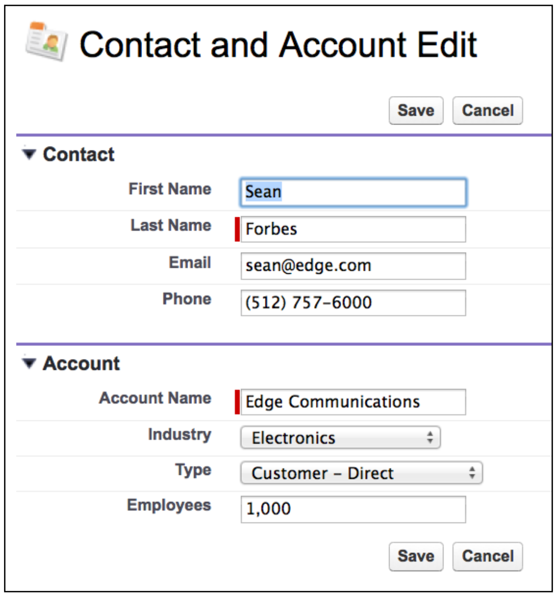
How to do it...

1. First, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ContactAndAccountEditExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ContactAndAccountEdit` in the **Label** field.
8. Accept the default **ContactAndAccountEdit** that is automatically generated for the **Name** field.
9. Paste the contents of the `ContactAndAccountEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ContactAndAccountEdit** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactAndAccountEdit** page: `https://<instance>/apex/ContactAndAccountEdit?id=<contact_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the ID of any contact record in your organization.



Contact and Account Edit

Contact

First Name	Sean
Last Name	Forbes
Email	sean@edge.com
Phone	(512) 757-6000

Account

Account Name	Edge Communications
Industry	Electronics
Type	Customer - Direct
Employees	1,000

The controller extension retrieves the related account record based on the contact managed by the standard controller. If the contact does not have a parent account, a new account record is created.

```
cont=(Contact) stdCtrl.getRecord();
if ( (null!=cont.Id) && (null!=cont.AccountId) )
{
  acc=[select id, Name, Type, NumberOfEmployees, Industry
      from Account where id=:cont.AccountId];
}
else
{
  acc=new Account();
}
```

The action method invoked by clicking on the **Save** button upserts the account record, sets the parent account ID of the contact record if a new account record is created, and then delegates to the standard controller `save` action method to update the contact record.

```
upsert acc;

if (null==cont.AccountId)
{
  cont.AccountId=acc.id;
}

return stdCtrl.save();
```

See also

- The *Managing a list of records* recipe in this chapter shows how to maintain a list of sObject records of a single type.
- The *Managing a hierarchy of records* recipe in this chapter shows how to maintain a hierarchy of sObject records of different types.

Managing a list of records

Salesforce users often require the capability to work with a number of records at once. For example, a sales user may be communicating with a number of contacts, while he/she may also be creating or deleting contacts in response to information received through a number of channels. The Salesforce enhanced list view functionality allows a set of records that share a common record type to be inline-edited, but doesn't provide a way to add or remove records.

In this recipe, we will create a Visualforce page to allow a user to edit the details of a collection of existing contact records, and create or delete records dynamically. Upon saving the list, existing records will be updated, any new records will be inserted, and any records previously deleted from the collection will also be deleted from the database.

Getting ready

This recipe makes use of a wrapper class that needs to be created before the Visualforce page.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ContactKeyWrapper.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

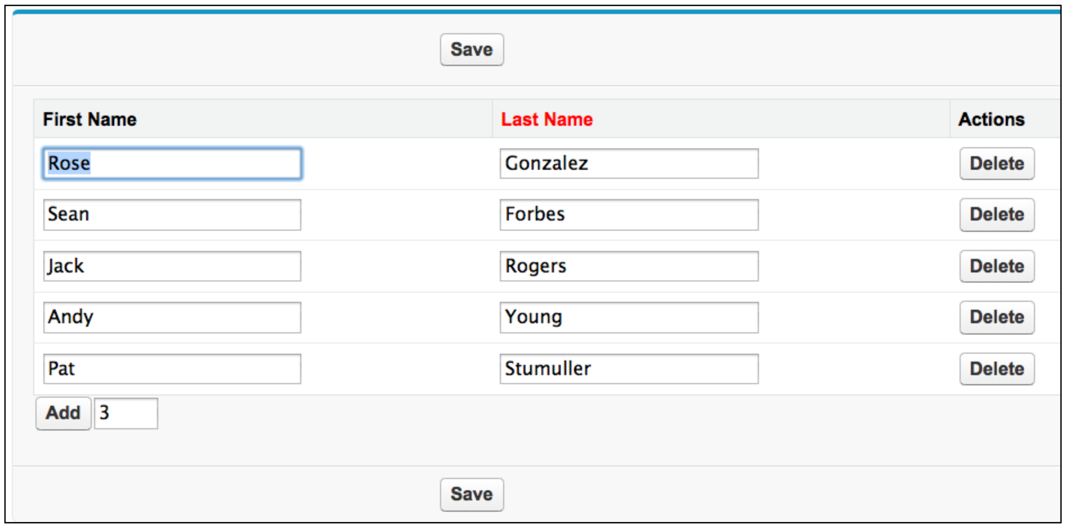
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ContactListEditController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ContactListEdit` in the **Label** field.
8. Accept the default **ContactListEdit** that is automatically generated for the **Name** field.
9. Paste the contents of the `ContactListEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ContactListEdit** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ContactListEdit** page: `https://<instance>/apex/ContactListEdit`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



First Name	Last Name	Actions
Rose	Gonzalez	Delete
Sean	Forbes	Delete
Jack	Rogers	Delete
Andy	Young	Delete
Pat	Stumuller	Delete

Each contact record managed by the custom controller is encapsulated in an instance of the `ContactKeyWrapper` class. This ensures that both new and existing records have a unique key even if they have not been written to the database.

```
wrappers=new List<ContactKeyWrapper>();
List<Contact> contacts=[select id, FirstName, LastName from Contact
order by CreatedDate limit 5];
for (Contact cont : contacts)
{
  wrappers.add(new ContactKeyWrapper(mainKey++, cont));
}
```

The **Delete** button uses an `<apex:param />` component to send the key of the contact wrapper record to be deleted.

```
<apex:commandButton value="Delete" action="{!removeItem}"
rerender="block">
  <apex:param name="keyToDelete" value="{!wrap.key}"
  assignTo="{!keyToDelete}" />
</apex:commandButton>
```

The action method invoked by clicking on the **Delete** button locates the record identified by the controller property the parameter is assigned to and inspects its `id` field. If this is `null`, the record can simply be removed from the list being managed, as it has not been written to the database yet. If it is not `null`, the record is not only removed from the list, but also added to a list of records to be deleted when the changes are saved.

```
for (ContactKeyWrapper wrap : wrappers)
{
  if (wrap.key==keyToDelete)
  {
    found=true;
    if (null!=wrap.cont.id)
    {
      toDelete.add(wrap.cont);
    }
    break;
  }

...

if (found)
{
  wrappers.remove(idx);
}
```

The user can add one or more records to the list by entering the number of records and clicking on the **Add** button. This creates the appropriate number of instances of the `ContactKeyWrapper` class and appends these to the list.

Clicking on the **Save** button iterates the list of records and identifies those that have fields populated. These records are validated to ensure that all the required fields are populated. The method then executes a DML `upsert` for the populated records, updating existing records and inserting new ones. The records in the `toDelete` list are then deleted from the database.

See also

- The *Editing a record and its parent* recipe in this chapter shows how to maintain an sObject record and its parent record from a single page.
- The *Managing a hierarchy of records* recipe in this chapter shows how to maintain a hierarchy of sObject records of different types.

Converting a lead

The standard Salesforce lead conversion page allows a user to create a new account and contact record or merge with existing records, and optionally create a new opportunity record. Information from the lead is copied to the existing or new records based on the lead field mapping configuration. If a user wishes to specify information after clicking on the **Convert** button, they must exit the conversion and edit the lead record.

In this recipe, we will create a Visualforce page to allow a user to convert a lead and in addition to creating or merging with existing records, populate additional fields on the opportunity that is created as part of the conversion.

Getting ready

This recipe makes use of a controller extension so this must be present before the Visualforce page can be created.

How to do it...

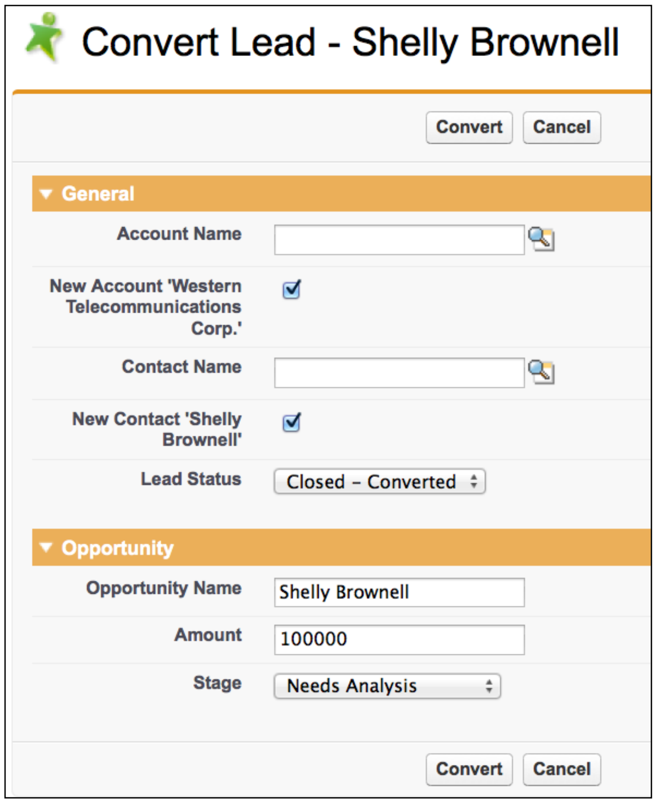
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `LeadConvertExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `LeadConvert` in the **Label** field.
8. Accept the default **LeadConvert** that is automatically generated for the **Name** field.
9. Paste the contents of the `LeadConvert.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **LeadConvert** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **LeadConvert** page: `https://<instance>/apex/LeadConvert?id=<lead_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<lead_id>` is the ID of any unconverted lead record in your organization.


Convert Lead - Shelly Brownell

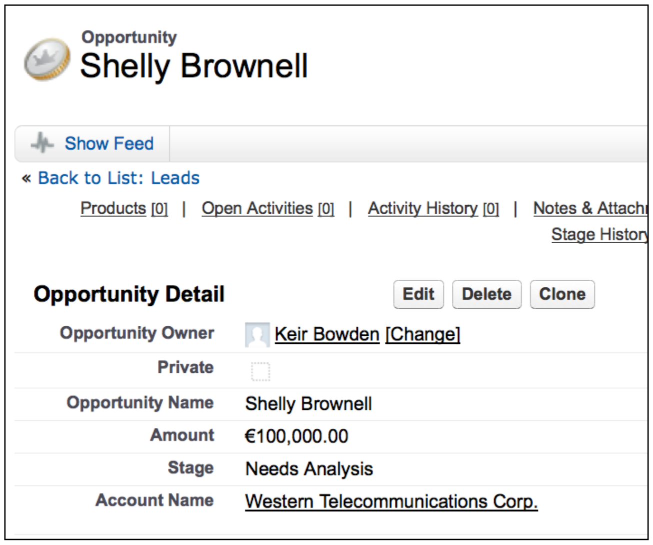
General

Account Name	
New Account 'Western Telecommunications Corp.'	✓
Contact Name	
New Contact 'Shelly Brownell'	✓
Lead Status	Closed – Converted

Opportunity

Opportunity Name	Shelly Brownell
Amount	100000
Stage	Needs Analysis

Populating the opportunity fields and clicking on the **Convert** button converts the lead to create new or merge with existing account and contact records, and redirects the user to the new opportunity record.



Opportunity Detail

Opportunity Owner	Keir Bowden [Change]
Private	☐
Opportunity Name	Shelly Brownell
Amount	€100,000.00
Stage	Needs Analysis
Account Name	Western Telecommunications Corp.

A case record is used as the carrier object for the account and contact information.

A carrier object is an sObject that is used to hold a set of fields for a page, rather than for its intended purpose. In this recipe, we need to hold references to account and contact records. While this could be achieved using `Id` properties, using a case sObject allows the `<apex:inputField />` standard component to be used, which generates appropriate HTML markup for a record lookup.

```
<apex:pageBlockSection title="General" columns="1">
  <apex:inputField value="{!carrier.AccountId}"/>
  <apex:pageBlockSectionItem>
    <apex:outputLabel value="New Account '{!Lead.Company}'" />
    <apex:inputCheckbox value="{!newAccount}" />
  </apex:pageBlockSectionItem>
  <apex:inputField value="{!carrier.ContactId}"/>
  <apex:pageBlockSectionItem>
    <apex:outputLabel value="New Contact '{!Lead.FirstName} {!Lead.
LastName}'" />
    <apex:inputCheckbox value="{!newContact}" />
  </apex:pageBlockSectionItem>
```

The list of available status field values for the converted lead is created by extracting all values from the `LeadStatus` database table where the `IsConverted` field is set to `true`, and creating `SelectOption` instances for each value.

```
List<LeadStatus> states=[select id, MasterLabel from
    LeadStatus where IsConverted=true];

for (LeadStatus state : states)
{
  if (null==convertedStatus)
  {
    convertedStatus=state.MasterLabel;
  }

  SelectOption option=new
  SelectOption(state.MasterLabel, state.MasterLabel);
  result.add(option);
}
```

If values are not populated for the account and contact lookup fields, the controller creates a new account and contact record. The checkboxes on the page are provided as a visual indicator to the user and the controller does not use the values.

```
if (null!=carrier.AccountId)
{
  leadConvert.setAccountId(carrier.AccountId);
}
```

If the **Opportunity Name** field is populated, a new opportunity is also created when the lead is converted.

```
if (String.IsBlank(opp.Name))
{
  leadConvert.setDoNotCreateOpportunity(true);
}
else
{
  leadConvert.setOpportunityName(opp.Name);
}
```

The additional opportunity fields are captured to an opportunity record instance in the controller, which is used to update the new opportunity after the lead conversion has taken place.

```
if (!String.IsBlank(opp.Name))
{
  Opportunity newOpp=[select id from Opportunity where
  id=:convertResult.getOpportunityId()];
```

```
    if (!String.IsBlank(opp.StageName))
    {
      newOpp.StageName=opp.StageName;
    }
```

Finally, the user is directed to one of the accounts or opportunity view pages depending on whether a new opportunity was created as part of the lead conversion or not.

```
  if (!String.IsBlank(opp.Name))
  {

  ...

    result=new PageReference('/' +
           convertResult.getOpportunityId());
  }
  else
  {
    result=new PageReference('/' + convertResult.getAccountId());
  }
```

There's more...

This Visualforce page can be configured as an override to the standard lead convert button.

1. Navigate to **Your Name** | **Setup** | **Customize** | **Leads** | **Buttons and Links**.
2. Locate the **Convert** entry on the resulting page and click on the **Edit** link.
3. On the following page, locate the **Override With** entry, check the **Visualforce Page** radio button, and choose **LeadConvert** from the list of available pages.
4. Click on the **Save** button.

Managing a hierarchy of records

Salesforce records often form part of a deep and wide hierarchy; for example, an account can contain a number of cases, each of which can have a number of comments associated with them. Creating and maintaining the elements of the hierarchy in isolation is a cumbersome and time-consuming task, as to add a comment a user must click through from the account record to the case record, and then click on the **New** button on the case comments related list to open the new comment page.

In this recipe we will create a Visualforce page that allows a user to maintain an account, its associated cases, and the comments associated with those cases. The user can update or delete existing records, or create new records at any level of the hierarchy.

Getting ready

This recipe makes use of two wrapper classes which need to be created before the Visualforce page and controller.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `CaseCommentKeyWrapper.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
6. Click on the **New** button.
7. Paste the contents of the `CaseKeyWrapper.cls` Apex class from the code download into the Apex Class area.
8. Click on the **Save** button.

How to do it...

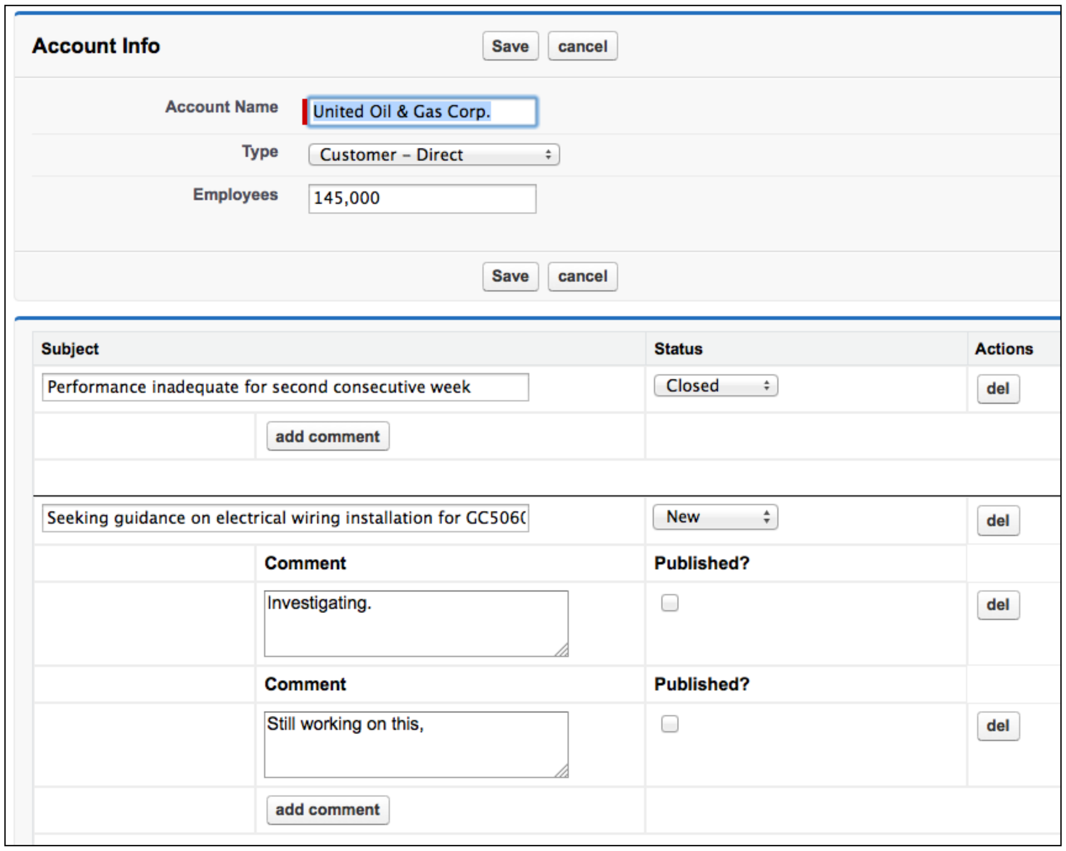
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `AccountCasesCommentsExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `AccountCasesCommentsEdit` in the **Label** field.
8. Accept the default **AccountCasesCommentsEdit** that is automatically generated for the **Name** field.
9. Paste the contents of the `AccountCasesCommentsEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

11. Locate the entry for the **AccountCasesCommentsEdit** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the custom setting maintenance settings page: `https://<instance>/apex/AccountCasesCommentsEdit?id=<account_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<account_id>` is the ID of any account record in your organization.



Account Name	United Oil & Gas Corp.
Type	Customer - Direct
Employees	145,000

Subject	Status	Actions
Performance inadequate for second consecutive week	Closed	del
Seeking guidance on electrical wiring installation for GC506(	New	del

Comment	Published?	
Investigating.	☐	del
Still working on this,	☐	del

Each case and child case comments managed by the controller are encapsulated in an instance of the `CaseKeyWrapper` and `CaseCommentKeyWrapper` classes respectively. This ensures that both new and existing records have a unique key even if they have not been written to the database. The cases are encapsulated in the controller extension constructor.

```
List<Case> cases=[select id, Status, Subject,
  (select id, CommentBody, IsPublished, ParentId from
      CaseComments)
  from Case
  whereAccountId=:stdCtrl.getId()];

caseWrappers=new list<CaseKeyWrapper>();
for (Case cs : cases)
{
  caseWrappers.add(new CaseKeyWrapper(key++, cs,
          cs.CaseComments));
}
```

While the comments are encapsulated in the `CaseKeyWrapper` constructor.

```
comments=new List<CaseCommentKeyWrapper>();

if (null!=inComments)
{
  for (CaseComment cc : inComments)
  {
    comments.add(new CaseCommentKeyWrapper(commentKey++,
                                          cc));
  }
}
```

The `commentKey` is an integer property of the `CaseKeyWrapper` class that starts at 1 and is incremented for each comment processed. This means that it must be concatenated with the key from its parent `CaseKeyWrapper` class in order to generate a unique value.

The **del** button to delete a case uses an `<apex:param />` component to send the key of the case wrapper record to be deleted.

```
<apex:commandButton value="del" action="{!deleteCase}"
rerender="list">
  <apex:param name="caseToDel" value="CS{!caseWrap.key}"
  assignTo="{!caseToDel}"/>
</apex:commandButton>
```

The **del** button to delete a case comment also uses an `<apex:param />` component, but requires the keys of both the comment wrapper and its parent case wrapper to be encoded in the parameter passed back by the controller in order to generate a unique value for the comment, as explained earlier:

```
<apex:commandButton value="del" action="{!deleteCaseComment}"
      rerender="list">
  <apex:param name="ccToDel"
      value="CS{!caseWrap.key}:CC{!commentWrap.key}"
      assignTo="{!ccToDel}"/>
</apex:commandButton>
```

The action method associated with each **del** button locates the record identified by the controller property the parameter is assigned to and inspects its `id` field. If this is `null`, the record can simply be removed from the list being managed, as it has not been written to the database yet. If it is not `null`, the record is not only removed from the list, but also added to a list of records to be deleted when the changes are saved.

Clicking on the **Save** button iterates the list of case wrappers, extracts the encapsulated cases, and executes a DML `upsert`. The case wrappers are then iterated again, and the child case comments are extracted from their wrapper class instances. The `parentId` field of any newly added comments will be `null`, so this will be set to `id` of the parent case.

```
List<CaseComment>caseComments=new List<CaseComment>();
for (CaseKeyWrapper wrapper : caseWrappers)
{
  for (CaseCommentKeywrapperccWrapper : wrapper.comments)
  {
    CaseComment comment=ccWrapper.comment;
    if (null==comment.ParentId)
    {
      comment.parentId=wrapper.cs.id;
    }
    caseComments.add(comment);
  }
}
```

A DML `upsert` operation is then carried out for the case comment records. Finally, DML operations are carried out to remove any deleted cases or comments from the database.

See also

- The *Managing a list of records* recipe in this chapter shows how to maintain a list of sObject records of a single type.
- The *Managing a record and its parent* recipe in this chapter shows how to maintain an sObject record and its parent record from a single page.

Inline-editing a record from a list

In the previous recipes, all records in a list have been editable. This works well when the user is expecting to edit a number of records, but is not a great experience for users that predominantly view records. In this situation, it is better to give the user a read-only view of the records and a rapid way to edit a record if required.

In this recipe, we will create a Visualforce page that presents a list of contacts in read-only mode. The user may double-click on any field in order to edit the record details. Any changes to the contact records are stored locally, until the user chooses to save or discard them.

> Standard Visualforce markup provides support for inline editing, but this requires each field to be double-clicked to edit individually.

Getting ready

This recipe makes use of a custom controller that must be present before the Visualforce page can be created.

How to do it...

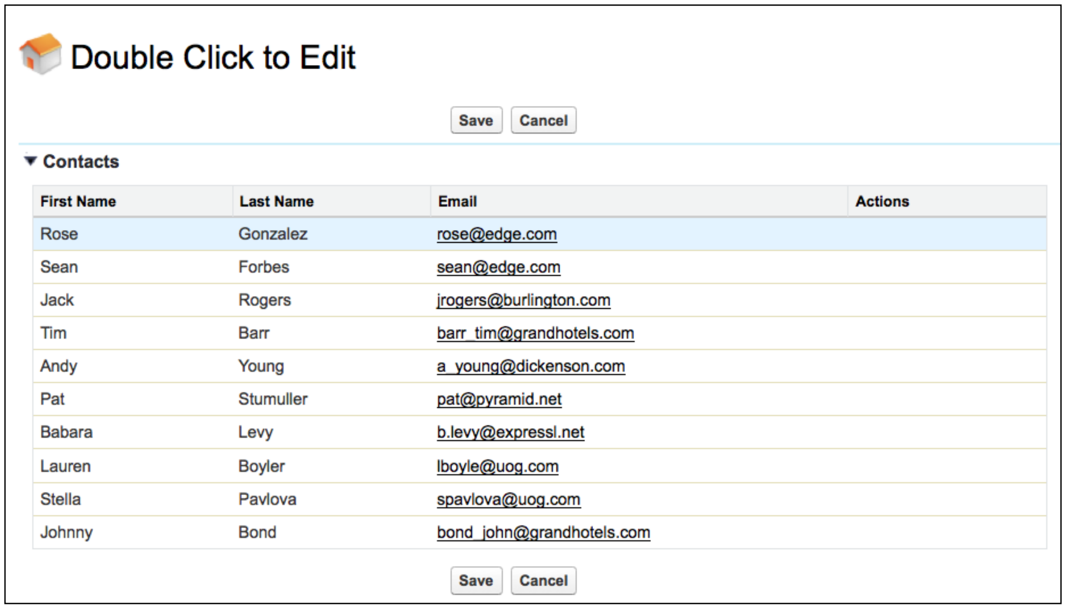
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ContactDblClickEditController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ContactDblClickEdit` in the **Label** field.
8. Accept the default **ContactDblClickEdit** that is automatically generated for the **Name** field.
9. Paste the contents of the `ContactDblClickEdit.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ContactDblClickEdit** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

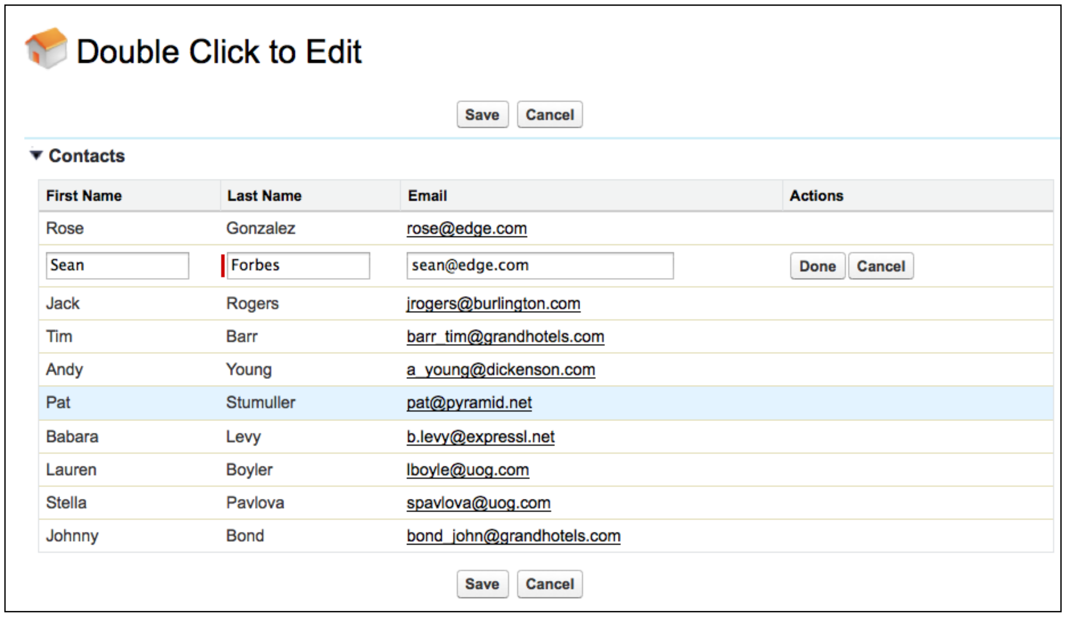
Opening the following URL in your browser displays the custom setting maintenance settings page: `https://<instance>/apex/ContactDblClickEdit`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



First Name	Last Name	Email	Actions
Rose	Gonzalez	rose@edge.com	
Sean	Forbes	sean@edge.com	
Jack	Rogers	jrogers@burlington.com	
Tim	Barr	barr_tim@grandhotels.com	
Andy	Young	a_young@dickenson.com	
Pat	Stumuller	pat@pyramid.net	
Babara	Levy	b.levy@expressl.net	
Lauren	Boyler	lboyle@uog.com	
Stella	Pavlova	spavlova@uog.com	
Johnny	Bond	bond_john@grandhotels.com	

Double-clicking on any of the fields changes the specific row into edit mode.



Double Click to Edit

▼ Contacts

First Name	Last Name	Email	Actions
Rose	Gonzalez	rose@edge.com	
Sean	Forbes	sean@edge.com	Done Cancel
Jack	Rogers	jrogers@burlington.com	
Tim	Barr	barr_tim@grandhotels.com	
Andy	Young	a_young@dickenson.com	
Pat	Stumuller	pat@pyramid.net	
Babara	Levy	b.levy@expressl.net	
Lauren	Boyler	lboyle@uog.com	
Stella	Pavlova	spavlova@uog.com	
Johnny	Bond	bond_john@grandhotels.com	

Clicking on the **Done** button applies any changes made to the record stored in the controller and reverts the row to read-only mode. Clicking on **Cancel** not only reverts the row to read-only mode, but also discards any changes that the user has made.

The Visualforce page defines two sets of columns for the list of records: one set to render for records in read-only mode, and another to render input fields if the record is in edit mode.

```
<apex:pageBlockTable style="width:75%" value="{!contacts}"
var="contact" >
  <apex:column style="width:10%" headerValue="First Name"
    value="{!contact.FirstName}"
    rendered="{!contact.id!=chosenContactId}"
    ondblclick="editContact('{!contact.id}')" />

   ...
  <apex:column style="width:10%" headerValue="First Name"
    rendered="{!contact.id==chosenContactId}">
    <apex:inputField style="width:80%"
    value="{!contact.FirstName}" />
  </apex:column>
</apex:pageBlockTable>
```

The `ondblclick` event handler for the read-only columns executes an action function that simply passes a parameter to the controller, identifying the record that the user would like to edit.

```
<apex:actionFunction name="editContact" rerender="contacts, msgs">
  <apex:param name="chosenContactId" value=""
    assignTo="{!chosenContactId}" />
</apex:actionFunction>
```

This causes the page to be refreshed and the edit-mode columns to be rendered for the record whose ID matches the `chosenContactId` controller property.

The buttons rendered in the action column for the record in edit mode are both associated with the same action method; this simply clears the value of the `chosenContactId` controller property. If the user is keeping their edits, the form submission will automatically update the details of the stored record. If the user chooses to cancel the edit, the `immediate="true"` attribute on the button will discard any data entered by the user and revert the page to display the last saved data from the controller.

```
<apex:commandButton action="{!done}" rerender="contacts, msgs"
    value="Done" />
<apex:commandButton action="{!done}" rerender="contacts, msgs"
    value="Cancel" immediate="true" />
```

[Note that editing a record and then double-clicking a different row will save the outstanding edit as though the **Done** button had been clicked.]

See also

- The *Managing a list of records* recipe in this chapter shows how to create an editable list of records.
- The *Managing a hierarchy of records* recipe in this chapter shows how to maintain a hierarchy of sObject records of different types.

Creating a Visualforce report

Salesforce provides powerful analytic capabilities through the report and dashboard builders, but there are times when reporting requirements cannot be satisfied through the standard functionality; for example, where data from a number of different sources is required to be presented in multiple formats. In this scenario, Visualforce can give fine-grained control over the layout of the results, while a custom controller allows retrieval of any accessible data in the system.

In this recipe, we will create a Visualforce report that retrieves all cases matching criteria specified by the user and outputs these in a tabular format containing details of all cases, keeping a running total of the number of cases with the same status and origin. Two tables that provide the total count of cases for each status and origin value follow this.

> Note that the replacement of standard reporting functionality with Visualforce should only be carried out as a last resort. Coding complex reporting requirements can consume significant amounts of time and removes the capability for users to customize reports to their own requirements.

Getting ready

This recipe makes use of a wrapper class that needs to be created before the Visualforce page and controller.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `CaseAndTotals.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

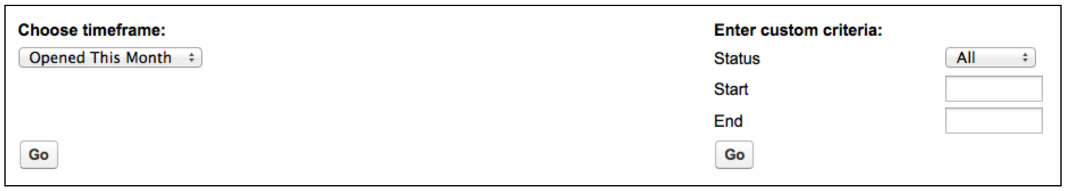
How to do it...

1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `CasesReportController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `CasesReport` in the **Label** field.
8. Accept the default **CasesReport** that is automatically generated for the **Name** field.
9. Paste the contents of the `CasesReport.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **CasesReport** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

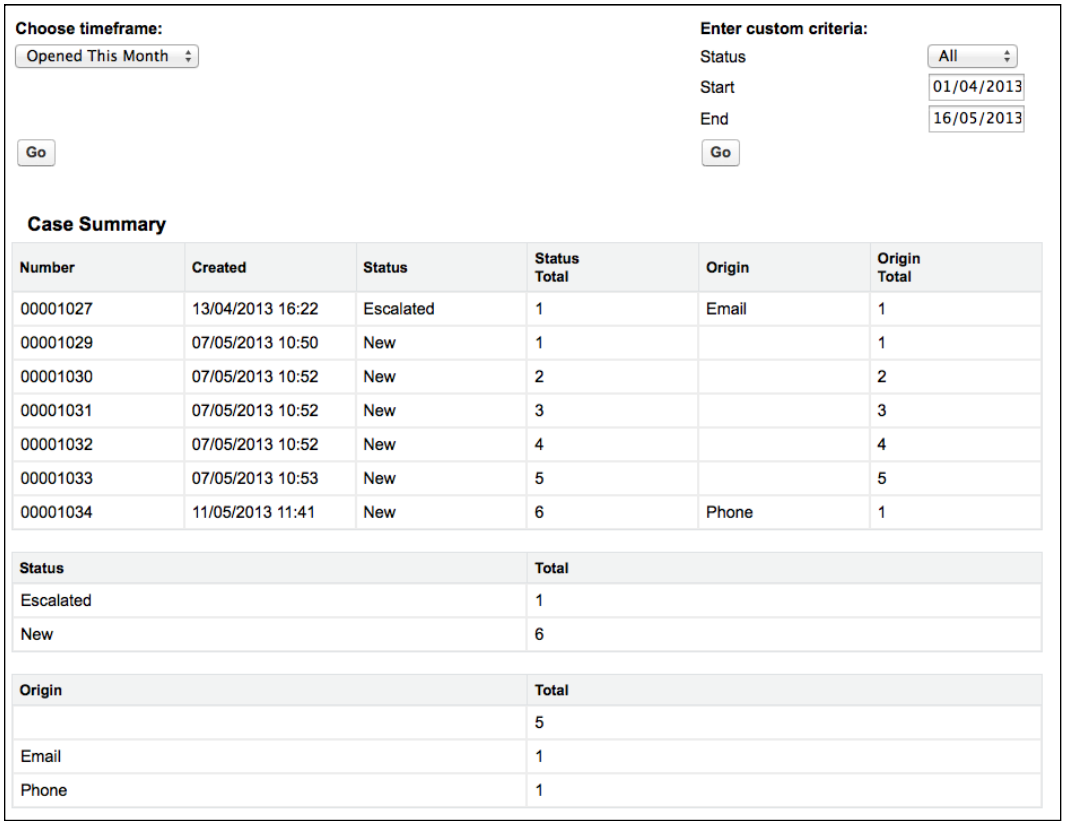
Opening the following page in your browser displays the **CasesReport** page: `https://<instance>/apex/CasesReport`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.


Choose timeframe: Opened This Month

Enter custom criteria:
Status: All
Start:
End:

Choosing either an option from the **Choose timeframe:** picklist or populating the **Enter custom criteria:** fields and clicking on the associated **Go** button retrieves the cases that match the criteria for display in a tabular format, and also renders the case totals broken down by status and origin.



Choose timeframe:
Opened This Month
Go

Enter custom criteria:
Status: All
Start: 01/04/2013
End: 16/05/2013
Go

Case Summary

Number	Created	Status	Status Total	Origin	Origin Total
00001027	13/04/2013 16:22	Escalated	1	Email	1
00001029	07/05/2013 10:50	New	1		1
00001030	07/05/2013 10:52	New	2		2
00001031	07/05/2013 10:52	New	3		3
00001032	07/05/2013 10:52	New	4		4
00001033	07/05/2013 10:53	New	5		5
00001034	11/05/2013 11:41	New	6	Phone	1

Status	Total
Escalated	1
New	6

Origin	Total
	5
Email	1
Phone	1

Each **Go** button has an associated action method to execute a query and retrieve the matching cases. The button associated with the **Choose timeframe:** picklist executes the `chooseTimeframe` method. As the value of each option in the picklist is a SOQL date literal, the chosen value can be bound directly into the query.

```
String queryStr='select id, CaseNumber, CreatedDate, Status,
                Origin from Case where CreatedDate=' + timeFrame +
                ' order by CreatedDateasc';

processCases(Database.query(queryStr));
```

More information on SOQL date literals can be found in the Salesforce help at `https://login.salesforce.com/help/doc/en/custom_dates.htm`.

The **Go** button associated with the **Enter custom criteria:** section executes the `runCustomQuery` action method; this constructs the query based on the user's inputs.

```
Date startDate=carrier1.ActivityDate;
Date endDate=carrier2.ActivityDate;
String queryStr='select id, CaseNumber, CreatedDate, Status,
                Origin from Case where CreatedDate>=:startDate ' +
                ' andCreatedDate<=:endDate ';
if (statusCriteria!='All')
{
  if ('Open'==statusCriteria)
  {
    queryStr+='and Status!=\'Closed\'';
  }
  else if ('Closed'==StatusCriteria)
  {
    queryStr+='and Status=\'Closed\'';
  }
}

queryStr+=' order by CreatedDateasc';

processCases(Database.query(queryStr));
```

The custom criteria section uses task records as carriers for the **Start Date** and **End Date** inputs. A carrier object is an sObject that is used to hold a set of fields for a page, rather than for its intended purpose. While this could be achieved using `String` properties, using a task sObject allows the `<apex:inputField />` standard component to be used, which generates appropriate HTML markup for a datepicker.

Each method executes the appropriate query and delegates the results to the `processCases` method. This method updates the total number of cases for the case status and origin, encapsulates the record in a wrapper class, and adds it to the list of records to be displayed on the page.

The totals for case origin and status are stored in maps that are keyed by the status and origin values.

```
public Map<String, Integer>statusTotals {get; set;}
```

The page uses dynamic Visualforce bindings to iterate the keys from the map and output the associated values.

```
<apex:repeat value="{!statusTotals}" var="status">
  <tr>
    <td style="width:30%">
      <apex:outputText value="{!status}" />
    </td>
    <td style="width:30%">
      <apex:outputText value="{!statusTotals[status]}" />
    </td>
  </tr>
</apex:repeat>
```

[More information on using dynamic Visualforce bindings to reference Apex maps and lists can be found at `http://www.salesforce.com/us/developer/docs/pages/Content/pages_dynamic_vf_maps_lists.htm`.]

Loading records asynchronously

In the previous recipes, all lists of records being managed by the page or related to the record being managed have been loaded synchronously; that is, the records have been retrieved by the controller and displayed when the page is initially loaded. In the event that the query retrieving the records is complex (and thus, time consuming), or where the payload for the records is large due to the volume of records or the size of each individual record, this can result in a delay before the page is loaded. A delay of this nature is invariably a negative experience for the user, often leading them to conclude that the application has failed in some way.

In this recipe, we will create a Visualforce page that loads an account record prior to rendering the page for the first time, and then loads the opportunity records associated with the account asynchronously. A spinning GIF is displayed to the user indicating that the asynchronous load is taking place.

Getting ready

This recipe makes use of a controller extension, so this will need to be present before the Visualforce page can be created.

How to do it...

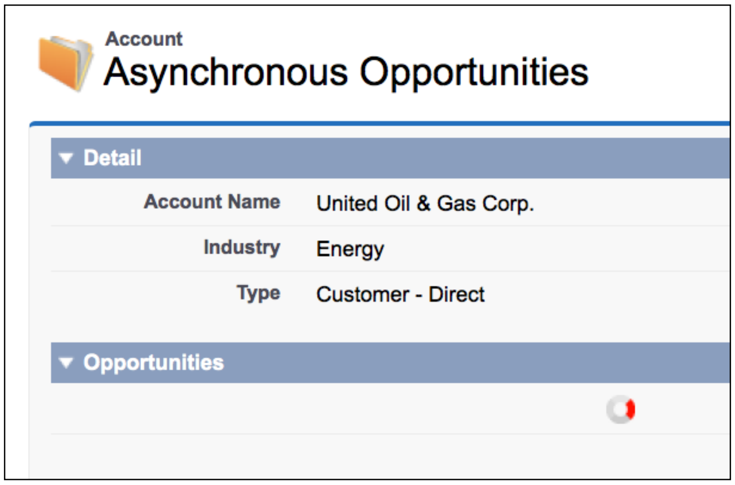
1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `AsynchLoadExt.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `AsynchLoad` in the **Label** field.
8. Accept the default **AsynchLoad** that is automatically generated for the **Name** field.
9. Paste the contents of the `AsynchLoad.pagefile` from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **AsynchLoad** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **AsynchLoad** page: `https://<instance>/apex/AsynchLoad?id=<account_id>`.

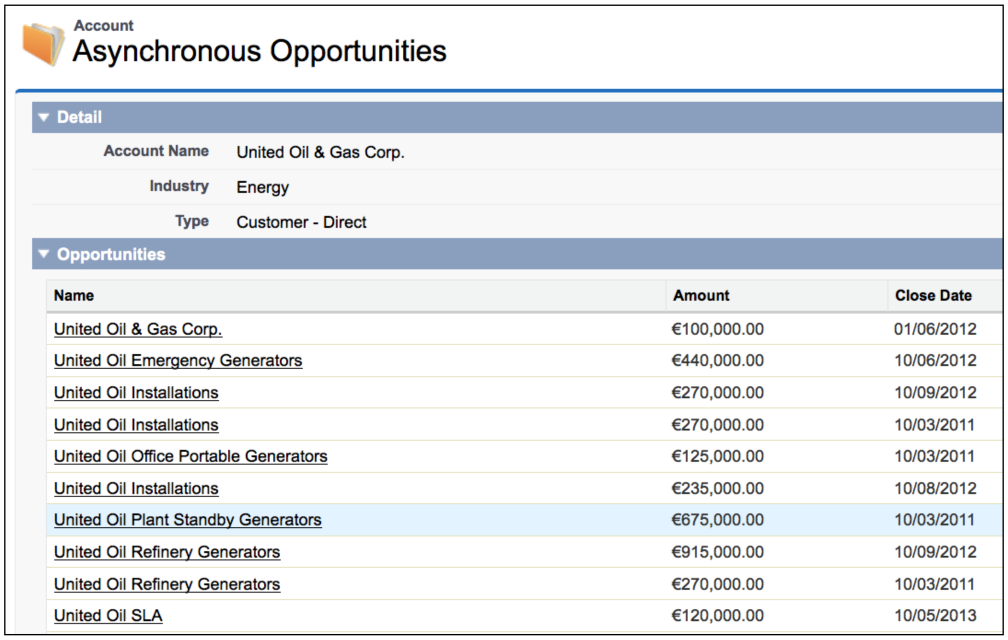
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<account_id>` is the ID of any account record in your organization.

This initially displays details of the account and the spinning GIF in the **Opportunities** section.



Account Name	United Oil & Gas Corp.
Industry	Energy
Type	Customer - Direct

Once the asynchronous load completes, the opportunity details are rendered into the **Opportunities** section.



Account Name	United Oil & Gas Corp.
Industry	Energy
Type	Customer - Direct

Name	Amount	Close Date
United Oil & Gas Corp.	€100,000.00	01/06/2012
United Oil Emergency Generators	€440,000.00	10/06/2012
United Oil Installations	€270,000.00	10/09/2012
United Oil Installations	€270,000.00	10/03/2011
United Oil Office Portable Generators	€125,000.00	10/03/2011
United Oil Installations	€235,000.00	10/08/2012
United Oil Plant Standby Generators	€675,000.00	10/03/2011
United Oil Refinery Generators	€915,000.00	10/09/2012
United Oil Refinery Generators	€270,000.00	10/03/2011
United Oil SLA	€120,000.00	10/05/2013

The controller extension defines a property, `oppsNeeded`, that indicates if the opportunities have been asynchronously loaded or not. When the page is initially loaded, an `<apex:outputPanel />` component is rendered to execute an action function to load the opportunities and set the value of the `oppsNeeded` value to `false`.

```
<apex:outputPanel rendered="{!oppsNeeded}">
  <script>
    loadOppsJS();
  </script>
</apex:outputPanel>
```

Upon completion of the action function, the opportunities section is rerendered. This section displays a spinning GIF if the `oppsNeeded` property has a value of `false`, or the list of opportunities if the value is `true`.

```
<apex:outputPanel rendered="{!oppsNeeded}">
  <apex:pageBlockSection title="Opportunities">
    <div id="spinner">
      <p align="center" style='{font-family:"Arial", Helvetica, sans-
serif; font-size:20px;}'>
        <apex:image value="/img/loading.gif"/>
      </p>
    </div>
  </apex:pageBlockSection>
</apex:outputPanel>

<apex:outputPanel rendered="{!NOT(oppsNeeded)}">
  <apex:pageBlockSection title="Opportunities" columns="1">
    <apex:pageBlockTable value="{!opps}" var="opp"
rendered="{!oppsFound}">
      <apex:column
       headerValue="{!$ObjectType.Opportunity.fields.Name.label}">
        <apex:outputLink value="/{!opp.id}">
          {!opp.Name}
        </apex:outputLink>
      </apex:column>
      ...
    </apex:pageBlockTable>
  </apex:pageBlockSection>
</apex:outputPanel>
```

6
Visualforce Charts

In this chapter, we will cover the following recipes:

- Creating a bar chart
- Creating a line chart
- Customizing a chart
- Adding multiple series
- Creating a stacked bar chart
- Adding a third axis
- Embedding a chart in a record view page
- Multiple charts per page

Introduction

Visualforce charting allows custom charts to be embedded into any Visualforce page using standard components, only server-side code is required. A key difference from the standard charting functionality available in reports and dashboards is that the data is provided by the Visualforce page controller and can be derived from any number of sObjects, regardless of whether any relationships between the sObjects exist.

Visualforce charts became *Generally Available* in the Winter '13 release of Salesforce. Prior to this, custom charts required use of a JavaScript framework, such as Dojo Charting or Google Charts.

In this chapter, we will create a number of Visualforce charts of increasing complexity, add a chart to a standard Salesforce record view page, and generate a number of charts on a single page, much like a standard Salesforce dashboard.

Creating a bar chart

Bar charts allow easy comparison of groups of data. A typical use in Salesforce is to view performance on a month-by-month basis; for example, to identify the effectiveness of a process improvement.

In this recipe, we will create a Visualforce page containing a bar chart that displays the total value of won opportunities per month for the previous 12 months. This allows a sales manager to view at a glance whether sales are increasing or decreasing, and to identify any problem months that require further analysis.

Getting ready

This recipe makes use of a custom controller, so this must be created before the Visualforce page.

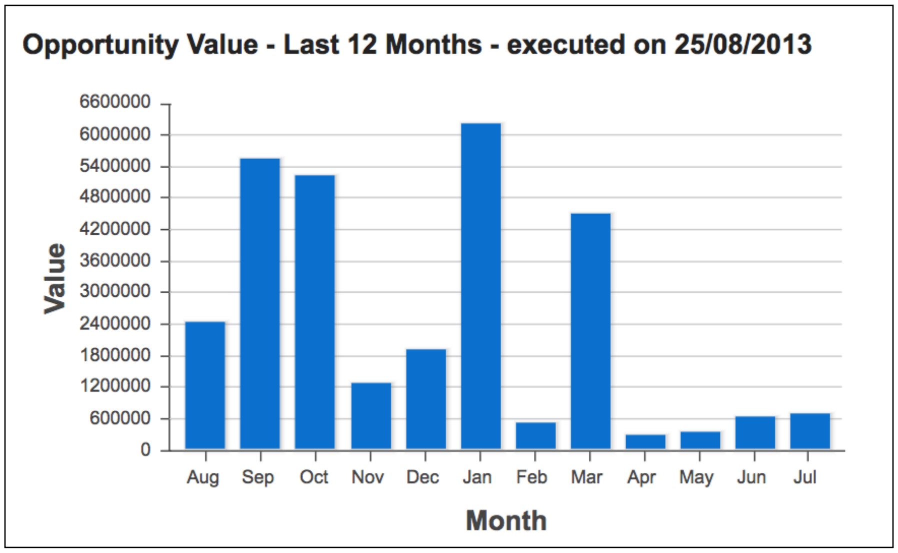
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `BarChartController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `BarChart` in the **Label** field.
8. Accept the default **BarChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `BarChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **BarChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **BarChart** page: `https://<instance>/apex/BarChart`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="300" width="550" data="{!chartData}">
```

The bar series component defines the values from the chart data that will be used to plot the x and y values.

```
<apex:barSeries orientation="vertical" axis="bottom" xField="name"
      yField="oppTotal" />
```

The x and y values *must* appear in every record of the chart data collection.

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the total opportunity value.

```
<apex:axis type="Category" position="bottom" fields="name"
title="Month" />
<apex:axis type="Numeric" position="left" fields="oppTotal"
title="Value" grid="true"/>
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  public Decimal oppTotal { get; set; }
}
```

Here, the `name` property contains the month name, while the `oppTotal` property contains the total value of opportunities closed in that month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all opportunities closed in the last year and adds the opportunity amount to the wrapper class instance for the month that the opportunity closed in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
    Time.newInstance(23, 59, 59, 999));

    ...
for (Opportunity opp : [select id, CloseDate, Amount
      from Opportunity
      where IsClosed = true
        and IsWon = true
      and CloseDate>=:startDT.date()
      and CloseDate<=:endDT.date()])
{
  Data cand=dataByMonth.get(opp.CloseDate.month()-1);
  cand.oppTotal+=opp.Amount;
}
```

In order to ensure that a bar is rendered for each month, the controller iterates the months and generates a random value for any month that has an opportunity total value of zero.

```
for (Integer idx=0; idx<12; idx++)
{
  Data cand=dataByMonth.get(idx);
  if (0.0==cand.oppTotal)
  {
    cand.oppTotal=Math.random()*750000;
  }
}
```

See also

- The *Creating a stacked bar chart* recipe in this chapter shows how to create a chart where each bar contains a breakdown of the data set.
- The *Adding multiple series* recipe in this chapter shows how to plot a bar and a line series on the same chart.

Creating a line chart

Line charts are useful to demonstrate changes in data over time. A typical use in Salesforce is to view the number of records with a particular characteristic over a period of time.

In this recipe, we will create a Visualforce page containing a line chart that displays the total number of closed cases per month for the previous 12 months.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

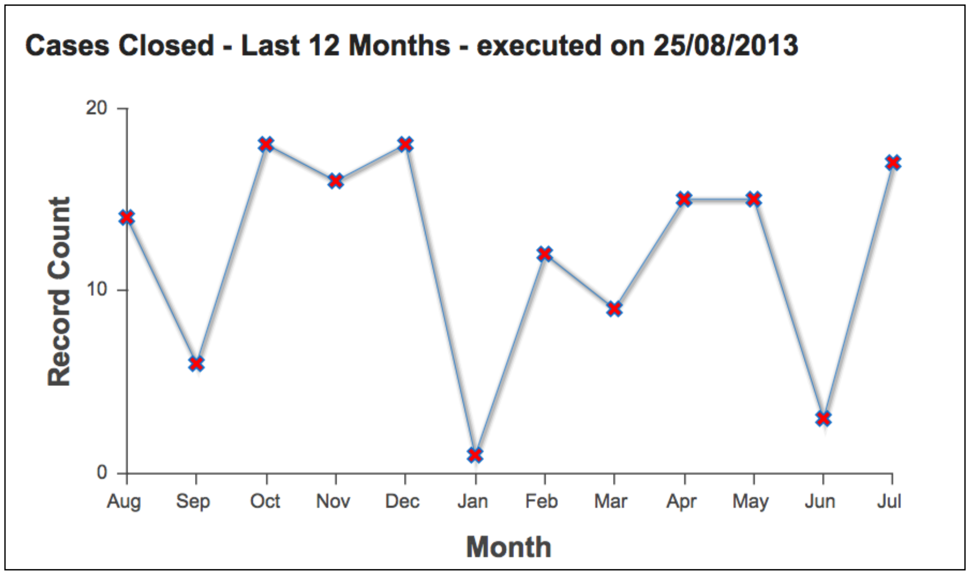
1. First, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `LineChartController.cls` Apex class from the code download into the Apex Class area.

4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `LineChart` in the **Label** field.
8. Accept the default **LineChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `LineChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **LineChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **LineChart** page: `https://<instance>/apex/LineChart`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="300" width="550" data="{!chartData}">
```

The line series component defines the values from the chart data that will be used to plot the x and y values, and how each point in the series should be decorated.

```
<apex:lineSeries axis="bottom" fill="false" xField="name"
    yField="recordCount"markerType="cross" markerSize="4"
    markerFill="#FF0000"/>
```

Unlike the majority of Visualforce components, no error will be generated at save time if `xField` or `yField` refer to properties that do not exist in the chart data collection. In this instance, no chart will be rendered and the browser will generate a JavaScript error.

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the record count value.

```
<apex:axis type="Numeric" position="left" fields="recordCount"
title="Record Count" grid="false" steps="1"/>
<apex:axis type="Category" position="bottom" fields="name"
title="Month" />
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  publicDecimalrecordCount { get; set; }
}
```

Here, the `name` property contains the month name, while the `recordCount` property contains the total number of cases closed in that month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all opportunities closed in the last year and increments the record count in the wrapper class instance for the month that the opportunity closed in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
        Time.newInstance(23, 59, 59, 999));

        ...
    for (Case cs : [select id, ClosedDate
        from Case
          where IsClosed = true
        and ClosedDate>=:startDT
        and ClosedDate<=:endDT])
    {
      Data cand=dataByMonth.get(cs.ClosedDate.date().month()-1);
      cand.recordCount++;
    }
```

In order to ensure that a point is plotted for each month, the controller iterates the months and generates a random value for any month that has a record count of zero.

```
    for (Integer idx=0; idx<12; idx++)
    {
      Data cand=dataByMonth.get(idx);
      if (0.0==cand.recordCount)
      {
        cand.recordCount=(Math.random()*20).intValue();
      }
    }
```

See also

- The *Adding multiple series* recipe in this chapter shows how to plot a bar and a line series on the same chart.

Customizing a chart

Visualforce charts are highly customizable; colors, markers, line widths, highlighting, legends, labels, and more are under the control of the developer.

In this recipe we will create a Visualforce page containing a bar chart displaying the total value of won opportunities per month for the last year.

The chart will be customized to display horizontal bars in a custom dark blue color that do not highlight when the user hovers over a bar. Finally, a legend will be displayed to show the user what the bars represent.

Getting ready

This recipe relies on the custom controller from the *Creating a bar chart* recipe in this chapter. If you have already completed that recipe, you can skip this section.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `BarChartController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

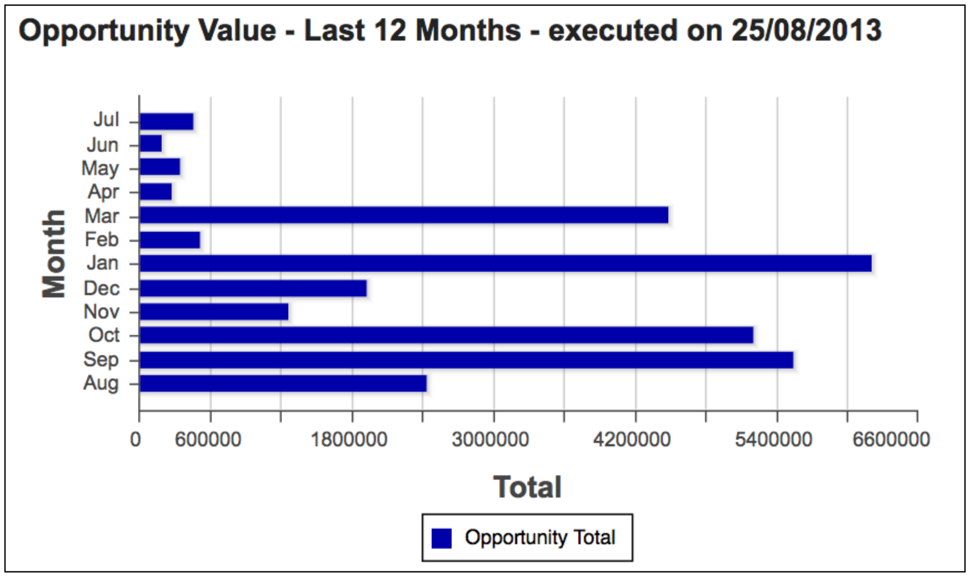
How to do it...

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CustomBarChart` in the **Label** field.
4. Accept the default **CustomBarChart** that is automatically generated for the **Name** field.
5. Paste the contents of the `CustomBarChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **CustomBarChart** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **CustomBarChart** page: `https://<instance>/apex/CustomBarChart`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="400" width="550" data="{!chartData}">
```

The bar series component defines the values from the chart data that will be used to plot the x and y values.

```
<apex:barSeries axis="bottom" xField="oppTotal" yField="name"
  colorSet="#00A" highlight="false" title="Total"
  orientation="horizontal"/>
```

The following attributes override the default functionality of the bar series:

- `colorSet`: This defines the custom color for the bars
- `highlight`: This specifies whether the bar should be highlighted when the user hovers their mouse over it
- `orientation`: This specifies whether the bars should be drawn vertically (the default) or horizontally

The chart legend is created by an `<apex:legend/>` component nested inside the chart component.

```
<apex:legend position="bottom"/>
```

The axes for the chart are defined by `<apex:axis/>` components: one for the left-hand axis displaying the month name and another for the bottom axis displaying the opportunity total value.

```
<apex:axis type="Category" position="left" fields="name"
title="Month" />
<apex:axis type="Numeric" position="bottom" fields="oppTotal"
title="Total" grid="true"/>
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  public Decimal oppTotal { get; set; }
}
```

Here, the `name` property contains the month name, while the `oppTotal` property contains the total value of opportunities closed in that month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all opportunities closed in the last year and adds the opportunity amount to the wrapper class instance for the month that the opportunity closed in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
    Time.newInstance(23, 59, 59, 999));

    ...
for (Opportunity opp : [select id, CloseDate, Amount
      from Opportunity
      where IsClosed = true
        and IsWon = true
      and CloseDate>=:startDT.date()
      and CloseDate<=:endDT.date()])
{
  Data cand=dataByMonth.get(opp.CloseDate.month()-1);
  cand.oppTotal+=opp.Amount;
}
```

Adding multiple series

In the previous recipes in this chapter, each chart contained a single series. Visualforce charts are not limited to this and can plot multiple sets of data, regardless of whether there is a relationship between the data sets.

In this recipe we will create a Visualforce page containing a chart that plots two series against the month for the last year. The first is a bar series of the number of opportunities lost in the month, while the second is a line series of the number of opportunities won in the month. This allows a sales director to see if the won/lost ratio is improving over time.

Getting ready

This recipe makes use of a custom controller, so this must be present before the Visualforce page can be created.

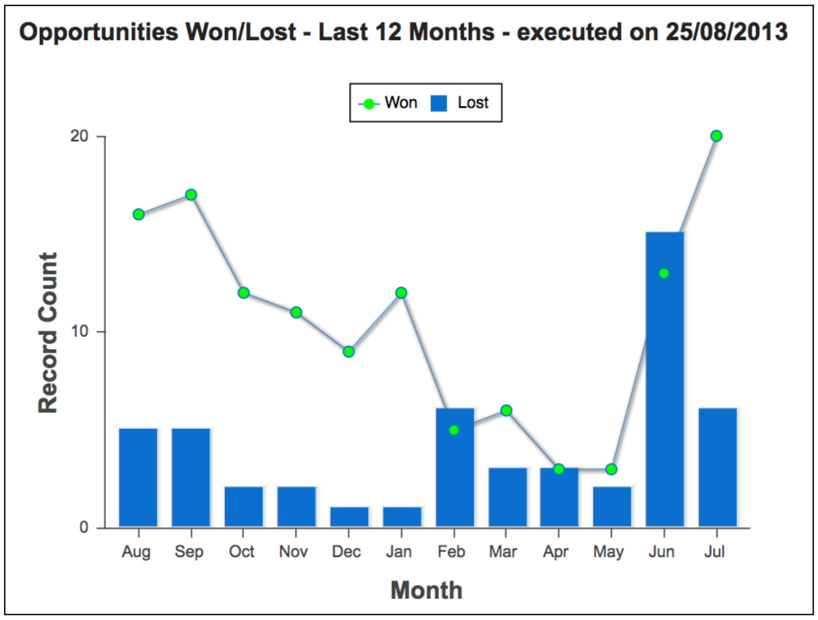
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `MultiSeriesChartController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `MultiSeriesChart` in the **Label** field.
8. Accept the default **MultiSeriesChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `MultiSeriesChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **MultiSeriesChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **MultiSeriesChart** page: `https://<instance>/apex/MultiSeriesChart`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="300" width="550" data="{!chartData}">
```

A bar series component defines the values from the chart data that will be used to plot the lost opportunity x and y values. The `title` attribute defines the title that will be used in the legend for the series.

```
<apex:barSeries orientation="vertical" axis="bottom" xField="name"
    yField="lostCount" title="Lost" />
```

A line series component defines the values from the chart data that will be used to plot the x and y values, and how each point in the series should be decorated. Once again the `title` attribute defines the title that will be used in the legend for the series.

```
<apex:lineSeries axis="bottom" fill="false" xField="name"
    yField="wonCount" markerType="circle" markerSize="4"
    markerFill="#00FF00" title="Won"/>
```

The chart legend is created by an `<apex:legend/>` component nested inside the chart component. The `position` attribute defines the location of the legend; in this recipe it will be displayed above the chart.

```
<apex:legend position="top"/>
```

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the won and lost record count values. As there are multiple series being plotted against the left-hand axis, a comma-separated list of the chart data properties is specified as the value of the `fields` attribute.

```
<apex:axis type="Numeric" position="left"
fields="wonCount,lostCount"title="Total" grid="true"
steps="1"/>
<apex:axis type="Category" position="bottom" fields="name"
title="Month" />
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  publicIntegerwonCount { get; set; }
  publicIntegerlostCount { get; set; }
}
```

Here, the `name` property contains the month name, the `wonCount` property contains the total number of opportunities won in that month, and the `lostCount` property contains the total number of opportunities lost in that month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all opportunities closed in the last year and increments the won or lost count in the wrapper class instance for the month that the opportunity closed in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
      Time.newInstance(23, 59, 59, 999));

      ...
  for (Opportunity opp : [select id, CloseDate
        from Opportunity
        where IsClosed = true
          and CloseDate>=:startDT.date()
        and CloseDate<=:endDT.date()])
  {
    Data cand=dataByMonth.get(opp.CloseDate.month()-1);
    if (opp.IsWon)
    {
      cand.wonCount++;
    }
    else
    {
      cand.lostCount++;
    }
  }
```

In order to ensure that a bar and point are plotted for each month, the controller iterates the months and generates a random value for any month that has a won or lost record count of zero.

```
  for (Integer idx=0; idx<12; idx++)
  {
    Data cand=dataByMonth.get(idx);
    if (0.0==cand.wonCount)
    {
      cand.wonCount=(Math.random()*50).intValue();
    }

    if (0.0==cand.lostCount)
    {
      cand.lostCount=(Math.random()*50).intValue();
    }
  }
```

See also

- The *Creating a bar chart* recipe in this chapter shows how to plot a data set as a series of bars.
- The *Creating a line chart* recipe in this chapter shows how to plot a data set as a line.

Creating a stacked bar chart

Stacked bar charts allow the contributing parts of data to be compared to the whole. An example of this is displaying a bar that represents the total number of opportunities that are currently open, with sections of the bar displaying the count of opportunity records that are in each stage of the sales process.

In this recipe, we will create a Visualforce page containing a stacked bar chart where each bar displays the total opportunity value that closed that month, both won and lost, for the last 12 months. Each bar is divided into two segments: the lower segment shows the total value of opportunities lost in a month, while the upper segment shows the total value won.

Getting ready

This recipe makes use of a custom controller, so this must be present before the Visualforce page can be created.

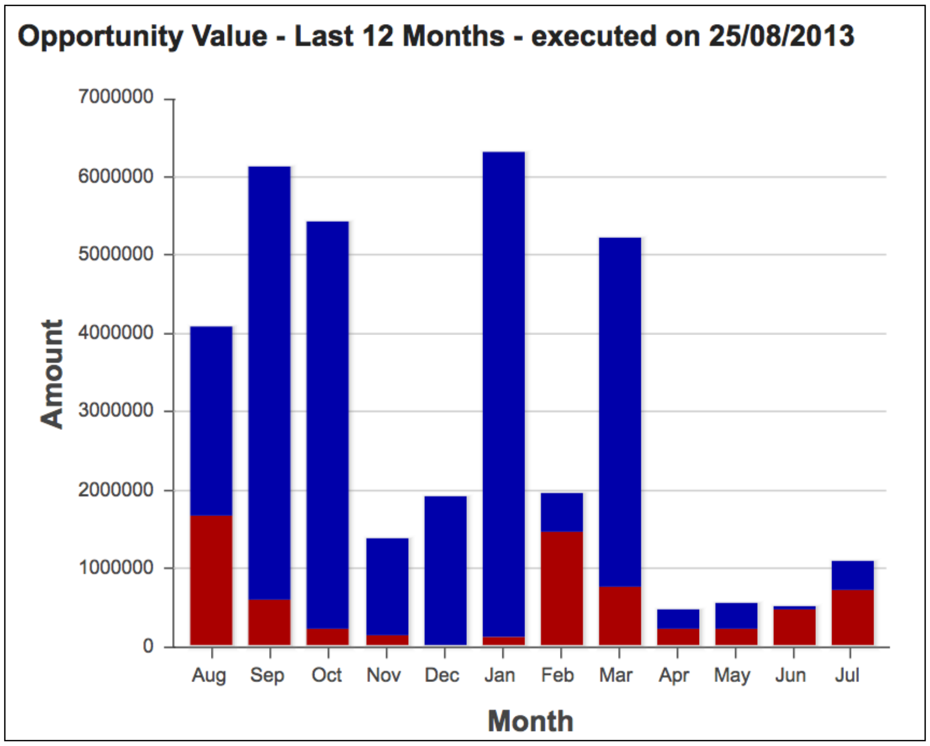
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `StackedBarChartController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter **StackedBarChart** in the **Label** field.
8. Accept the default **StackedBarChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `StackedBarChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **StackedBarChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the custom setting maintenance settings page: `https://<instance>/apex/StackedBarChart`.

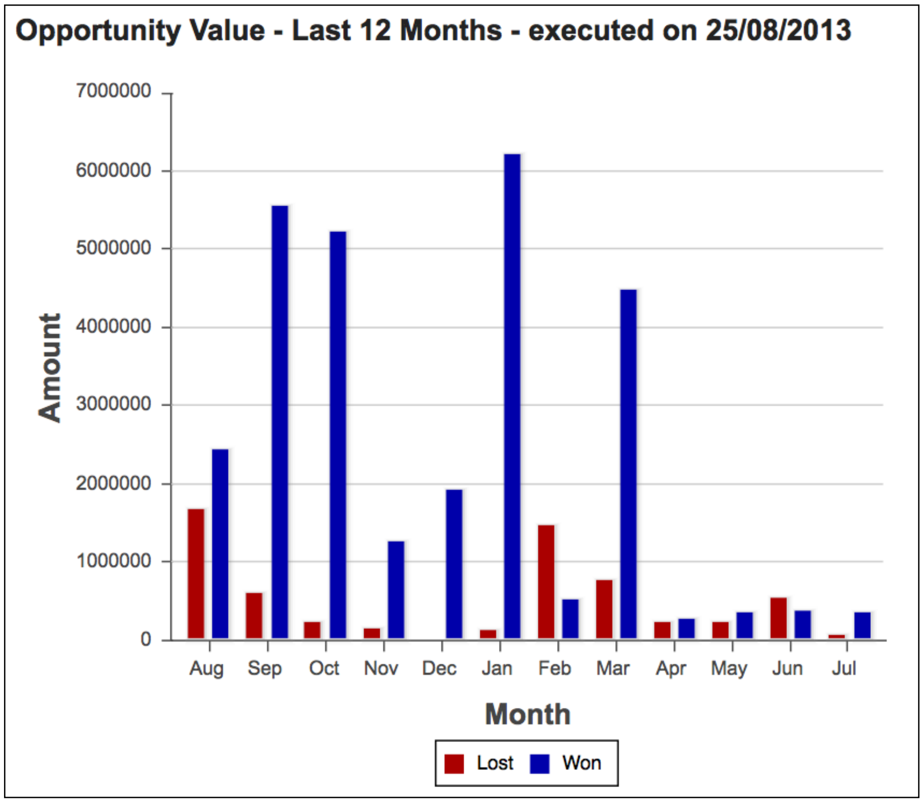
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="450" width="550" data="{!chartData}">
```

A bar series component defines the values from the chart data that will be used to plot the won/lost opportunity x and y values. Setting the `stacked` attribute to `true` specifies that the values should be stacked on top of each other in a single bar. Setting the `stacked` attribute to `false` causes the bars to be rendered side by side as shown in the following screenshot:



As this component is generating a stacked bar chart, the `yField` attribute contains comma-separated values for all segments of the bar, in this case, the lost and won opportunity totals.

```
<apex:barSeriescolorSet="#A00,#00A" orientation="vertical"
    axis="bottom" xField="name"
    yField="lostTotal,wonTotal"
    title="Lost, Won" stacked="true"/>
```

The `colorSet` and `title` attributes also contain comma-separated values for each bar segment. The `colorSet` attribute defines the color to apply to each segment, while the `title` attribute defines the text to be displayed in the legend for each segment.

> The colors in the colorSet attribute are specified as RGB (Red, Green, Blue) values, where #A00 equates to a dark red color and #00A equates to a dark blue color.

The chart legend is created by an <apex:legend/> component nested inside the chart component. The position attribute defines the location of the legend; in this recipe it will be displayed below the chart.

```
<apex:legend position="bottom"/>
```

The axes for the chart are defined by <apex:axis/> components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the won and lost total values. As a stacked bar series is being plotted against the left-hand axis, a comma-separated list of the chart data properties is specified as the value of the fields attribute.

```
<apex:axis type="Category" position="bottom" fields="name"
  title="Month" />
<apex:axis type="Numeric" position="left"
fields="wonTotal,lostTotal"title="Amount"
grid="true"/>
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  publicDecimalwonTotal { get; set; }
  publicDecimallostTotal { get; set; }
}
```

Here, the name property contains the month name, the wonTotal property contains the total value of opportunities won in that month, and the lostTotal property contains the total value of opportunities lost in that month.

The chart data collection is provided by the getChartData() controller method, which iterates all opportunities closed in the last year and applies the opportunity value to the appropriate won or lost property in the wrapper class instance for the month that the opportunity closed in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
    Time.newInstance(23, 59, 59, 999));

    ...
```

```
for (Opportunity opp : [select id, CloseDate, Amount
      from Opportunity
      where IsClosed = true
        and CloseDate>=:startDT.date()
      and CloseDate<=:endDT.date()])
{
  Data cand=dataByMonth.get(opp.CloseDate.month()-1);
  if (opp.IsWon)
  {
    cand.wonTotal+=opp.Amount;
  }
  else
  {
    cand.lostTotal+=opp.Amount;
  }
}
```

In order to ensure that a stacked bar is plotted for each month, the controller iterates the months and generates a random value for any month that has a won or lost total of zero.

```
for (Integer idx=0; idx<12; idx++)
{
  Data cand=dataByMonth.get(idx);
  if (0.0==cand.wonTotal)
  {
    cand.wonTotal=(Math.random()*750000).intValue();
  }

  if (0.0==cand.lostTotal)
  {
    cand.lostTotal=(Math.random()*750000).intValue();
  }
}
```

See also

- The *Creating a bar chart* recipe in this chapter shows how to plot a data set as a series of bars.
- The *Adding multiple series* recipe in this chapter shows how to plot a line and a bar series on the same chart.

Adding a third axis

Plotting multiple series on a single graph can be problematic if the values of the two series vary widely. For example, if the total value of won opportunities were plotted against the record count of won opportunities, the total value number would be likely to be several hundred thousand times the record count number. Plotting these on a single chart would result in the record count plot being as close to zero as to be indistinguishable from zero.

The solution to this problem is to display a third axis. The axis is scaled appropriately to the data set that is plotted against it.

In this recipe we will create a Visualforce page containing a chart that displays the total value of the won and lost opportunities per month for the last year. The won/lost information is displayed as a stacked bar chart. The chart also displays a line series chart where each point on the line series is the number of opportunities that were won/lost in that month. As the number of opportunities will be considerably lower than the total value, a third axis is added for the opportunity number values.

Getting ready

This recipe makes use of a custom controller that must be present before the Visualforce page can be created.

How to do it...

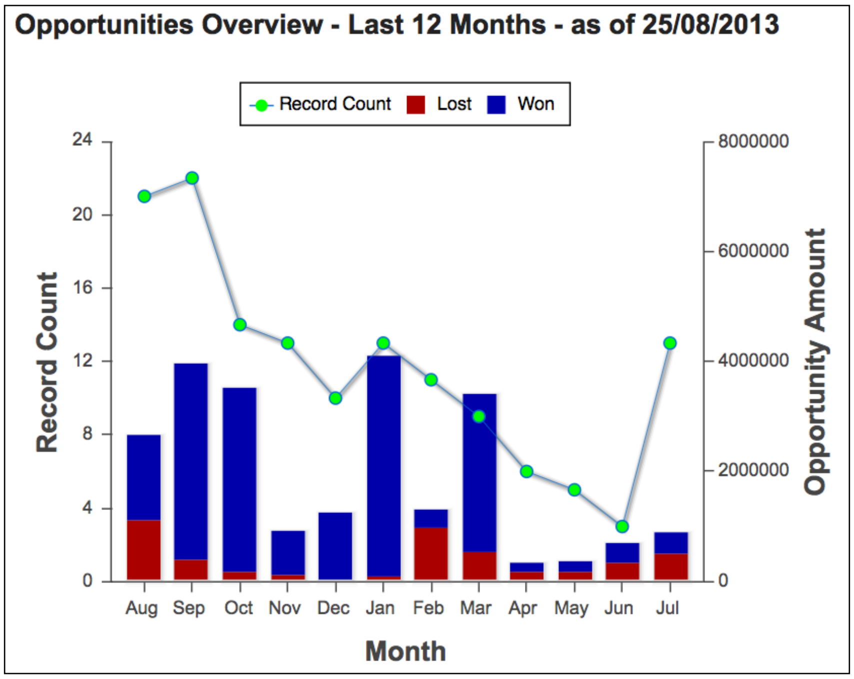
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `MultiAxisChartController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `MultiAxisChart` in the **Label** field.
8. Accept the default **MultiAxisChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `MultiAxisChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **MultiAxisChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **MultiAxisChart** page: `https://<instance>/apex/MultiAxisChart`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="450" width="550" data="{!chartData}">
```

A bar series component defines the values from the chart data that will be used to plot the won/lost opportunity x and y values. Setting the `stacked` attribute to `true` specifies that the values should be stacked on top of each other in a single bar.

As this component is generating a stacked bar chart, the `yField` attribute contains comma-separated values for all segments of the bar, in this case, the lost and won opportunity totals.

```
<apex:barSeriescolorSet="#A00,#00A" orientation="vertical"
      axis="bottom" xField="name"
      yField="lostAmount,wonAmount"
      title="Lost, Won" stacked="true"/>
```

The `colorSet` and `title` attributes also contain comma-separated values for each bar segment. The `colorSet` attribute defines the color to apply to each segment, while the `title` attribute defines the text to be displayed in the legend for each segment.

A line series component defines the values from the chart data that will be used to plot the x and y values, and how each point in the series should be decorated. Once again the `title` attribute defines the title that will be used in the legend for the series.

```
<apex:lineSeries axis="bottom" fill="false" xField="name"
      yField="recordCount"markerType="circle"
      markerSize="4" markerFill="#00FF00"
      title="Record Count"/>
```

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name, one for the left-hand axis displaying the opportunity record count, and one for the right-hand axis displaying the won/lost total values. As a stacked bar series is being plotted against the right-hand axis, a comma-separated list of the chart data properties is specified as the value of the `fields` attribute.

```
<apex:axis type="Category" position="bottom" fields="name"
title="Month" />
<apex:axis type="Numeric" position="left" fields="recordCount"
title="Record Count" grid="false" steps="5"/>
<apex:axis type="Numeric" position="right"
fields="wonAmount,lostAmount"
title="Opportunity Amount" grid="false" steps="5"/>
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  publicDecimallostAmount { get; set; }
  publicDecimalwonAmount { get; set; }
  publicIntegerrecordCount { get; set; }
}
```

Here, the `name` property contains the month name, the `wonAmount` property contains the total value of opportunities won in that month, the `lostAmount` property contains the total value of opportunities lost in that month, and the `recordCount` property contains the number of opportunities won/lost in that month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all opportunities closed in the last year, applies the opportunity value to the appropriate won or lost property in the wrapper class instance for the month that the opportunity closed in, and increments the opportunity record count:

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
    Time.newInstance(23, 59, 59, 999));

    ...
for (Opportunity opp : [select id, CloseDate, Amount
      from Opportunity
      where IsClosed = true
        and CloseDate>=:startDT.date()
      and CloseDate<=:endDT.date()])
{
  Data cand=dataByMonth.get(opp.CloseDate.month()-1);
  if (opp.IsWon)
  {
    cand.wonAmount+=opp.Amount;
  }
  else
  {
    cand.lostAmount+=opp.Amount;
  }
  cand.recordCount+=opp.Amount;
}
```

In order to ensure that a stacked bar and point are plotted for each month, the controller iterates the months and generates a random value for any month that has a won total, lost total, or record count of zero.

```
for (Integer idx=0; idx<12; idx++)
{
  Data cand=dataByMonth.get(idx);
  if (0.0==cand.wonAmount)
  {
    cand.wonAmount=(Math.random()*750000).intValue();
    }

    if (0.0==cand.lostAmount)
    {
      cand.lostAmount=(Math.random()*750000).intValue();
    }

    if (0.0==cand.recordCount)
    {
      cand.recordCount=(Math.random()*20).intValue();
    }
  }
```

See also

- The *Adding multiple series* recipe in this chapter shows how to plot a line and a bar series on the same chart.

Embedding a chart in a record view page

Visualforce charts can be generated wherever a Visualforce page can be displayed, including sidebar components, the homepage, and in standard record view pages.

In this recipe, we will create a Visualforce page containing a chart that displays a stacked bar chart containing the total number of activities carried out with a contact per month for the last year. The stacked bars contain a segment for the events and tasks that make up the activity total. This Visualforce page is embedded into the standard contact record view page to allow a sales manager to see at a glance whether a contact is being neglected or receiving more than its fair share of attention.

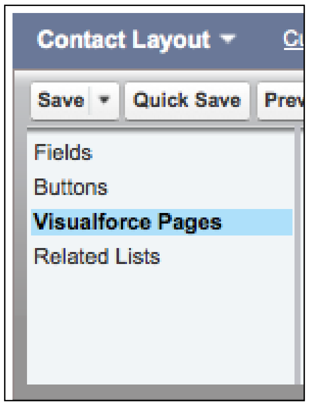
Getting ready

This recipe makes use of a controller extension that must be present before the Visualforce page can be created.

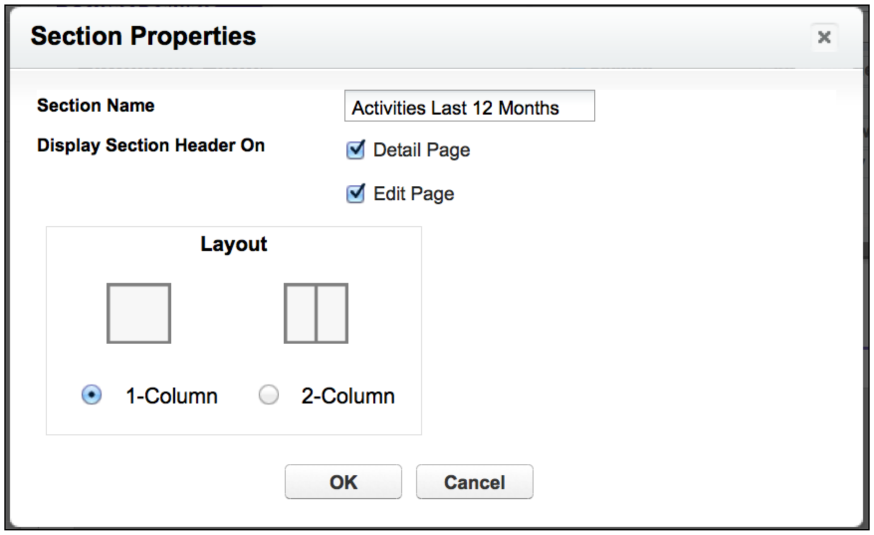
How to do it...

1. First, create the controller extension for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ContactActivitiesChartExt.cls` Apex class from the code download into the Apex Class area.

4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ContactActivitiesChart` in the **Label** field.
8. Accept the default **ContactActivitiesChart** that is automatically generated for the **Name** field.
9. Paste the contents of the `ContactActivitiesChart.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ContactActivitiesChart** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Finally, add the page to the standard contact page layout. Navigate to the contact Page Layouts page by clicking on **Your Name** | **Setup** | **Customize** | **Contact** | **Page Layouts**.
14. Locate the first page layout to add the page to and click on the **Edit** link in the **Action** column.
15. On the resulting page layout editor page, click on the **Visualforce Pages** link in the left-hand column of the palette as shown in the following screenshot:



16. Drag the **+Section** option from the right-hand side of the palette and drop this beneath the standard and custom buttons.
17. In the **Section Properties** popup, set the **Section Name** to `Activities Last 12 Months`, select the **1-column** radio button in the **Layout** section, and click on the **OK** button, as shown in the following screenshot:



Section Properties

Section Name: Activities Last 12 Months

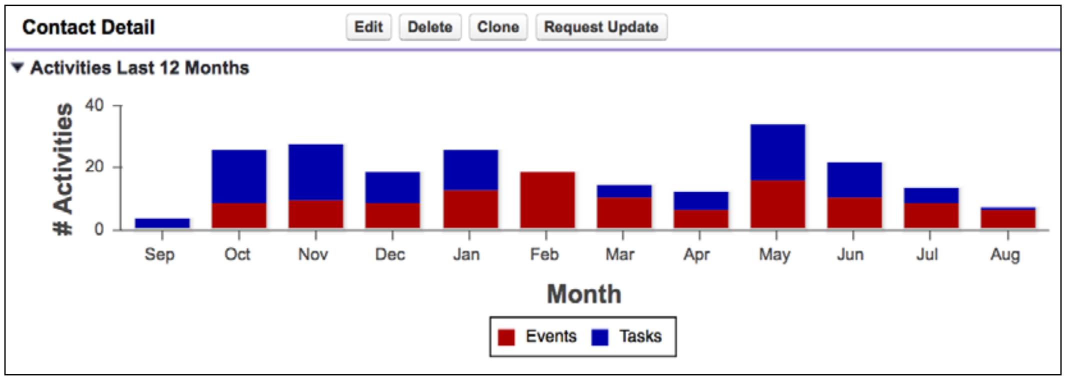
Display Section Header On: [✓ Detail Page] [✓ Edit Page]

Layout: [✓ 1-Column] 2-Column

18. Drag the **ContactActivitiesChart** page from the right-hand side of the palette and drop this beneath the **Activities Last 12 Months** section.
19. Click on the **Save** button to commit the page layout changes.
20. Repeat steps 14 to 19 to add the Visualforce page to additional page layouts as required.

How it works...

Navigating to the record view page of any contact in your Salesforce instance displays the contact detail page with the new **Activities Last 12 Months** section as shown in the following screenshot:



The Visualforce chart is generated via an `<apex:chart/>` standard component, which defines the dimensions of the chart and the collection of data that that will be plotted.

```
<apex:chart height="200" width="100%" data="{!chartData}">
```

A bar series component defines the values from the chart data that will be used to plot the won/lost opportunity x and y values. Setting the `stacked` attribute to `true` specifies that the values should be stacked on top of each other in a single bar.

As this component is generating a stacked bar chart, the `yField` attribute contains comma-separated values for all segments of the bar, in this case, the event and task record counts.

```
<apex:barSeriescolorSet="#A00,#00A" orientation="vertical"
  axis="bottom" xField="name" yField="events,tasks"
  title="Events, Tasks" stacked="true"/>
```

The `colorSet` and `title` attributes also contain comma-separated values for each bar segment. The `colorSet` attribute defines the color to apply to each segment, while the `title` attribute defines the text to be displayed in the legend for each segment.

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the activity record count. As a stacked bar series is being plotted against the right-hand axis, a comma-separated list of the chart data properties is specified as the value of the `fields` attribute.

```
<apex:axis type="Category" position="bottom" fields="name"
title="Month"/>
<apex:axis type="Numeric" position="left" fields="events,tasks"
title="# Activities" grid="false" steps="1"/>
```

The chart data is a collection of inner classes defined in the custom controller.

```
public class Data
{
  public String name { get; set; }
  public Integer events { get; set; }
  public Integer tasks { get; set; }
}
```

Here, the `name` property contains the month name, the `events` property contains the event record count for the month, and the `tasks` property contains the task record count for the month.

The chart data collection is provided by the `getChartData()` controller method, which iterates all events and tasks associated with the contact in the last year, and increments the appropriate record count property in the wrapper class instance for the month that the activity took place in.

```
DateTimestartDT=DateTime.newInstance(
    Date.today().addYears(-1).toStartOfMonth(),
    Time.newInstance(0, 0, 0, 0));
DateTimeendDT=DateTime.newInstance(Date.today(),
    Time.newInstance(23, 59, 59, 999));

    ...
for (Event ev : [select id , EndDateTime
      from Event where WhoId=:cont.id
      and EndDateTime>=:startDT
      and EndDateTime<=:endDT])
```

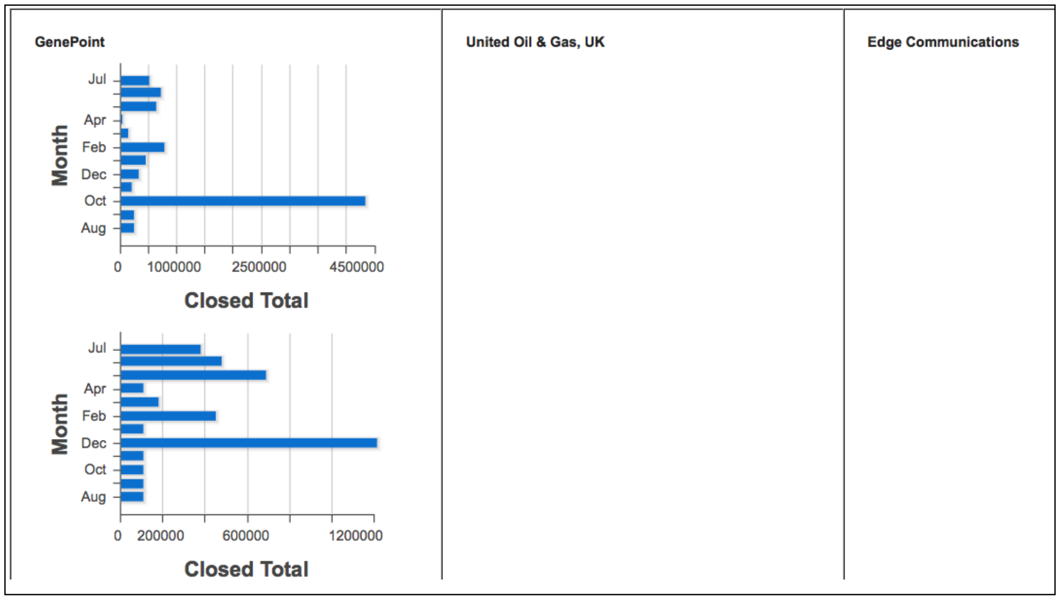
```
{
  Data cand=dataByMonth.get(ev.EndDateTime.date().month()-1);
  cand.events++;
}

for (Task ts : [select id, ActivityDate
      from Task where WhoId=:cont.id
      and ActivityDate>=:startDT.date()
      and ActivityDate<=:endDT.date()])
{
  Data cand=dataByMonth.get(ts.ActivityDate.month()-1);
  cand.tasks++;
}
```

In order to ensure that a stacked bar is plotted for each month, the controller iterates the months and generates a random value for any month that has a task or activity count of zero.

```
for (Integer idx=0; idx<12; idx++)
{
  Data cand=dataByMonth.get(idx);
  if (0==cand.events)
  {
    cand.events=(Math.random()*20).intValue();
  }
  if (0==cand.tasks)
  {
    cand.tasks=(Math.random()*20).intValue();
  }
}
```

Multiple charts per page

A common use case for Visualforce charting is producing a number of custom charts arranged into rows and columns, much like a standard dashboard. Simply adding chart components to HTML table cells results in all the charts being displayed in the top-left cell of the table, as shown in the following screenshot:



The solution to this is to use the chart `renderTo` attribute to specify the DOM component that the chart should be rendered inside.

In this recipe we will create a Visualforce page that displays a table of bar charts. Each bar chart displays the total won opportunity value per month for the last year for a specific account.

Getting ready

This recipe makes use of a custom controller, so this will need to be present before the Visualforce page can be created.

How to do it...

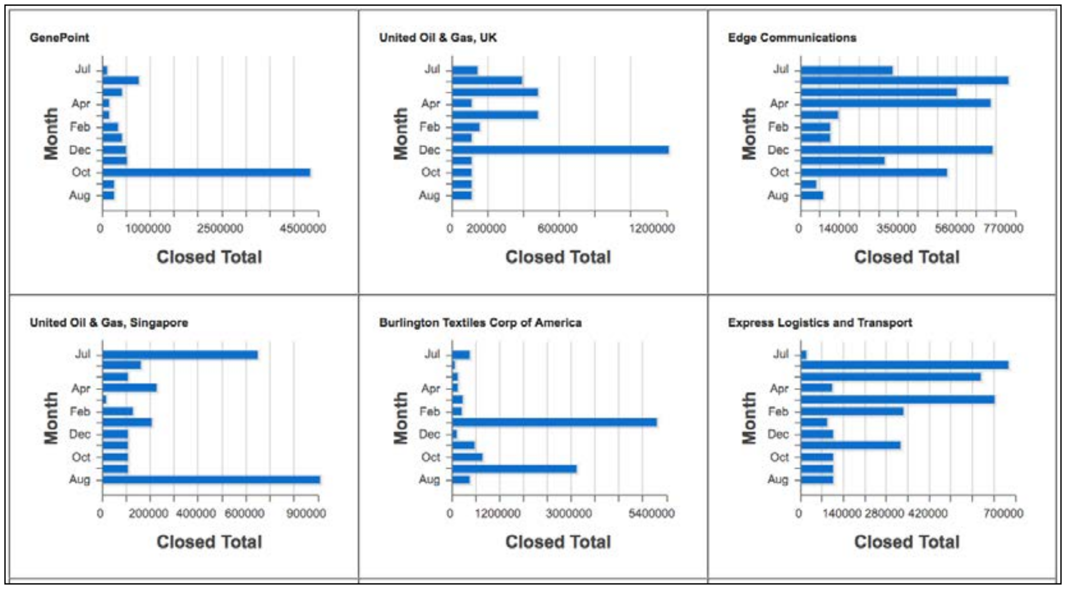
1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ChartTableController.cls` Apex class from the code download into the Apex Class area.

4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ChartTable` in the **Label** field.
8. Accept the default **ChartTable** that is automatically generated for the **Name** field.
9. Paste the contents of the `ChartTable.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ChartTable** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ChartTable** page: `https://<instance>/apex/ChartTable`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



The Visualforce charts are generated via a `<apex:chart/>` standard components in an HTML table. Each chart is nested in a `div` element that the chart is rendered to.

```
<div id="CHART{!chart.idx}">
  <apex:chart height="230" width="300" data="{!chart.months}"
      renderto="CHART{!chart.idx}">
```

A bar series component defines the values from the chart data that will be used to plot the won opportunity x and y values.

```
<apex:barSeries orientation="horizontal" axis="left"
      xField="oppTotal" yField="name" />
```

The axes for the chart are defined by `<apex:axis/>` components: one for the bottom axis displaying the month name and another for the left-hand axis displaying the won opportunity value.

```
<apex:axis type="Category" position="left" fields="name"
      title="Month" />
<apex:axis type="Numeric" position="bottom" fields="oppTotal"
      title="Closed Total" grid="true"/>
```

The data is a collection of inner classes defined in the custom controller.

```
public class MonthData
{
  public String name { get; set; }
  public Decimal oppTotal { get; set; }
}
```

A second inner class associates a collection of the `MonthData` records with a unique index and the name of the account.

```
public class ChartData
{
  public Integer idx {get; set;}
  public String name {get; set;}
  public List<MonthData>months {get; set;}
}
```

A third and final inner class encapsulates a number of the `ChartData` instances in a collection. This class is used to generate a single row in the chart table.

```
public class Row
{
  public List<ChartData> charts {get; set;}

  public Row()
```

```
      {
        charts=new List<ChartData>();
      }
    }
```

The chart data collection is provided by the `getRows()` controller method, which queries nine accounts and generates three rows of chart data, each containing chart data for three accounts.

```
    for (Account acc : [select id, name from Account order by
                       CreatedDate limit 9])
    {
      if (0==Math.mod(idx,3))
      {
        row=new Row();
        result.add(row);
      }
      row.charts.add(getAccountChartData(idx++, acc));
    }
```

The `getAccountChartData()` method iterates the opportunities won for the account over the last year, and adds the opportunity amount to the wrapper class instance for the month that the opportunity was closed in.

```
    DateTimestartDT=DateTime.newInstance(
        Date.today().addYears(-1).toStartOfMonth(),
        Time.newInstance(0, 0, 0, 0));
    DateTimeendDT=DateTime.newInstance(Date.today(),
        Time.newInstance(23, 59, 59, 999));

        ...
    for (Opportunity opp : [select id, CloseDate, Amount
            from Opportunity
            where AccountId=:acc.Id
            and IsClosed = true
            and IsWon = true
            and CloseDate>=:startDT.date()
            and CloseDate<=:endDT.date()])
    {
      MonthDatacand=dataByMonth.get(opp.CloseDate.month()-1);
      cand.oppTotal+=opp.Amount;
    }
```

In order to ensure that a bar is plotted for each month per account, the controller iterates the months and generates a random value for any month that has an opportunity total of zero.

```
for (Integer idx=0; idx<12; idx++)
{
  MonthDatacand=dataByMonth.get(idx);
  if (0.0==cand.oppTotal)
  {
    cand.oppTotal=(Math.random()*750000).intValue();
  }
}
```

7
JavaScript

In this chapter, we will cover the following recipes:

- Using action functions
- Avoiding race conditions
- The confirmation dialog
- Pressing Enter to submit
- Tooltips
- The character counter
- The onload handler
- Collapsible list elements
- The scrolling news ticker
- Carousel messages
- Hiding buttons on submit
- Client-side validation
- Trapping navigation away

Introduction

JavaScript is used on a huge number of websites to add visual effects, validation, server interaction, and many more features. As JavaScript executes at the client side, it removes the latency involved with a round trip to the web server, resulting in more responsive applications and an improved user experience. JavaScript can also provide functionality that is not possible using HTML and server-side processing, for example, handling individual key clicks or mouse movements.

Visualforce has built-in capability to allow JavaScript interaction with the page controller. For example, the `<apex:actionSupport />` component allows controller action methods to be called in response to JavaScript events, while the `<apex:actionFunction />` component generates a JavaScript function that encapsulates a controller action method. Further, many components provide the `on<event>` attributes, such as `onclick` and `onchange`, to allow custom JavaScript to be invoked in response to user actions.

JavaScript is not without its downsides; however, browser compatibility is a common problem. Using a framework such as **jQuery** (`http://jquery.com/`), **Prototype** (`http://prototypejs.org/`), or **Dojo** (`http://dojotoolkit.org/`) simplifies the task of creating cross-browser compatible JavaScript, as well as providing a powerful set of utilities to traverse and manipulate the **Document Object Model** (**DOM**).

In this chapter, we will use JavaScript to provide a variety of client-side enhancements, from ensuring that a user does not lose their work or commit it too early through to providing a visual indicator of the number of remaining characters that an input can accommodate. We will also use JavaScript to create dynamic content, such as scrolling news and carousel messages.

A number of the recipes in this chapter include JavaScript and CSS files from one or more Content Delivery Networks (CDNs). This does introduce a dependency on the site hosting the CDN and in the event that this site was unavailable or access blocked, the recipe functionality would cease to work.

Using action functions

An **action function** allows an action method from a controller to be executed from JavaScript. The standard Visualforce `<apex:actionFunction />` component generates a named function that can be called from any JavaScript code.

In this recipe, we will create a Visualforce page that displays a list of cases and a countdown timer implemented in JavaScript. Once the timer expires, an action method from the page's controller is executed, which redirects the user's browser to the standard case tab.

Getting ready

This recipe makes use of a custom controller, so this must be created before the Visualforce page.

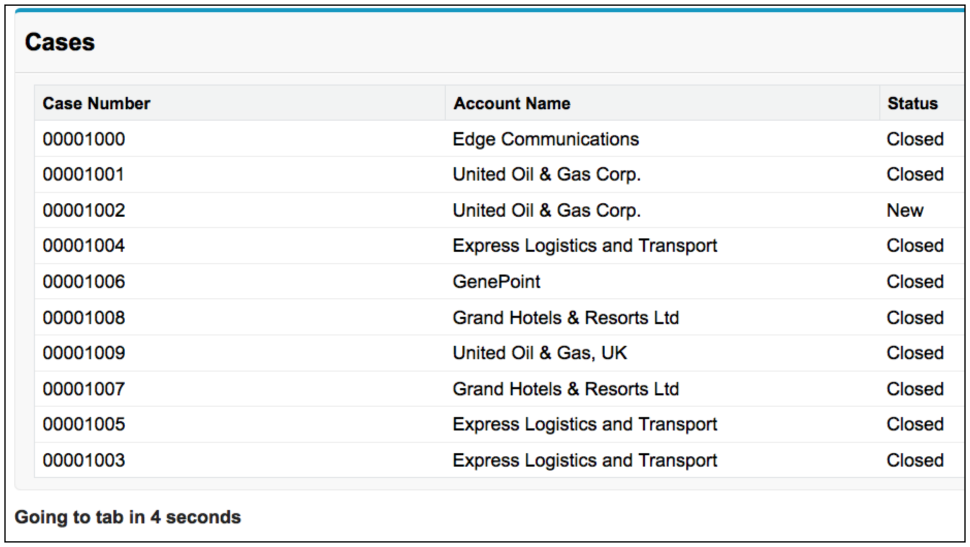
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `ActionFunctionController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionFunction` in the **Label** field.
8. Accept the default **ActionFunction** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionFunction.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ActionFunction** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionFunction** page: `https://<instance>/apex/ActionFunction`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Cases

Case Number	Account Name	Status
00001000	Edge Communications	Closed
00001001	United Oil & Gas Corp.	Closed
00001002	United Oil & Gas Corp.	New
00001004	Express Logistics and Transport	Closed
00001006	GenePoint	Closed
00001008	Grand Hotels & Resorts Ltd	Closed
00001009	United Oil & Gas, UK	Closed
00001007	Grand Hotels & Resorts Ltd	Closed
00001005	Express Logistics and Transport	Closed
00001003	Express Logistics and Transport	Closed

Going to tab in 4 seconds

The countdown timer at the bottom-left is updated by a JavaScript function that executes every second.

```
countDownObj.count = function(i)
{
  countDownObj.innerHTML = 'Going to tab in ' + i + ' seconds';
  if (i == 0) {
    fn();
    return;
  }
  setTimeout(function()
     {
        countDownObj.count(i - 1);
      },
      pause);
}
```

The controller action method is exposed as a JavaScript function via an action function.

```
<apex:form >
  <apex:actionFunction name="goCasesTabJS"
                       action="{!goCasesTab}" />
</apex:form>
```

Note that as the action function executes an action method, a form submission takes place. For this reason, the `<apex:actionFunction />` component must always be nested inside the `<apex:form />` tags.

The action method invoked when the `goCasesTabJS` JavaScript function is executed simply returns the page reference for the case tab.

```
public PageReference goCasesTab()
{
  PageReference result=new PageReference('/500/o');
  return result;
}
```

Avoiding race conditions

An action function provides a way to submit a form programmatically via a JavaScript function call. When an action function is executed from a JavaScript event handler, the default browser behavior continues once the event handler has completed. If the event handler is attached to a Visualforce component that submits the form, an `onclick` handler for an `<apex:commandLink />` or `<apex:commandButton />` component, for example, the default browser behavior is to continue with the form submission. This results in a race condition as to which form submission request will be processed first by the server and will often produce unexpected results.

In this recipe, we will create a Visualforce page to execute a search for accounts matching a user-entered string of characters. When the user clicks on the button to start the search, a JavaScript function is invoked that checks the number of characters entered. If two or more characters have been entered, the search is executed via an action function. If fewer than two characters have been entered, any existing results are cleared and the search is not executed. In either case, the default form submission from the button is stopped.

Getting ready

This recipe makes use of a custom controller, so this will need to be created before the Visualforce page.

How to do it...

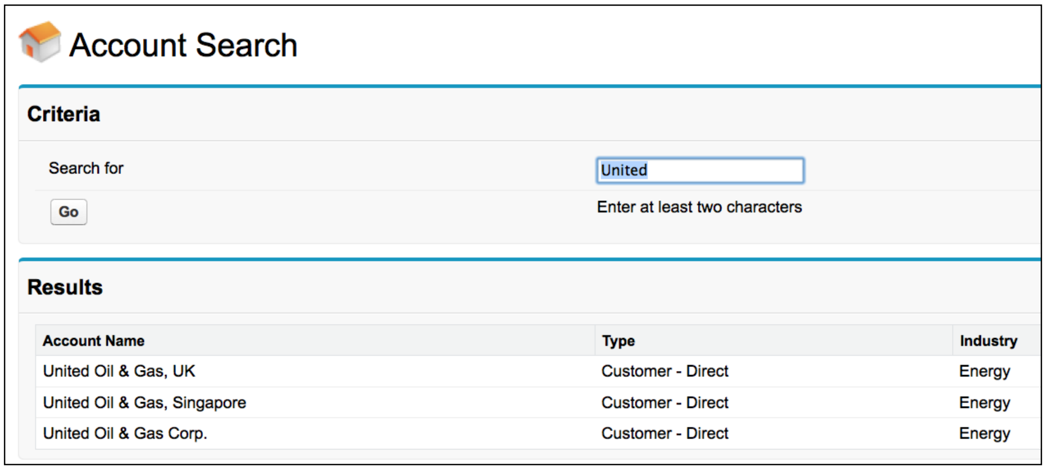
1. First, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.

3. Paste the contents of the `ActionFunctionSearchController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ActionFunctionSearch` in the **Label** field.
8. Accept the default **ActionFunctionSearch** that is automatically generated for the **Name** field.
9. Paste the contents of the `ActionFunctionSearch.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ActionFunctionSearch** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ActionFunctionSearch** page: `https://<instance>/apex/ActionFunctionSearch`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.


Account Search

Criteria

Search for: United

Go

Enter at least two characters

Results

Account Name	Type	Industry
United Oil & Gas, UK	Customer - Direct	Energy
United Oil & Gas, Singapore	Customer - Direct	Energy
United Oil & Gas Corp.	Customer - Direct	Energy

The **Go** button defines an `onclick` handler that executes a JavaScript function to determine the number of characters entered and execute a search or clear the results.

```
function runSearch()
{
  // don't run the search unless there are enough characters
  var str = document.getElementById(
    '{!$Component.frm.crit_pb.crit_pbs.crit_str}').value;
  if (str.length>=2)
  {
    doSearchJS();
  }
  else
  {
    alert('Please enter at least two characters');
    clearResultsJS();
  }
}
```

Visualforce will automatically generate an ID for each HTML element based on the `id` attribute of the element and the `id` attribute of each ancestor element, which can make identification of an individual element challenging. To assist this, Visualforce provides a `$Component` global merge variable that uses a dot notation based on the component hierarchy to identify an element.

For more information visit `http://www.salesforce.com/us/developer/docs/pages/Content/pages_best_practices_accessing_id.htm`.

The `onclick` handler returns `false` once the function has completed; this instructs the browser to stop handling the click event at that point, rather than continuing with the default behavior of submitting the form.

```
<apex:commandButton value="Go" onclick="runSearch();
    return false;" />
```

See also

- For more information on action functions, refer to the *Using action functions* recipe.

The confirmation dialog

A feature missing from the standard Salesforce record edit functionality is the ability for the user to confirm that they wish to execute an action. If a user inadvertently clicks on the **Cancel** button, their work will be discarded.

In this recipe we will create a Visualforce page that allows a user to create an account record. If the user clicks on a button to save the record or cancel the creation, they will be requested to confirm that they wish to continue with the action.

Getting ready

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

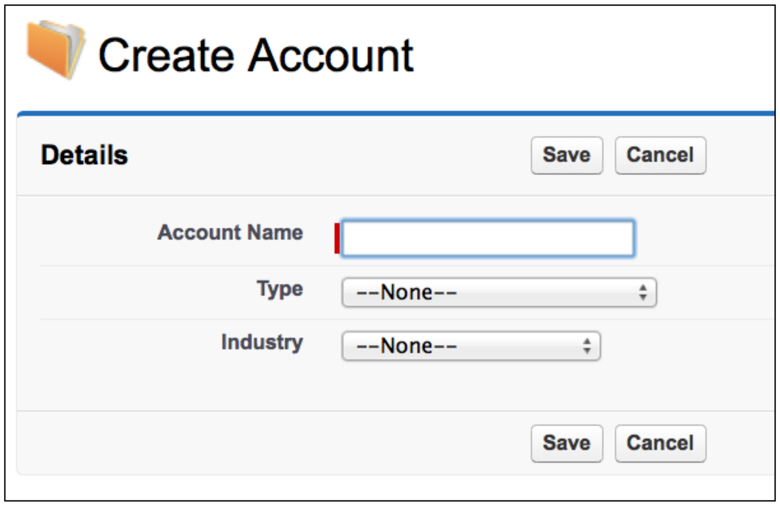
How to do it...

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `Confirmation` in the **Label** field.
4. Accept the default **Confirmation** that is automatically generated for the **Name** field.
5. Paste the contents of the `Confirmation.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **Confirmation** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

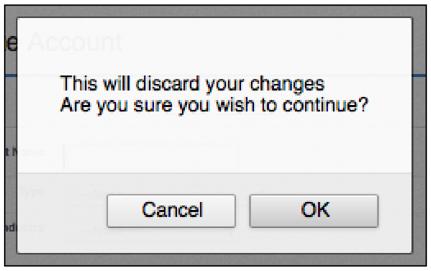
How it works...

Opening the following URL in your browser displays the **Confirmation** page: `https://<instance>/apex/Confirmation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



If the user clicks on the **Save** or **Cancel** button, they are asked to confirm the action.


This will discard your changes
Are you sure you wish to continue?

The **Save** and **Cancel** buttons each define an `onclick` handler that executes a JavaScript function. This function opens an appropriate confirmation dialog to ask the user to confirm the action.

```
function confirmCancel()
{
  return confirm("This will discard your changes\nAre you sure you
wish to continue?");
}
```

The `onclick` handlers return the result of the JavaScript function, ensuring that if the user chooses not to proceed, the default browser behavior of continuing with the form submission will not take place.

```
<apex:commandButton value="Cancel" action="{!cancel}"
    onclick="return confirmCancel();" immediate="true" />
```

See also

- The *Trapping navigation away* recipe in this chapter shows how to ask the user to confirm they wish to navigate away from a page, even if they haven't clicked a button or link on the page.
- The *Pressing Enter to submit* recipe in this chapter intercepts the pressing of an *Enter* key in a form and asks the user to confirm they wish to save their changes.

Pressing Enter to submit

When the *Enter* key is pressed and a single-line HTML form element has focus, modern browsers will submit the form via the first submit button. If the user has pressed the *Enter* key expecting to move on to a new line and remain in the input element, this can lead to the submission of a partially filled in form, resulting in a low quality record being created.

In this recipe we will create a Visualforce page that allows a user to create an opportunity. If the user presses the *Enter* key while filling in any of the opportunity fields, they will be asked to confirm that they wish to submit the form.

Getting ready

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

How to do it...

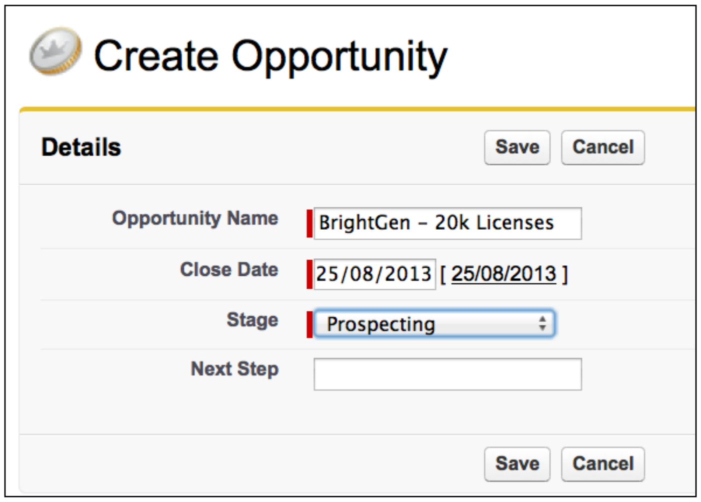
1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `PressEnter` in the **Label** field.
4. Accept the default **PressEnter** that is automatically generated for the **Name** field.
5. Paste the contents of the `PressEnter.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

7. Locate the entry for the **PressEnter** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

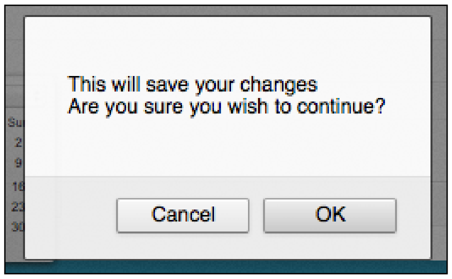
Opening the following URL in your browser displays the **PressEnter** page: `https://<instance>/apex/PressEnter`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Opportunity Name: BrightGen – 20k Licenses
Close Date: 25/08/2013 [25/08/2013]
Stage: Prospecting
Next Step:

If the user presses the *Enter* key while filling in one of the fields, they will be asked to confirm that they wish to submit the form.


This will save your changes
Are you sure you wish to continue?

Each field in the form defines an `onkeypress` event that executes a JavaScript function.

```
<apex:inputField value="{!Opportunity.Name}"
      onkeypress="return keypress(event);" />
```

This function inspects the key pressed and if it is the *Enter* key, opens a confirmation dialog to ask the user to confirm the form submission. If the user confirms that they wish to continue, the **Save** button is located and programmatically clicked.

```
var result=true;
if (keyCode == 13)
{
  var ele=document.getElementById(
    '{!$Component.frm.pb.pb_btns.savebtn}');
  if (confirm("This will save your changes\nAre you sure you wish to
continue?"))
  {
    ele.click();
  }
  result=false;
}

return result;
```

The result of the function indicates whether the default browser behavior should continue; if the key pressed is not the *Enter* key, the result is `true` and the default behavior of adding the character to the field continues. If the key pressed is *Enter*, the result is `false` which stops the default form submission.

The key pressed is identified based on its code. For a list of JavaScript key code, visit `http://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes`.

See also

- The *Trapping navigation away* recipe in this chapter shows how to ask the user to confirm they wish to navigate away from a page.
- The *The confirmation dialog* recipe in this chapter asks the user to confirm that they wish to execute the action associated with the button they have just clicked.

Tooltips

Tooltips allow additional information to be provided when an HTML element is hovered over. The standard `title` attribute available for any HTML element provides a basic tooltip, but the output of this attribute cannot be styled or use custom transitions to appear in interesting ways.

In this recipe we will create a Visualforce page containing a list of opportunities. Each column heading in the list provides an explanatory tooltip when the header text is hovered over. The tooltip is styled and slides down to reveal itself.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework and jQuery User Interface (`http://jqueryui.com/`) library to produce, style, and transition the tooltip. The JavaScript and CSS files are included from the Google Hosted Libraries content delivery network rather than being uploaded as Salesforce static resources, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

This recipe makes use of a standard controller, so this must be created before the Visualforce page.

How to do it...

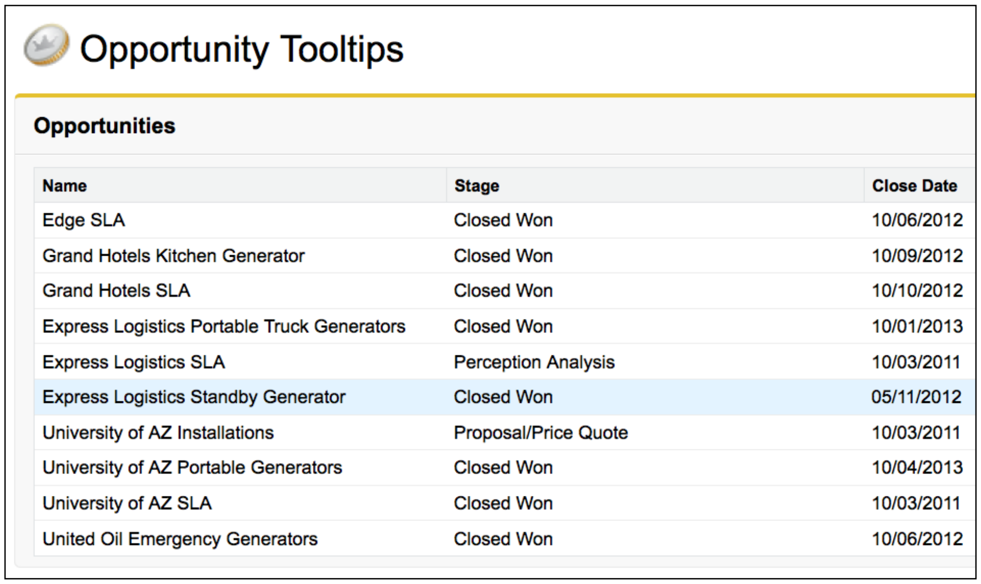
1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `TooltipsController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `ToolTips` in the **Label** field.
8. Accept the default **ToolTips** that is automatically generated for the **Name** field.
9. Paste the contents of the `ToolTips.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **ToolTips** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ToolTips** page: `https://<instance>/apex/ToolTips`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.

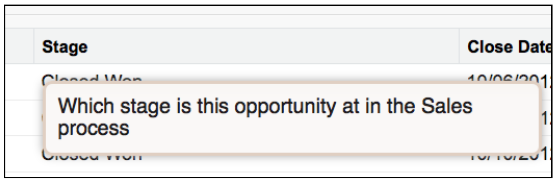


Opportunity Tooltips

Opportunities

Name	Stage	Close Date
Edge SLA	Closed Won	10/06/2012
Grand Hotels Kitchen Generator	Closed Won	10/09/2012
Grand Hotels SLA	Closed Won	10/10/2012
Express Logistics Portable Truck Generators	Closed Won	10/01/2013
Express Logistics SLA	Perception Analysis	10/03/2011
Express Logistics Standby Generator	Closed Won	05/11/2012
University of AZ Installations	Proposal/Price Quote	10/03/2011
University of AZ Portable Generators	Closed Won	10/04/2013
University of AZ SLA	Closed Won	10/03/2011
United Oil Emergency Generators	Closed Won	10/06/2012

Hovering over any of the column headings reveals a tooltip explaining the purpose of the field contained in the column.



The `header` facet of each column is encapsulated in a `span` element with the `tooltip` style class.

```
<apex:facet name="header">
  <span class="tooltip"
  title="Which stage is this opportunity at in the Sales
process">Stage</span>
</apex:facet>
```

When the page has finished loading, the tooltips are activated via a jQuery selector on the `tooltip` style class.

```
$(document).ready(function(){
                $(".tooltip").tooltip({
                       show: {
                       effect: "slideDown",
                       delay: 250
                       }
           }
           );
       });
```

The `tooltip` function turns the selected elements into jQuery UI tooltips and the `show` options define how the tooltip should reveal itself, in this case, by sliding down 250 milliseconds after the user hovers over the element.

See also

- The *Collapsible list elements* recipe in this chapter shows how jQuery can be used to collapse and expand items in a list.

The character counter

A number of field types allow a set number of characters to be entered, for example, text area and long text area. A useful addition to the user interface for these fields is a mechanism to inform the user how many more characters they may enter.

In this recipe we will create a Visualforce page that allows a user to create a case. The subject standard field can contain a maximum of 255 characters. A counter of the number of characters remaining is displayed beneath the **Subject** input field. The character counter is updated when the user types a character, or if they cut or paste information from or to the field.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework to handle the client-side events. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to a new version simply by changing the URL of the included file.

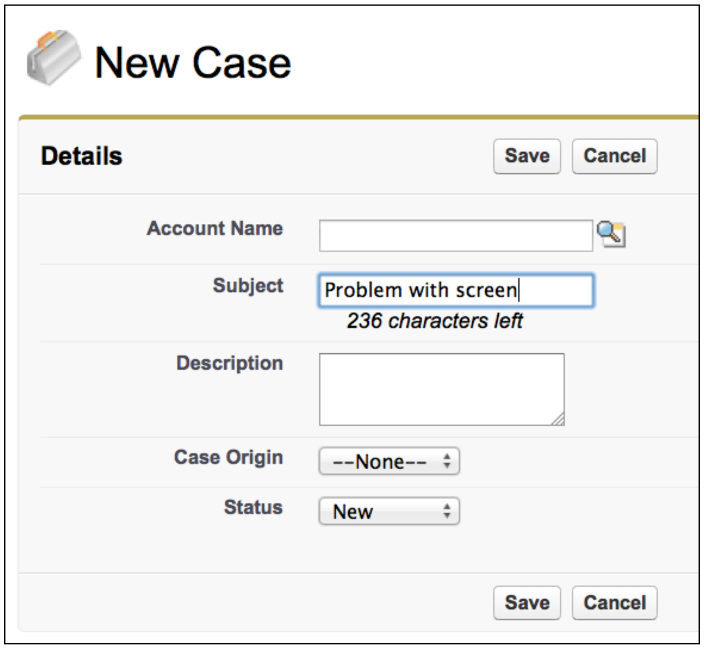
How to do it...

1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CharacterCounter` in the **Label** field.
4. Accept the default **CharacterCounter** that is automatically generated for the **Name** field.
5. Paste the contents of the `CharacterCounter.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **CharacterCounter** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **CharacterCounter** page: `https://<instance>/apex/CharacterCounter`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Account Name	
Subject	Problem with screen
	236 characters left
Description	
Case Origin	--None--
Status	New

A JavaScript function limits the number of characters entered to 255 and updates the content of the `div` element with the ID of `avail` to show the number of characters remaining:.

```
function availChars(ele)
{
  if($(ele).val().length > 255)
    $(ele).val($(ele).val().substr(0, 255));
  var diff=255-$(ele).val().length;
  $('#avail').html(diff + ' characters left');
}
```

When the page has finished loading, the function is bound to a number of events for the **Subject** input field, including the `keyup` event.

```
$(document).ready(function(){
        $('[id$=subject]').keyup( function(event) {
            availChars(event.target);

        $('[id$=subject]').bind('paste', function(event) {
             setTimeout(function() {
                     availChars(event.target);
                     }, 10);
         });
       });
```

Note that the `paste` event handler introduces a delay of 10 milliseconds before executing the `availChars` function; this allows the `paste` event to complete and update the value of the input field.

Note also that to identify the `subject` input element, we used a jQuery selector to locate the element whose ID ends with **subject**. When using a JavaScript framework that provides selectors, this is an alternative approach to the `$Component` global merge field described in the *Avoiding race conditions* recipe in this chapter.

The onload handler

An `onload` handler allows JavaScript code to be executed when an HTML page has completed loading. While adding an `onload` handler to a Visualforce page, care must be taken not to interfere with any default `onload` handler added by the platform, to give focus to the first input field in the page, for example.

In this recipe we will create a Visualforce page that allows a user to create an opportunity. An `onload` handler in the page executes a JavaScript function to set the default value for the opportunity `amount` field to `100000`. If the platform has specified an `onload` handler function, this is executed before the `amount` value is set.

How to do it...

This recipe makes use of a standard controller, so we only need to create the Visualforce page.

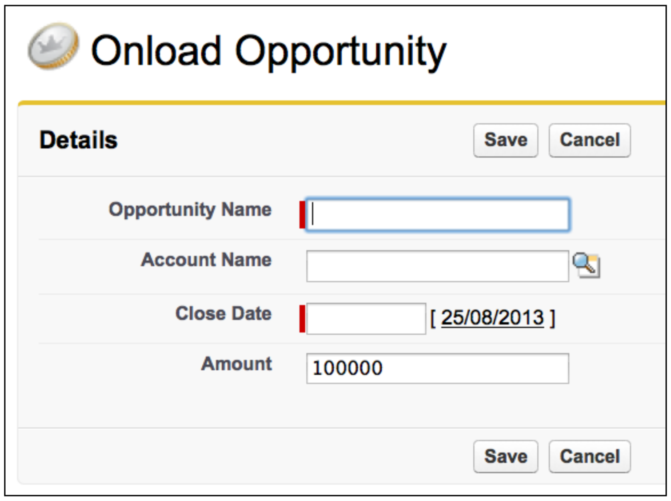
1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `Onload` in the **Label** field.

4. Accept the default **Onload** that is automatically generated for the **Name** field.
5. Paste the contents of the `Onload.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **Onload** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Onload** page: `https://<instance>/apex/Onload`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Opportunity Name:
Account Name:
Close Date: [25/08/2013]
Amount: 100000

The opportunity **Amount** value is defaulted to **100000** by the JavaScript function executed by the `onload` handler.

```
function()
{
  document.getElementById(
    '{!$Component.frm.pb.pbs.amount}').value=
                                          '100000';
}
```

The `onload` handler is added by a JavaScript function that checks if there is an existing handler. If there is, a new function is created to execute the existing handler, followed by the new one. If there is not an existing handler, the function is applied as the `onload` handler.

```
function addLoadEvent(fn)
{
  var currentHandler = window.onload;
  if (typeof currentHandler != 'function')
  {
    window.onload = fn;
  }
  else
  {
    window.onload = function()
    {
      if (currentHandler)
      {
        currentHandler();
      }
      fn();
    }
  }
}
```

Note that the function to be executed when the page is loaded is passed as a parameter to the `addLoadEvent` function.

This mechanism allows any number of `onload` handlers to be chained together.

See also

- The *The confirmation dialog* and *Pressing Enter to submit* recipes in this chapter show how to handle other JavaScript events.
- The *The character counter* recipe in this chapter shows how to use jQuery to bind a JavaScript function to specified events.

Collapsible list elements

When a number of records and related information are rendered as a list, a user is often presented with a large amount of data that they must scroll through in order to access the items that they are interested in. One way to improve this is to allow items to be collapsed, showing enough headline information to allow the item to be identified, but taking up the minimum amount of screen real estate.

In this recipe, we will create a Visualforce page that displays a list of account records and their associated contact records. Each account record is collapsed when the page is initially rendered, and the user may click a record to expand it and see the associated contact information.

Visualforce provides collapsible behavior for the standard `<apex:pageBlockSection />` component. However, this forces the content to be expanded or collapsed to be nested inside this component, which styles the content in a similar fashion to a standard Salesforce section. The solution presented in this recipe provides this behavior for a regular HTML table that may be used in a variety of situations with or without Salesforce styling.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework and jQuery User Interface (`http://jqueryui.com/`) library to produce, style, and transition the tooltip. The JavaScript and CSS files are included from the Google Hosted Libraries content delivery network rather than being uploaded as Salesforce static resources, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

This recipe also makes use of a custom controller, so this will need to be present before the Visualforce page can be created.

How to do it...

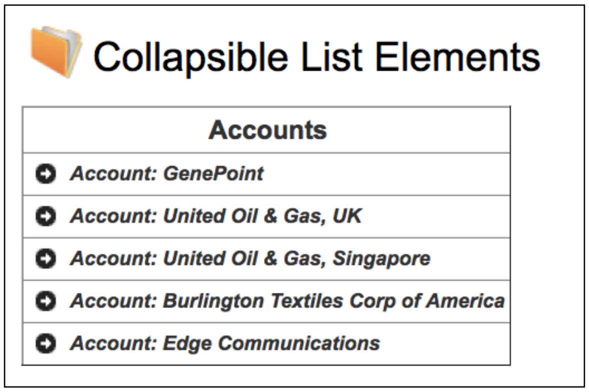
1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `CollapsibleController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

6. Click on the **New** button.
7. Enter `Collapsible` in the **Label** field.
8. Accept the default **Collapsible** that is automatically generated for the **Name** field.
9. Paste the contents of the `Collapsible.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **Collapsible** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Collapsible** page: `https://<instance>/apex/Collapsible`.

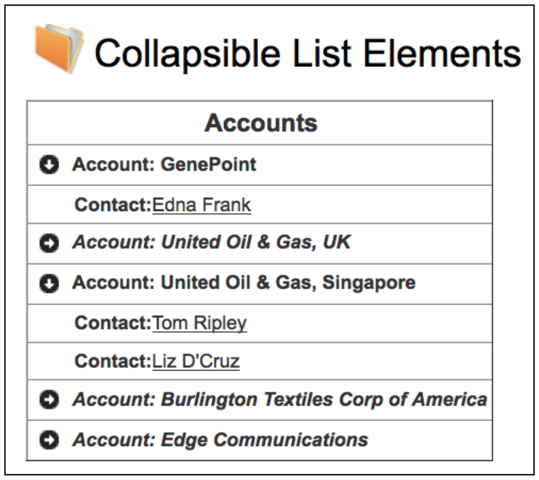
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Collapsible List Elements

Accounts
Account: GenePoint
Account: United Oil & Gas, UK
Account: United Oil & Gas, Singapore
Account: Burlington Textiles Corp of America
Account: Edge Communications

Clicking on the right arrow icon next to the account name opens the account record to display the associated contact records.



Accounts
● **Account: GenePoint**
Contact: Edna Frank
● ***Account: United Oil & Gas, UK***
● **Account: United Oil & Gas, Singapore**
Contact: Tom Ripley
Contact: Liz D'Cruz
● ***Account: Burlington Textiles Corp of America***
● ***Account: Edge Communications***

The list of accounts is rendered as an HTML `<table>` element. Two `<tbody>` elements are rendered for each account, the first containing just the account name that is visible when the page is rendered.

```
<tbody id="{!acc.id}-collapsed}">
  <tr onclick="toggle('{!acc.id}');">
    <td colspan="2">
      <span style="float:left;" class="ui-accordion-header-icon ui-
icon ui-icon-circle-arrow-e">&nbsp;</span>
      &nbsp;
      <span style="font-weight:bold; font-style:italic;">Account:
{!acc.Name}</span>
    </td>
  </tr>
</tbody>
```

The second `<tbody>` element contains the account name and the associated contact records, which is hidden when the page is initially rendered.

```
<tbody id="{!acc.id}-expanded" style="display:none">
  <tr onclick="toggle('{!acc.id}');">
    <td colspan="2">
      <span style="float:left;" class="ui-accordion-header-icon ui-
icon ui-icon-circle-arrow-s">&nbsp;</span>
      &nbsp;
      <span style="font-weight:bold">Account: {!acc.Name}</span>
    </td>
  </tr>
  <apex:repeat value="{!acc.Contacts}" var="cont">
    <tr>
      <td style="border-right: none">
        &nbsp;
      </td>
      <td style="border-left: none">
        <span style="font-weight:bold">Contact:</span>
        <apex:outputLink
            value="/{!cont.id}">{!cont.Name}
        </apex:outputLink>
      </td>
    </tr>
  </apex:repeat>
</tbody>
```

Each `<tbody>` element defines an `onclick` handler.

```
onclick="toggle('{!acc.id}');"
```

The `toggle` function uses the jQuery `toggle` function (`http://api.jquery.com/toggle/`) to swap the visibility of both `<tbody>` elements associated with an account, hiding the currently visible element and showing the currently hidden element.

```
function toggle(baseId)
{
  $('tbody[id*="' + baseId + '"]').toggle();
}
```

See also

- The *Tooltips* recipe in this chapter shows how jQuery can be used to generate styled tooltips that appear via a custom transition when a user hovers over an element.

The scrolling news ticker

A **scrolling news ticker** is a common feature on many websites, providing an eye-catching way to present headlines or updates to users.

In this recipe we will create a Visualforce page that displays a list of news items based on 10 recently closed opportunities. The page initially displays the first three opportunities and then scrolls through the remaining, removing the top item, moving the others up vertically, and appending the next item to the bottom of the list.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework to scroll the list of stories. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to a new version simply by changing the URL of the included file.

This recipe makes use of a wrapper class to encapsulate the stories.

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `Story.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.

How to do it...

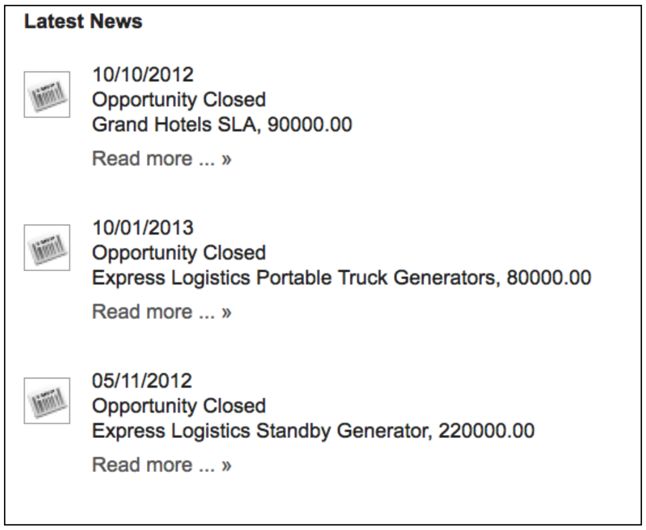
1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `NewsTickerController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `NewsTicker` in the **Label** field.
8. Accept the default **NewsTicker** that is automatically generated for the **Name** field.

9. Paste the contents of the `NewsTicker.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **NewsTicker** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **NewsTicker** page: `https://<instance>/apex/NewsTicker`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



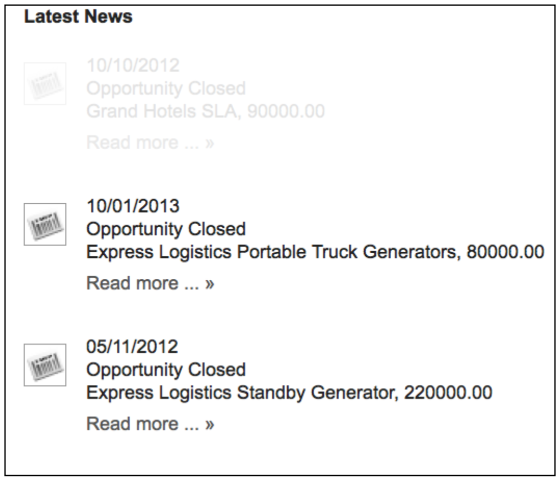
Latest News

10/10/2012
Opportunity Closed
Grand Hotels SLA, 90000.00
Read more ... »

10/01/2013
Opportunity Closed
Express Logistics Portable Truck Generators, 80000.00
Read more ... »

05/11/2012
Opportunity Closed
Express Logistics Standby Generator, 220000.00
Read more ... »

After a short interval, the top story fades out and the items scroll up.


Latest News

10/10/2012
Opportunity Closed
Grand Hotels SLA, 90000.00
Read more ... »

10/01/2013
Opportunity Closed
Express Logistics Portable Truck Generators, 80000.00
Read more ... »

05/11/2012
Opportunity Closed
Express Logistics Standby Generator, 220000.00
Read more ... »

The news items are generated from the list of story custom object instances managed by the custom controller. Each story displays an image, the publication date, the headline, a snippet of the body, and a read more link.

```
<ul id="listticker">
  <apex:repeat value="{!stories}" var="story">
    <li>
      <img style="float:left; margin: 5px 0 0 0; padding:1px;
border:1px solid #999999;"
         src="{!story.image}" height="24px" width="24px" />
      <p style="margin: 0 0 0 40px;">
        <apex:outputText value="{0, date, dd/MM/yyyy}">
          <apex:param value="{!story.pubDate}"/>
        </apex:outputText><br/>
        <apex:outputText value="{!story.headline}"/><br/>
        <apex:outputText value="{!story.snippet}"/><br/>
        <apex:outputLink style="text-decoration:none; margin: 5px 0 0
0; line-height:1.3em; display:block; color: #555;"
         value="/{!story.id}">Read more ... &raquo;
        </apex:outputLink>
      </p>
    </li>
  </apex:repeat>
</ul>
```

The list is sized to display the first three items with the remainder hidden from the view.

```
#listticker{
  height:260px;
  overflow:hidden;
  padding:0 0 14px 0;
}
```

A JavaScript function utilizes jQuery functionality to fade out the top element in the list and remove it.

```
function removeTop()
{
  first = $('ul#listticker li:first').html();
  $('ul#listticker li:first')
      .animate({opacity: 0}, speed)
      .fadeOut('slow', function() {$(this).remove();});
  addBottom(first);
}
```

The item removed from the list is passed to the `addBottom()` JavaScript function, which appends the item to the bottom of the list.

```
function addBottom(first)
{
  var last = '<li>'+first+'</li>';
  $('ul#listticker').append(last)
}
```

The standard JavaScript `setInterval()` function executes the `removeTop()` function at the specified interval.

```
interval = setInterval(removeTop, pause);
```

There's more...

While this recipe uses opportunities, the actual news items are instances of a wrapper class. This allows the recipe to be easily adapted to use any other standard or custom Salesforce sObject type.

See also

- The *Carousel messages* recipe in this chapter shows how to display a carousel of messages at the top of a page.

Carousel messages

A **carousel** is a mechanism for giving viewers of a web page access to a number of content items (for example, messages, news stories, and images) via a single element on the page. They are usually implemented as sliding or rotating horizontal panels where the content is updated after a set period of time.

In this recipe we will create a Visualforce page that rotates a carousel of recently closed opportunities. The page displays a single opportunity at a time and transitions between opportunities by fading out the old one and fading in the new one. We will then create a homepage component containing this page and add it to the **Home** page layout.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to a new version simply by changing the URL of the included file.

This recipe also uses the bxSlider (`http://bxslider.com/`) JavaScript library to provide the carousel functionality. As this is not available from a content delivery network, it must be present as a static resource.

1. Download the `jquery.bxslider.zip` file by navigating to the bxSlider homepage `http://bxslider.com/` and clicking on the **Download** button on the top-right of the page.
2. Navigate to the Static Resource setup page by clicking on **Your Name** | **Setup** | **Develop** | **Static Resources**.
3. Click on the **New** button.
4. Enter `BXSlider` in the **Name** field.
5. Enter `bxslider carousel` in the **Description** field.
6. Click on the **Browse** button and select the `jquery.bxslider.zip` file downloaded in step 1.

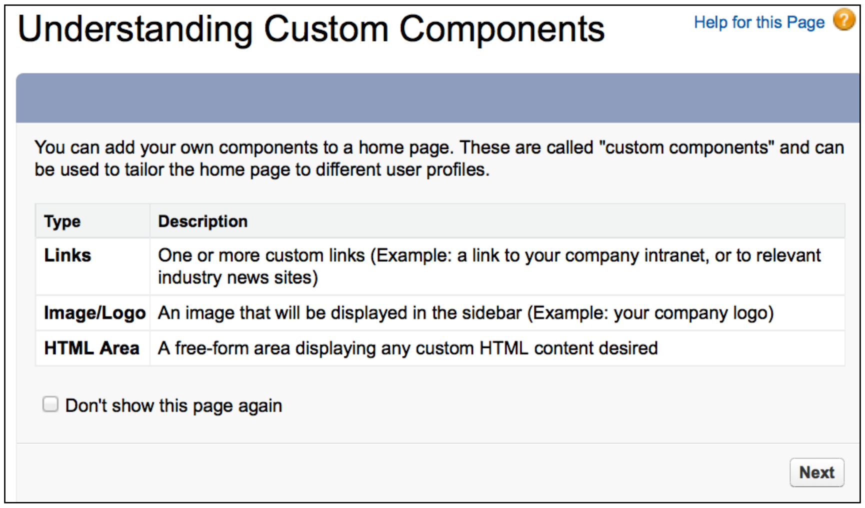
   The **Browse** button may have a different label depending on the browser; using Google Chrome, for example, results in a label of **Choose File**.

7. Accept the default **Private** value for the **Cache Control** field and click on the **Save** button.

The **Cache Control** field values are only applicable for a static resource that is used in a Force.com site.

How to do it...

1. First, create the custom controller for the Visualforce page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `BannerController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `Banner` in the **Label** field.
8. Accept the default **Banner** that is automatically generated for the **Name** field.
9. Paste the contents of the `Banner.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualfrce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **Banner** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.
13. Next, create the homepage component by navigating to the Home Page Components setup page by clicking **Your Name** | **Setup** | **Customize** | **Home** | **Home Page Components**.
14. Scroll down to the **Custom Components** section and click on the **New** button.
15. If the **Understanding Custom Components** information screen appears, as shown in the following screenshot, click on the **Next** button:



You can add your own components to a home page. These are called "custom components" and can be used to tailor the home page to different user profiles.

Type	Description
Links	One or more custom links (Example: a link to your company intranet, or to relevant industry news sites)
Image/Logo	An image that will be displayed in the sidebar (Example: your company logo)
HTML Area	A free-form area displaying any custom HTML content desired

☐ Don't show this page again

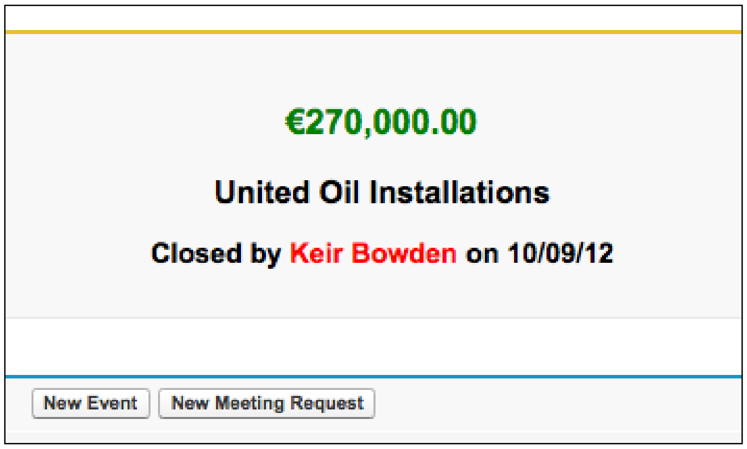
16. On the next page, **Step 1. New Custom Components**, enter `Top Sales` in the **Name** field, select the **HTML Area** option, and click on the **Next** button.
17. On the next page, **Step 2. New Custom Components**, select the **Wide (Right) Column** option.
18. Select the **Show HTML** box.
19. Paste the following markup into the editable area:

```
<iframe style="border: none" src="/apex/Banner"  seamless=""
width="100%"></iframe>
```

20. Click on the **Save** button.
21. Next, add the new component to one or more homepage layouts. Navigate to **Your Name** | **Setup** | **Customize** | **Home** | **Home Page Layouts**.
22. Locate the name of the homepage layout you wish to add the component to and click on the **Edit** link.
23. On the resulting page, **Step 1. Select the Components to show**, select the **Top Sales** box in the **Select Wide Components to Show** section and click on the **Next** button.
24. On the next page, **Step 2. Order the Components**, use the arrow buttons to move the **Top Sales** component to the top position in the **Wide (Right) Column** list and click on the **Save** button.
25. Repeat steps 22 to 24 for any other homepage layouts that will contain the sidebar component.

How it works...

Opening the homepage displays the **Top Sales** component.



€270,000.00

United Oil Installations

Closed by Keir Bowden on 10/09/12

The opportunities iterated by the carousel are output as an HTML unordered list.

```
<ul id="slider1">
  <apex:repeat value="{!opps}" var="opp">
    <li style="text-align:center; width: 280px; height:150px;">
      <p style="font-size:20px; font-weight:bold; color:green">
        <apex:outputField value="{!opp.Amount}"/>
      </p>
      <p style="font-size:18px; font-weight:bold">
        <apex:outputText value="{!opp.Name}"/>
      </p>
      <p style="font-size:16px; font-weight:bold">Closed by
        <span style="color: red">{!opp.Owner.Name}</span> on
        <apex:outputText value="{0, date, dd/MM/yy}">
           <apex:param value="{!opp.CloseDate}"/>
        </apex:outputText>
      </p>
    </li>
  </apex:repeat>
</ul>
```

When the page is loaded, the bxSlider is activated via JavaScript.

```
$('#slider1').bxSlider({
                auto: true,
                controls:false,
                mode:'fade',
                pause:8000,
                pager:false
                });
```

See also

- The *The scrolling news ticker* recipe in this chapter shows how to display a vertically scrolling ticker of news stories.
- There are a number of configuration options available for bxSlider. Visit `http://bxslider.com/options` for more details.

Hiding buttons on submit

When a user clicks on a button to submit a form, if they don't receive any feedback that the click was successful, they are likely to click the button again, resulting in a double form submission. Disabling buttons or the form when a button is clicked can introduce browser compatibility issues, as some browsers will interpret this as a request to cancel the form submission.

In this recipe we will create a Visualforce page that allows a user to edit some basic information about a contact. When the user clicks on the **Save** or **Cancel** button to save or discard their changes, the buttons will be swapped out with a pair of disabled buttons containing text to indicate that the form submission is taking place.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework to swap the buttons. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

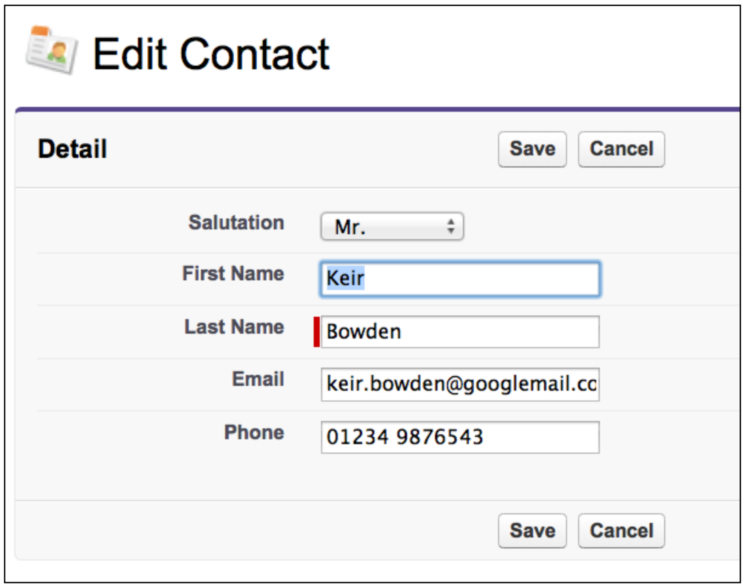
How to do it...

1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `WorkingButtons` in the **Label** field.
4. Accept the default **WorkingButtons** that is automatically generated for the **Name** field.
5. Paste the contents of the `WorkingButtons.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **WorkingButtons** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

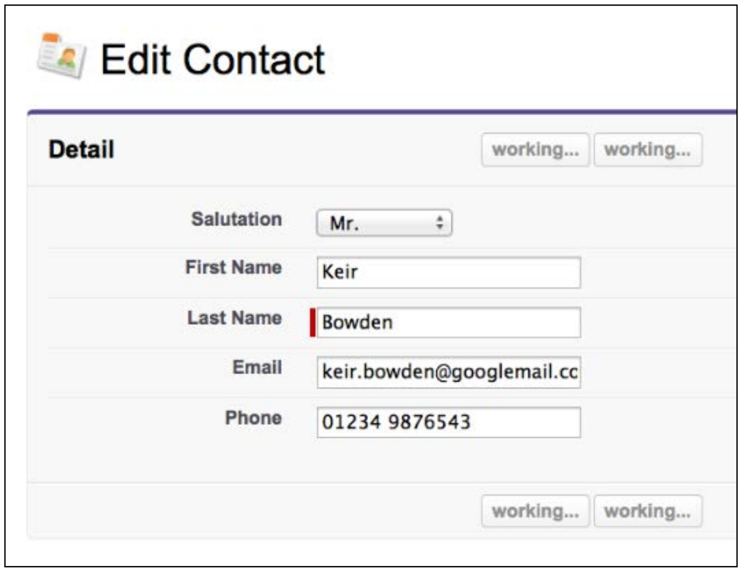
Opening the following URL in your browser displays the **WorkingButtons** page: `https://<instance>/apex/WorkingButtons?id=<contact_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the ID of any contact in your Salesforce instance.



Salutation	Mr.
First Name	Keir
Last Name	Bowden
Email	keir.bowden@googlemail.cc
Phone	01234 9876543

Editing details of the record and clicking on the **Save** button swaps out the buttons with their disabled equivalents.



Salutation: Mr.
First Name: Keir
Last Name: Bowden
Email: keir.bowden@googlemail.cc
Phone: 01234 9876543

Four buttons are rendered when the page is initially displayed; two visible and two hidden.

```
<apex:commandButton id="commandbtn1" value="Save" action="{!save}"
    onclick="workingButtons()"/>
<apex:commandButton id="commandbtn2" value="Cancel"
      action="{!cancel}" immediate="true"
      onclick="workingButtons()"/>
<apex:commandButton id="workingbtn1" value="working..."
      disabled="true" style="display:none" />
<apex:commandButton id="workingbtn2" value="working..."
      disabled="true" style="display:none" />
```

Each of the **Save** and **Cancel** buttons defines an `onclick` handler that executes a JavaScript function to swap the displayed buttons with the hidden buttons using the jQuery `toggle` function.

```
    function workingButtons()
    {
       $('[id*="commandbtn"]').toggle();
       $('[id*="workingbtn"]').toggle();
    }
```

No value is returned from the `onclick` hander; this indicates the browser that the standard behavior of continuing with the form submission should take place.

See also

- The *The confirmation dialog* recipe in this chapter shows how the standard browser behavior of continuing with a form submission can be stopped if a user so chooses.
- The *Collapsible list elements* recipe in this chapter shows how the jQuery `toggle` function can be used to open/close a section of a list of records.

Client-side validation

Carrying out validation in the browser when a user submits a form is a technique that has been popular for a number of years. The user is given immediate feedback rather than having to wait for a round trip to the server, and bandwidth is saved by not submitting the form if there are issues.

In this recipe we will create a Visualforce page that allows a user to create a contact record. When the user clicks on **Save**, client-side validation will take place to ensure that one of the e-mail address or phone number fields has been populated before submitting the form.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework to swap the buttons. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

This recipe also uses the jQuery Validation plugin to provide the validation functionality. This is included in the Microsoft Ajax Content Delivery Network.

> For more information about the jQuery Validation plugin, visit `http://jqueryvalidation.org/`.

How to do it...

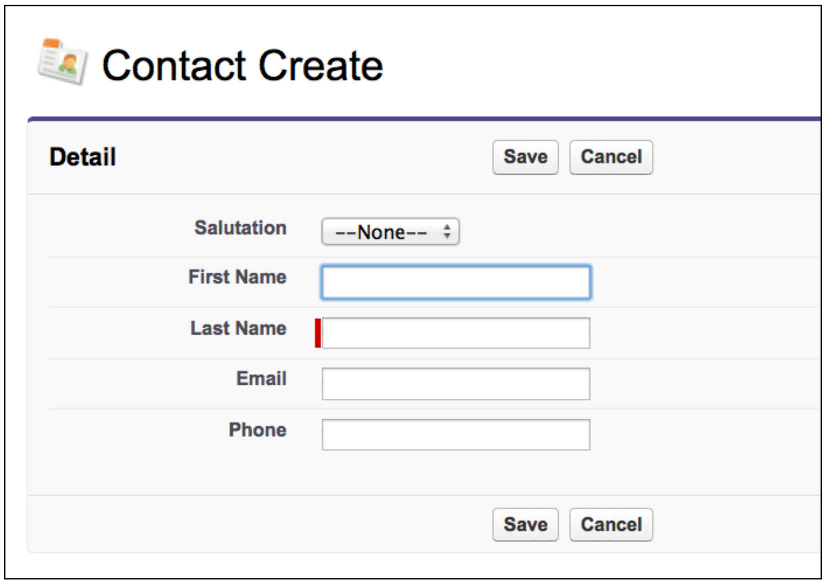
1. Create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `ClientSideValidation` in the **Label** field.
4. Accept the default **ClientSideValidation** that is automatically generated for the **Name** field.

5. Paste the contents of the `ClientSideValidation.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **ClientSideValidation** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

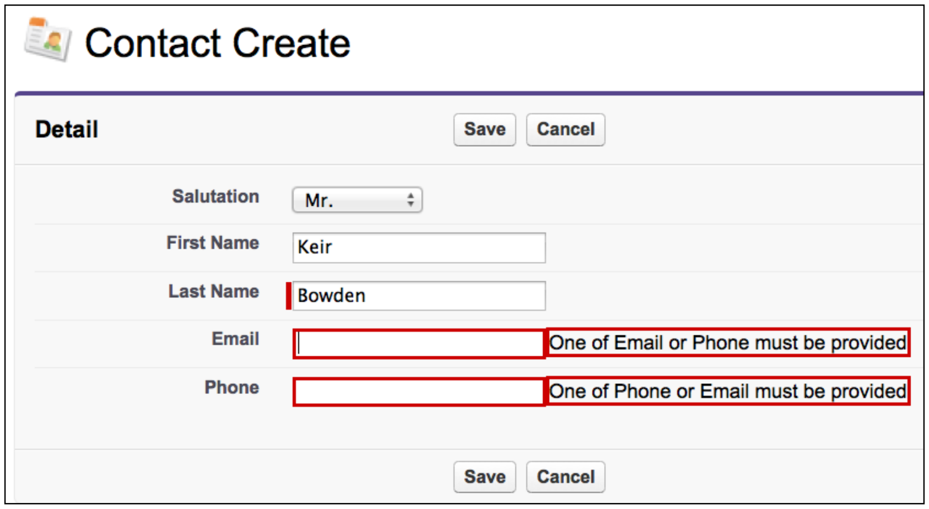
How it works...

Opening the following URL in your browser displays the **ClientSideValidation** page: `https://<instance>/apex/ClientSideValidation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Leaving both the **Email** and **Phone** fields blank when saving the record.



Salutation	Mr.
First Name	Keir
Last Name	Bowden
Email	One of Email or Phone must be provided
Phone	One of Phone or Email must be provided

When the page is loaded, the jQuery Validation Plugin is activated. Each of the **Email** and **Phone** fields is defined as required if the other field is empty,

```
               '{!$Component.createform.pblock.pbsection.
contactemail}': {
                                required: function() {
                                   return $('[id="{!$Component.
createform.pblock.pbsection.contactphone}"]').val()=='';
                                }
                             },  // email
```

Each field has an associated message that is displayed when the field is required but empty.

```
messages: {
   '{!$Component.createform.pblock.pbsection.contactemail}':
                      "One of Email or Phone must be provided",
   '{!$Component.createform.pblock.pbsection.contactphone}':
                      "One of Phone or Email must be provided",
} // messages
```

Note that as a user may have old browser software or have chosen to disable JavaScript, any client-side validation should also be replicated on the server side to ensure that the validation rules are applied.

See also

- The *Adding error messages to field inputs* recipe in *Chapter 3*, *Capturing Data Using Forms* shows how to validate server side that one of the e-mail and phone number fields is populated when creating a contact record.

Trapping navigation away

When a user is filling in a form, inadvertently clicking on a link to another page, or generating the page back, the keyboard shortcut sends the browser to a new page and discards all user inputs. In the event that the form is large and complex, this can represent a significant lost effort.

In this recipe we will create a Visualforce page that allows a user to create a contact record. If the user clicks on a button to save the record or cancel the creation, they will be requested to confirm that they wish to continue with the action. If the user clicks on the **Save** or **Cancel** button, this will submit the form without further confirmation.

Getting ready

This recipe uses the jQuery (`http://jquery.com/`) JavaScript framework to swap the buttons. The JavaScript file is included from the Google Hosted Libraries content delivery network rather than being uploaded as a Salesforce static resource, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

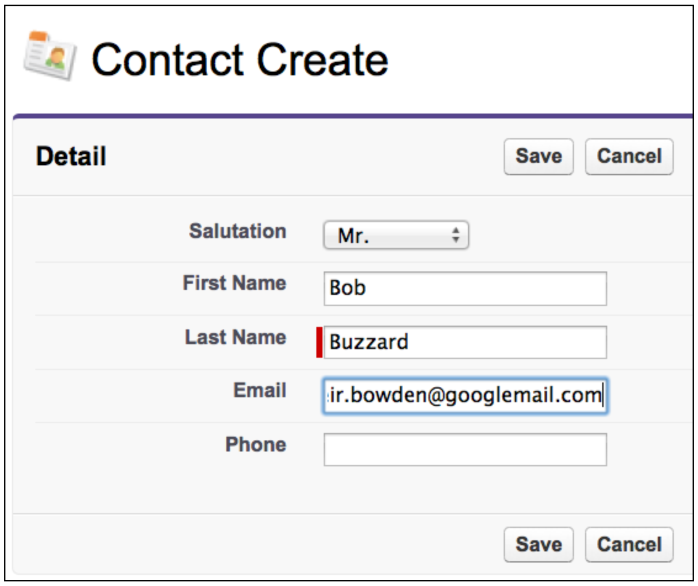
How to do it...

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `ConfirmLeavePage` in the **Label** field.
4. Accept the default **ConfirmLeavePage** that is automatically generated for the **Name** field.
5. Paste the contents of the `ConfirmLeavePage.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **ConfirmLeavePage** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **ConfirmLeavePage** page: `https://<instance>/apex/ConfirmLeavePage`.

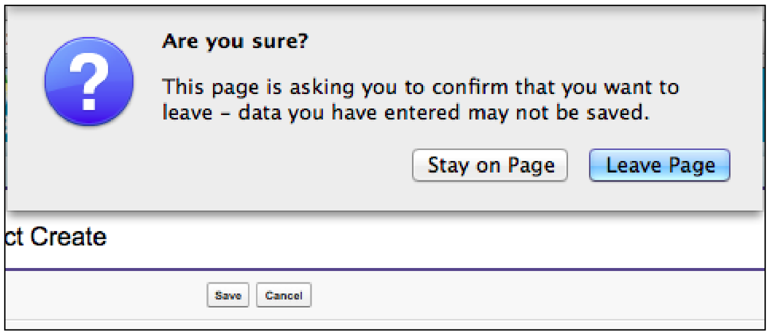
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.


Contact Create

Detail

Salutation	Mr.
First Name	Bob
Last Name	Buzzard
Email	ir.bowden@googlemail.com
Phone	

If the user clicks any of the other tabs on the page, a confirmation dialog asks them to confirm that they wish to leave the page.


Are you sure?

This page is asking you to confirm that you want to leave – data you have entered may not be saved.

When the page is loaded, jQuery is used to add an `onbeforeunload` event handler.

```
window.onbeforeunload = function()
{
    return 'This will lose any unsaved changes you have made';
}
```

The `onbeforeunload` event is fired when a page is about to unload its resources.

Each of the **Save** and **Cancel** buttons defines an `onclick` handler that removes the `onbeforeunload` handler, allowing the action to continue without requiring the user to confirm they wish to continue.

```
<apex:commandButton value="Save" action="{!save}"
                  onclick="clearConfirm();"/>
      ...
function clearConfirm()
{
  window.onbeforeunload=null;
}
```

See also

- The *The confirmation dialog* recipe in this chapter shows how to ask the user to confirm they wish to continue when they click on a button to submit a form.
- The *Pressing Enter to submit* recipe in this chapter intercepts the pressing of an *Enter* key in a form and asks the user to confirm they wish to save their changes.

8
Force.com Sites

In this chapter, we will cover the following recipes:

- Creating a site
- Record and field access
- Retrieving content from Salesforce
- Web to lead form
- Creating a website template
- Adding a header menu to a template
- Adding a sidebar to a template
- Conditional rendering in templates

Introduction

Force.com sites allow public websites to be created in and hosted by Salesforce, removing the requirement to configure, secure, and manage a web server. Visualforce pages that have direct access to Salesforce data via the page controller generate the site content.

In this chapter, we will create a Force.com site initially containing static content. We will then create a set of template pages to remove repetition of common markup. Finally, we will provide access to Salesforce data from a public website, allowing visitors to access records without logging in to Salesforce.

Unlike earlier chapters in this book, these recipes are best performed in order, as many recipes build on knowledge gained in earlier recipes and the first recipe, *Creating a site*, configures the Force.com site that is used to serve the content for all of the remaining recipes.

[Salesforce supports an additional technology to host websites, Site.com, which does not use Visualforce to generate content. For more information on Site.com visit `http://wiki.developerforce.com/page/Site.com`.]

Creating a site

In this recipe we will configure a Force.com site that displays a single page. The contents of the page are static, and the page will be publicly available to unauthenticated visitors.

Getting ready

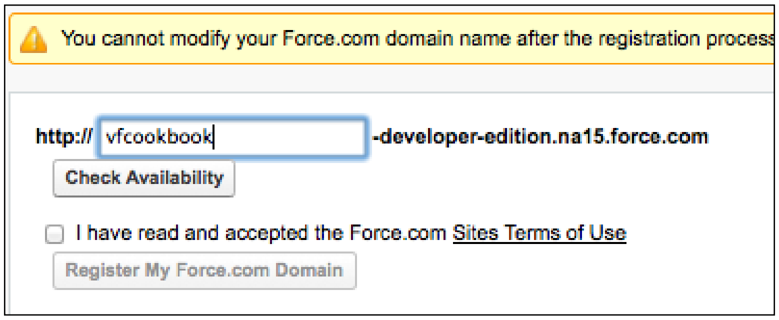
This recipe uses the Bootstrap framework (`http://twitter.github.io/bootstrap/`) to style the page. The JavaScript and CSS files are included from the BootStrapCDN (`http://www.bootstrapcdn.com/index.html`) content delivery network rather than being uploaded as Salesforce static resources, as this makes it straightforward to move to new versions simply by changing the URL of the included file. Bootstrap in turn relies on the jQuery (`http://jquery.com/`) JavaScript framework. This is included from the Google Hosted Libraries content delivery.

[This does introduce a dependency on the BootstrapCDN and Google Hosted Libraries sites and in the event that either site was unavailable or access blocked, the Bootstrap styling and functionality would be lost.]

Before the site can be configured, a subdomain prefix must be selected. This prefix will be used to generate the unique domain for the site.

1. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.

2. On the resulting page, choose your preferred subdomain, check the box to indicate you accept the terms of use, and click on the **Register My Force.com Domain** button.


You cannot modify your Force.com domain name after the registration process

http:// vfcookbook -developer-edition.na15.force.com

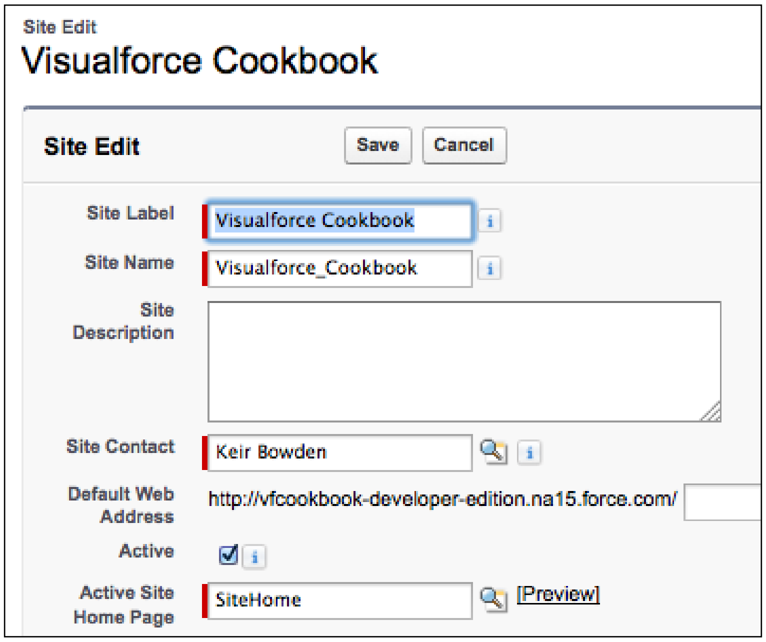
Check Availability
I have read and accepted the Force.com Sites Terms of Use
Register My Force.com Domain

Note that once you have chosen your domain name, it cannot be modified. As this domain will be used as the prefix for all Force.com sites created in your Salesforce instance, it should be a representative of your organization rather than a particular site.

How to do it...

1. First, create the Visualforce page that will be displayed by the site by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `SiteHome` in the **Label** field.
4. Accept the default **SiteHome** that is automatically generated for the **Name** field.
5. Paste the contents of the `SiteHome.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the site by navigating to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
7. Click on the **New** button.
8. Enter `Visualforce Cookbook` in the **Label** field.
9. Accept the default **Visualforce_Cookbook** that is automatically generated for the **Name** field.
10. Check the **Active** box to make the site active as soon as the configuration is saved.
11. Enter `SiteHome` in the **Active Site Home Page** field.

12. Leave all other fields with their default values and click on the **Save** button.



Site Edit

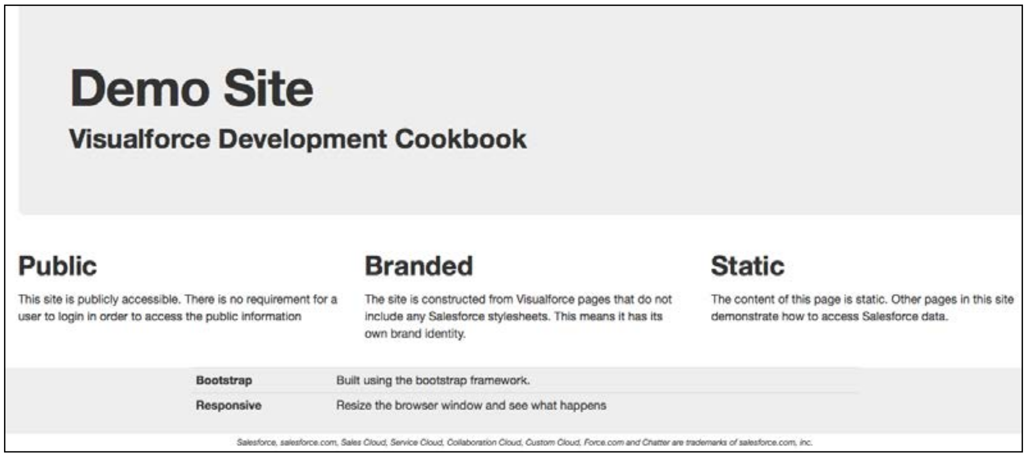
Visualforce Cookbook

Site Label	Visualforce Cookbook
Site Name	Visualforce_Cookbook
Site Description	
Site Contact	Keir Bowden
Default Web Address	http://vfcookbook-developer-edition.na15.force.com/
Active	✓
Active Site Home Page	SiteHome [Preview]

How it works...

Opening the following URL in your browser displays the **SiteHome** page: `http://<domain>/SiteHome`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Demo Site

Visualforce Development Cookbook

Public

This site is publicly accessible. There is no requirement for a user to login in order to access the public information

Branded

The site is constructed from Visualforce pages that do not include any Salesforce stylesheets. This means it has its own brand identity.

Static

The content of this page is static. Other pages in this site demonstrate how to access Salesforce data.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

The standard header, sidebar, and stylesheets are hidden via attributes in the enclosing `<apex:page/>` standard component. As the page contains `<html>` and `<body>` tags, additional attributes specify that Visualforce should not insert its own version of these.

```
<apex:page applyHtmlTag="false" applyBodyTag="false" sidebar="false"
showHeader="false"
    standardStyleSheets="false">
```

Bootstrap divides the page into a 12-column grid and style classes are defined to allow content to span a number of columns.

```
<div class="span4">
  <h2>Public</h2>
  <p>
    This site is publicly accessible. There is no requirement for
    a user to login in order to access the public information
  </p>
</div><!-- .span4 -->
```

See also

- The *Retrieving content from Salesforce* recipe in this chapter shows how to dynamically generate content for a Force.com site.
- The *Web to lead form* recipe in this chapter shows how to capture data into Salesforce from a Force.com site.

Record and field access

A common source of confusion for Visualforce developer is configuring a Force.com site to allow unauthenticated access to Salesforce records and specific fields. This is usually configured via the Profiles menu located at **Your Name** | **Setup** | **Administration Setup** | **Profiles**. However, access to records and fields for a Force.com site is configured via the setup page for the site in question.

In this recipe we will configure the Force.com site created in the first recipe to allow public access to contact records. We will then create a Visualforce page that allows a visitor to enter an e-mail address into a form on the Force.com site and extract the contact record matching the e-mail address, displaying the **First Name**, **Last Name**, and **Email** fields from the contact record.

Getting ready

This recipe requires that you have already completed the *Creating a Site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

How to do it...

1. First, add access to the contact sObject to Guest User Profile for the site. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
2. Click on the **Visualforce Cookbook** link in the **Sites** section.
3. Click on the **Public Access Settings** button: this displays the Guest user profile for the site.
4. On the resulting page click the **Edit** button, select the **Read** checkbox for the **Contact** sObject in the **Custom Object Permissions** sections, and click on the **Save** button.
5. Next, add the required field access for the profile. Scroll down to the **Field Level Security** section and click on the **[View]** link for the **Contact** element.
6. On the resulting page, confirm that the **Visible** checkbox for the **Email** field is selected. If it is not, click on the **Edit** button and select the checkbox, and then click on the **Save** button.

[Note that the **Name** field is always visible to all profiles, so no action needs to be taken for that field.]

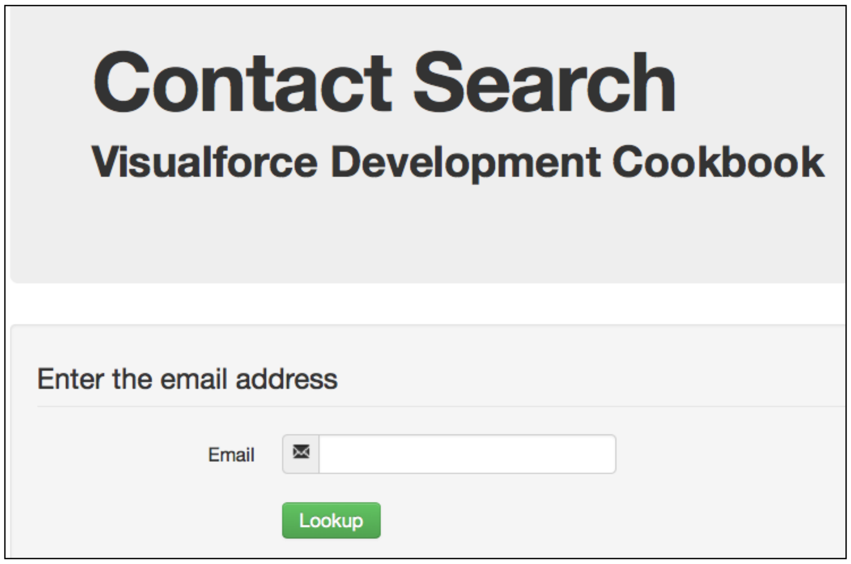
7. Next, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
8. Click on the **New** button.
9. Paste the contents of the `RetrieveContactController.cls` Apex class from the code download into the Apex Class area.
10. Click on the **Save** button.
11. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Click on the New button.
13. Enter `RetrieveContact` in the **Label** field.
14. Accept the default **RetrieveContact** that is automatically generated for the **Name** field.

15. Paste the contents of the `RetrieveContact.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
16. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
17. Click on the **Visualforce Cookbook** link in the **Sites** section.
18. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
19. On the resulting page, **Enable Visualforce Page Access**, select **RetrieveContact** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **RetrieveContact** page: `http://<domain>/RetrieveContact`.

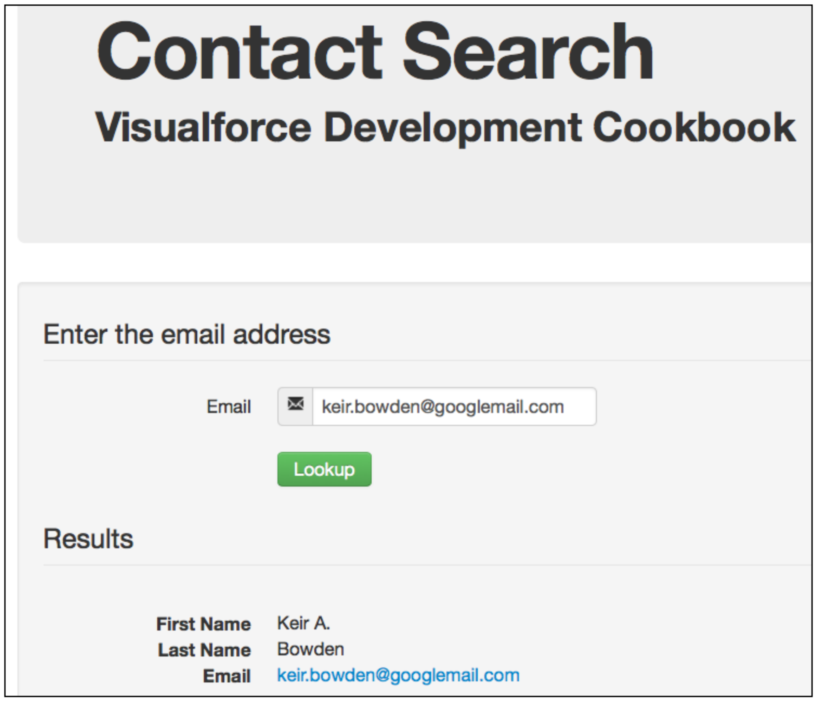
Here `<domain>` is the Force.com domain name chosen while configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.


Contact Search

Visualforce Development Cookbook

Enter the email address

Filling in the **Email** field and clicking on the **Lookup** button displays the details of the first contact with a matching e-mail address.



Email	keir.bowden@googlemail.com

First Name	Keir A.
Last Name	Bowden
Email	keir.bowden@googlemail.com

See also

- The *Creating a site* recipe in this chapter shows how to set up a Force.com site and make a static page publicly available.
- The *Retrieving content from Salesforce* recipe in this chapter shows how to extract records from Salesforce and use these to generate content on a Force.com site.

Retrieving content from Salesforce

Force.com sites allow unauthenticated visitors access to custom (and some standard) Salesforce sObjects. While providing public access to data stored in Salesforce might seem like a security risk, it is a perfect fit for dynamically generated website content; content authors don't need to know how to edit Visualforce pages in order to be able to update the content of the site.

In this recipe we will create a Visualforce page that renders content from the three most recently edited records of a custom sObject. We will then make this page publicly available via an unauthenticated Force.com site.

Getting ready

This recipe requires that you have already completed the *Creating a site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

This recipe requires a custom sObject that encapsulates the items to display on the site.

1. First, create the site content custom sObject by navigating to **Your Name** | **Setup** | **Create** | **Objects**.
2. Click on the **New Custom Object** button.
3. Enter `SiteItem` in the **Label** field.
4. Enter `SiteItems` in the **Plural Label** field.
5. Leave all other input values at their defaults and click on the **Save** button.
6. On the resulting page, create the field that will contain the content detail, scroll down to the **Custom Fields and Relationships** section, and click on the **New** button.
7. On the next page, **Step 1. Choose the field type**, select **Text Area** from the **Data Type** radio buttons and click on the **Next** button.
8. On the next page, **Step 2. Enter the details**, enter `Detail` in the **Field Label** field, leave all other fields at their default values, and click on the **Next** button.
9. On the next page, **Step 3. Establish field-level security for reference field**, leave all the fields at their default values and click on the **Next** button.
10. On the final page, **Step 4. Add to page layouts**, leave all the fields at their default values and click on the **Save** button.
11. Next, create a tab for the SiteItem object to allow easy record creation by navigating to **Your Name** | **Setup** | **Create** | **Tabs**.
12. Click on the **New** button in the **Custom Object Tabs** section.
13. On the next page, **Step 1. Enter the details**, choose **SiteItem** from the **Object** picklist, click on the lookup icon in the **Tab Style** field, and choose a style from the resulting popup. Finally, click on the **Next** button.
14. On the next page, **Step 2. Add to profiles**, leave all fields at their default values and click on the **Next** button.
15. On the next page, **Step 3. Add to custom apps**, leave all fields at their default values and click on the **Save** button.
16. Next, add access to the custom sObject to **Guest User Profile** for the site. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
17. Click on the **Visualforce Cookbook** link in the **Sites** section.
18. Click on the **Public Access Settings** button.

19. On the resulting page click on the **Edit** button, select the **Read** checkbox for the **SiteItem** sObject in the **Custom Object Permissions** sections, and click on the **Save** button.
20. Next, add access to the **Detail** field for the **Guest User Profile**. Scroll down to the **Field Level Security** section and click on the **[View]** link for the **SiteItem** element.
21. On the resulting page, click on the **Edit** button, select the **Visible** checkbox for the **Detail** field, and click on the **Save** button.
22. Finally, create at least three **SiteItem** records.

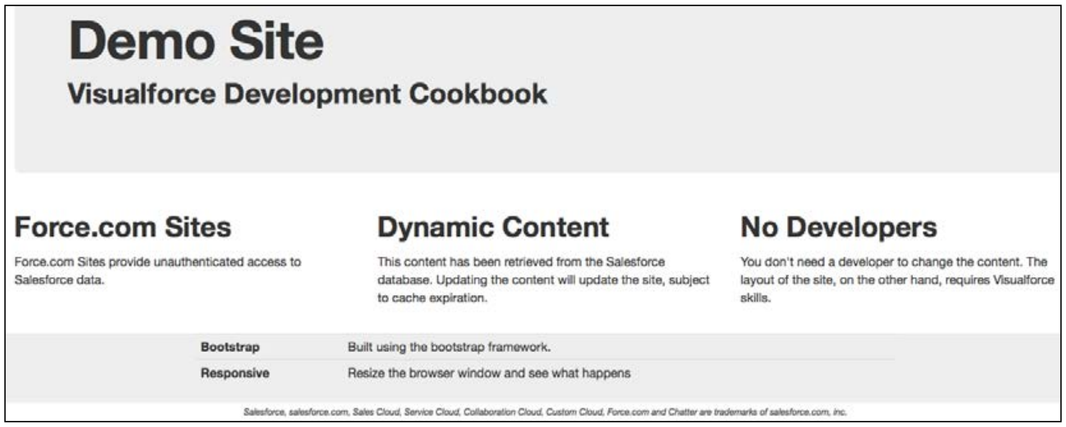
How to do it...

1. First, create the custom controller by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `SiteItemController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `SiteItem` in the **Label** field.
8. Accept the default **SiteItem** that is automatically generated for the **Name** field.
9. Paste the contents of the `SiteItem.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
11. Click on the **Visualforce Cookbook** link in the **Sites** section.
12. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
13. On the resulting page, **Enable Visualforce Page Access**, select **SiteItem** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **SiteItem** page: `http://<domain>/SiteItem`.

Here, `<domain>` is the Force.com domain name chosen while configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Demo Site

Visualforce Development Cookbook

Force.com Sites

Force.com Sites provide unauthenticated access to Salesforce data.

Dynamic Content

This content has been retrieved from the Salesforce database. Updating the content will update the site, subject to cache expiration.

No Developers

You don't need a developer to change the content. The layout of the site, on the other hand, requires Visualforce skills.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

Bootstrap divides the page into a 12-column grid and style classes are defined to allow content to span a number of columns. The **SiteItem** records from the custom controller are iterated to generate a row of data. As the controller only returns three records, each iteration element spans four grid columns.

```
<div class="row-fluid">
  <apex:repeat value="{!items}" var="item">
    <div class="span4">
      <h2>{!item.Name}</h2>
      <p>{!item.Detail__c}</p>
    </div><!-- .span4 -->
  </apex:repeat>
 </div> <!-- row-fluid -->
```

See also

- The *Creating a site* recipe in this chapter shows how to set up a Force.com site and make a static page publicly available.
- The *Web to lead form* recipe in this chapter shows how to capture data into Salesforce from a Force.com site.

Web to lead form

The standard Salesforce web to lead functionality allows a form to be embedded into a company's website to capture information that is then turned into a lead in the company's Salesforce instance. The form is submitted to a servlet that is common to all Salesforce instances and thus, may not be customized besides sending the user to a thank you page that is disconnected from the lead.

For more information on web to lead, visit `http://login.salesforce.com/help/doc/en/customize_leadcapture.htm`.

In this recipe we will create a Visualforce page that captures a lead and redirects the user to a personalized thank you page that displays the ID of the lead for future reference. We will then make this page publicly available via an unauthenticated Force.com site.

Getting ready

This recipe requires that you have already completed the *Creating a site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

How to do it...

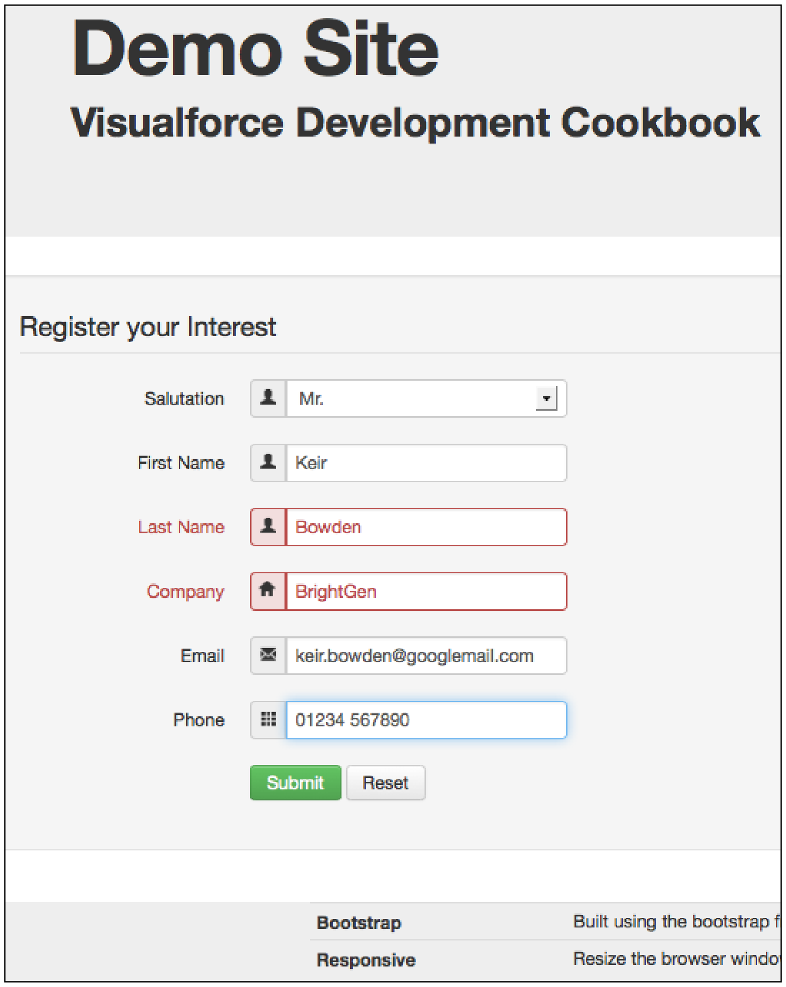
1. First, create the thank you Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `WebToLeadThanks` in the **Label** field.

4. Accept the default **WebToLeadThanks** that is automatically generated for the **Name** field.
5. Paste the contents of the `WebToLeadThanks.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the web to lead page controller extension by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
7. Click on the **New** button.
8. Paste the contents of the `WebToLeadExt.cls` Apex class from the code download into the Apex Class area.
9. Click on the **Save** button.
10. Next, create the web to lead Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Click on the **New** button.
12. Enter `WebToLead` in the **Label** field.
13. Accept the default **WebToLead** that is automatically generated for the **Name** field.
14. Paste the contents of the `WebToLead.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
15. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
16. Click on the **Visualforce Cookbook** link in the **Sites** section.
17. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
18. On the resulting page, **Enable Visualforce Page Access**, select **WebToLead** and **WebToLeadThanks** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.
19. Click on the **Public Access Settings** button.
20. On the resulting page click on the **Edit** button, select the **Read** and **Create** checkboxes for the **Lead** object in the **Standard Object Permissions** sections, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **WebToLead** page: `http://<domain>/WebToLead`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Demo Site

Visualforce Development Cookbook

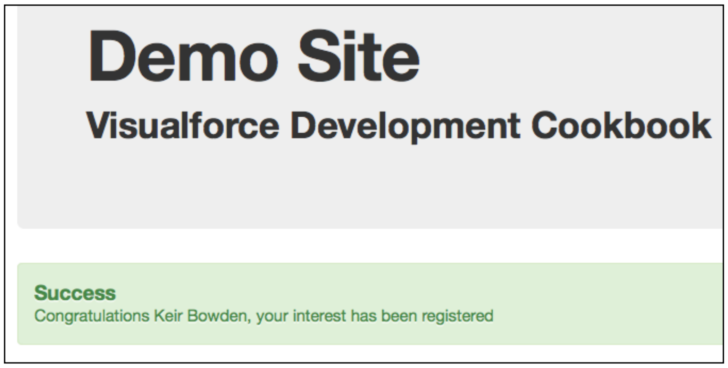
Register your Interest

Salutation	Mr.
First Name	Keir
Last Name	Bowden
Company	BrightGen
Email	keir.bowden@googlemail.com
Phone	01234 567890

Bootstrap	Built using the bootstrap f
Responsive	Resize the browser windo

For a full list of the icons available in Bootstrap, visit `http://getbootstrap.com/2.3.2/base-css.html#icons`.

Filling out and submitting the form takes the visitor to the personalized thank you page.



Success
Congratulations Keir Bowden, your interest has been registered

The form is laid out as a series of div elements using the Bootstrap style class `control-group`.

```
<div class="control-group">
  <apex:outputLabel styleClass="control-label"
    for="firstname" value="First Name" />
  <div class="controls">
    <div class="input-prepend">
      <span class="add-on"><i class="icon-user"></i></span>
      <apex:inputText id="firstname" value="{!Lead.FirstName}" />
    </div>
  </div>
</div>
```

The `<div>` class with the `input-prepend` style class works in conjunction with the enclosed `<span>` element to prepend the icon to the input element.

The Bootstrap `error style` class is used to indicate that a value is required for a form field.

```
<div class="control-group error">
  <apex:outputLabel styleClass="control-label" for="lastname"
        value="Last Name" />
  <div class="controls">
    <div class="input-prepend">
      <span class="add-on"><i class="icon-user"></i></span>
      <apex:inputText id="lastname" value="{!Lead.LastName}" />
    </div>
  </div>
</div>
```

Note that the `<apex:inputText />` components are used to capture the user input rather than an `<apex:inputField />` component. This stops the required field classes being added to the input element, which would otherwise cause the input element to appear underneath the icon.

See also

- The *Creating a site* recipe in this chapter shows how to set up a Force.com site and make a static page publicly available.
- The *Retrieving content from Salesforce* recipe in this chapter shows how to dynamically generate content for a Force.com site.

Creating a website template

In the previous recipes in this chapter, each page contained the entire Visualforce markup needed to display its content. This leads to repetition of common markup, to display headers and footers for example. In the event that the header or footer content needs to be changed, every page on a site needs to be updated with the new markup.

Visualforce provides a solution to this issue—templates. These allow the common elements of a site to be added to a template that is used as the starting point for rendering any page. The page then injects its specific content into the template at appropriate points.

In this recipe we will create a template version of the **SiteItem** Visualforce page from the *Retrieving content from Salesforce* recipe, where the template provides the header and footer markup. We will then make this page available publicly available via an unauthenticated Force.com site.

Getting ready

This recipe requires that you have already completed the *Creating a site* and *Retrieving content from Salesforce* recipes, as it relies on the custom domain and Force.com site created in the first recipe, and the custom objects and controller from the second.

How to do it...

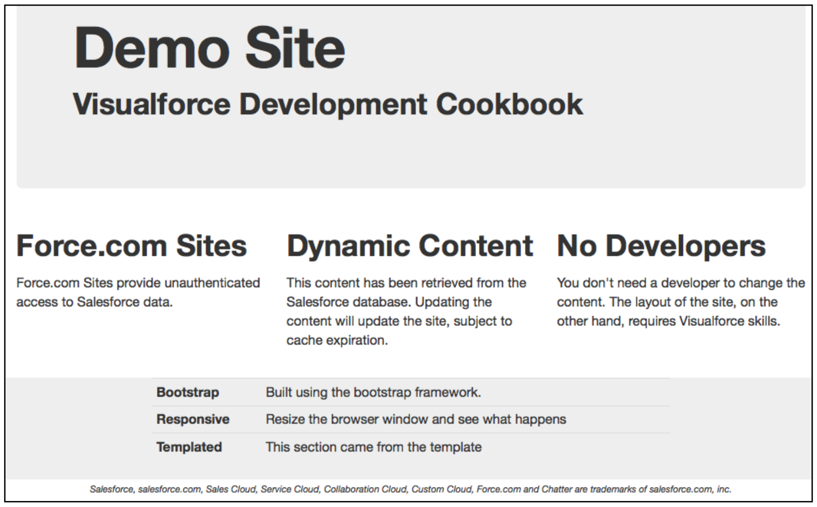
1. First, create the template; this is simply another Visualforce page. To do this, navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CookbookTemplate` in the **Label** field.
4. Accept the default **CookbookTemplate** that is automatically generated for the **Name** field.

5. Paste the contents of the `CookbookTemplate.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `SiteItemTemplated` in the **Label** field.
9. Accept the default **SiteItemTemplated** that is automatically generated for the **Name** field.
10. Paste the contents of the `SiteItemTemplated.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
11. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
12. Click on the **Visualforce Cookbook** link in the **Sites** section.
13. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
14. On the resulting page, **Enable Visualforce Page Access**, select **CookbookTemplate** and **SiteItemTemplated** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **SiteItemTemplated** page: `http://<domain>/SiteItemTemplated`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Demo Site

Visualforce Development Cookbook

Force.com Sites

Force.com Sites provide unauthenticated access to Salesforce data.

Dynamic Content

This content has been retrieved from the Salesforce database. Updating the content will update the site, subject to cache expiration.

No Developers

You don't need a developer to change the content. The layout of the site, on the other hand, requires Visualforce skills.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens
Templated	This section came from the template

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

The template defines the common content and where pages utilizing the template can inject their content. In the following code snippet, the `div` element with the style class of `hero-unit` generates the generic header and then inserts the body content provided by the page based on the template.

```
<!--  container -->
<div class="container-fluid">
  <div class="hero-unit">
    <h1>Demo Site</h1>
    <h2>Visualforce Developer Cookbook</h2>
  </div>

  <apex:insert name="body" />

</div> <!-- container -->
```

Note that there is no enclosing markup to indicate the page is a template.

The page making use of the template defines the content that will be inserted into the template when rendered, and encloses this in an `<apex:composition/>` component.

```
<apex:composition template="CookbookTemplate">
  <apex:define name="Title">
    Force.com Sites Recipe 4
  </apex:define>
       ...
</apex:composition>
```

See also

- The *Creating a site* recipe in this chapter shows how to set up a Force.com site and make a static page publicly available.
- The *Adding a header menu to a template* recipe in this chapter shows how to add a navigation menu to the header of a page template.

Adding a header menu to a template

A common requirement for a website is to display a navigation menu as part of the header. In the scenario where each page defines its own header and footer, it is straightforward to highlight a menu option to indicate the page that is currently being displayed. When a template provides the header and footer information, a mechanism is required to allow the page to identify itself to the template, which can then highlight the appropriate menu option.

In this recipe, we will create a Visualforce template that provides header and footer content to four other Visualforce pages: a **Home** page, an **About** page, a **Contact** page, and a **Links** page. We will then make these pages available publicly available via an unauthenticated Force.com site.

Getting ready

This recipe requires that you have already completed the *Creating a site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

How to do it...

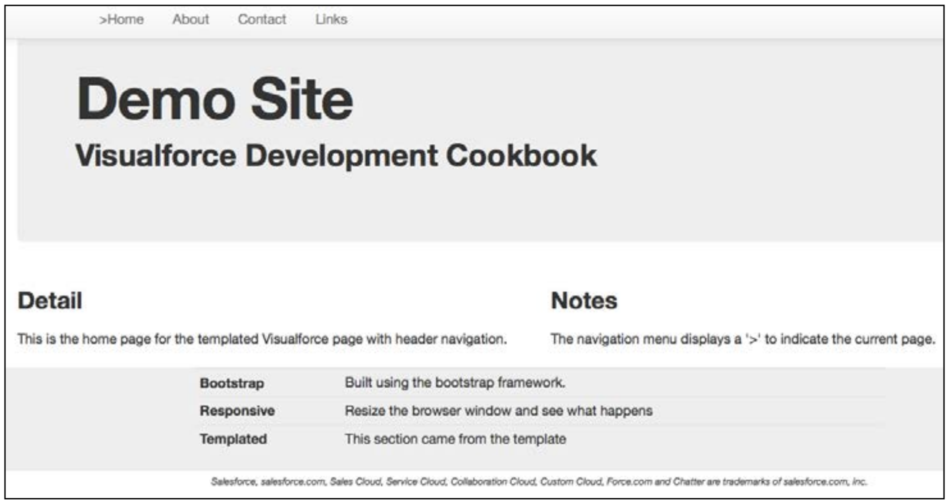
1. First, create the template; this is simply another Visualforce page. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CookbookTemplateV2` in the **Label** field.
4. Accept the default **CookbookTemplateV2** that is automatically generated for the **Name** field.
5. Paste the contents of the `CookbookTemplateV2.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the Home Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `Home` in the **Label** field.
9. Accept the default **Home** that is automatically generated for the **Name** field.

10. Paste the contents of the `Home.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
11. Next, create the About Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Click on the **New** button.
13. Enter `About` in the **Label** field.
14. Accept the default **About** that is automatically generated for the **Name** field.
15. Paste the contents of the `About.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
16. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
17. Next, create the Contact Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
18. Click on the **New** button.
19. Enter `Contact` in the **Label** field.
20. Accept the default **Contact** that is automatically generated for the **Name** field.
21. Paste the contents of the `Contact.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
22. Next, create the Links Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
23. Click on the **New** button.
24. Enter `Links` in the **Label** field.
25. Accept the default **Links** that is automatically generated for the **Name** field.
26. Paste the contents of the `Links.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
27. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
28. Click on the **Visualforce Cookbook** link in the **Sites** section.
29. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
30. On the resulting page, **Enable Visualforce Page Access**, select **CookbookTemplateV2**, **Home**, **About**, **Contact**, and **Links** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Home** page: `http://<domain>/Home`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Demo Site

Visualforce Development Cookbook

Detail

This is the home page for the templated Visualforce page with header navigation.

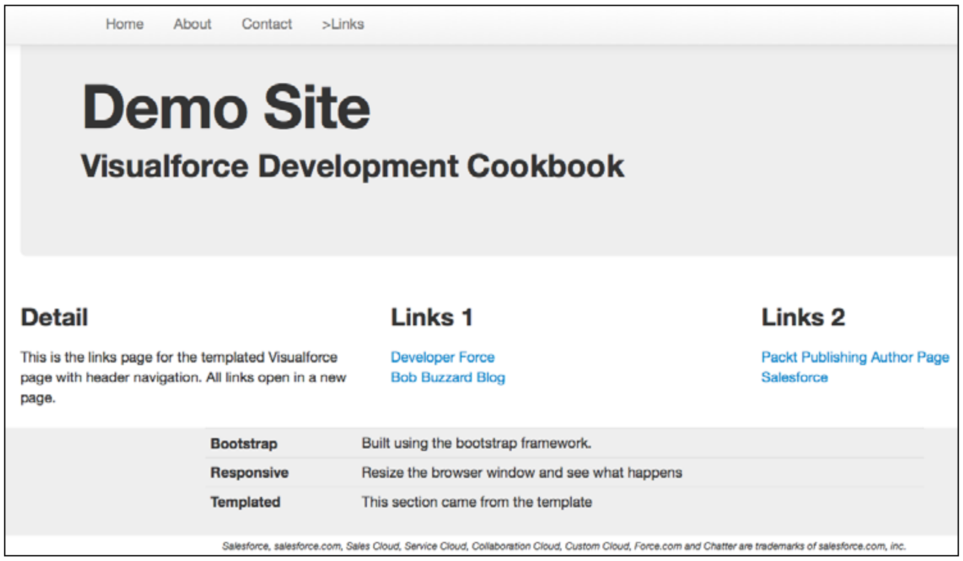
Notes

The navigation menu displays a '>' to indicate the current page.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens
Templated	This section came from the template

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

Clicking on any of the other pages in the header navigation bar updates the bar to indicate the current page.



Demo Site

Visualforce Development Cookbook

Detail

This is the links page for the templated Visualforce page with header navigation. All links open in a new page.

Links 1

Developer Force
Bob Buzzard Blog

Links 2

Packt Publishing Author Page
Salesforce

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens
Templated	This section came from the template

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

The template defines an empty value as the default.

```
<apex:variable var="page" value="" />
```

Each page defines markup to be injected into the template that overrides the `page` variable with its own value.

```
<apex:define name="page">
  <apex:variable var="page" value="home"/>
</apex:define>
```

The template inserts this markup:

```
<apex:insert name="page" />
```

The template then conditionally outputs a **>** character next to the menu item that equates to the page.

```
<div class="nav-collapse collapse">
  <ul class="nav">
    <li><a href="/apex/Home">
      <apex:outputText value=">" rendered="{!page=='home'}" />
      Home</a></li>
    <li><a href="/apex/About">
      <apex:outputText value=">" rendered="{!page=='about'}" />
      About</a></li>
    <li><a href="/apex/Contact">
      <apex:outputText value=">" rendered="{!page=='contact'}" />
      Contact</a></li>
    <li><a href="/apex/Links">
      <apex:outputText value=">" rendered="{!page=='links'}" />
      Links</a></li>
  </ul>
</div>
```

See also

- The *Retrieving content from Salesforce* recipe in this chapter shows how to dynamically generate content for a Force.com site.
- The *Adding a sidebar to a template* recipe in this chapter shows how to add a sidebar component to a template.

Adding a sidebar to a template

Website content is not always suited to being broken up across a number of pages. A table of contents, while being potentially quite long, may be more usable when displayed on a single page. A sidebar can improve the user experience by providing links to allow the user to rapidly navigate around the lengthy content.

In this recipe, we will create a Visualforce template that provides header and footer content to two other Visualforce pages: a **TableOfContents** page (containing information about all chapters) and a **Chapter1** page (containing detailed information about the first chapter). Each page has a sidebar to assist with navigation through the page content. We will then make these pages available publicly available via an unauthenticated Force.com site.

Getting ready

This recipe requires that you have already completed the *Creating a site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

How to do it...

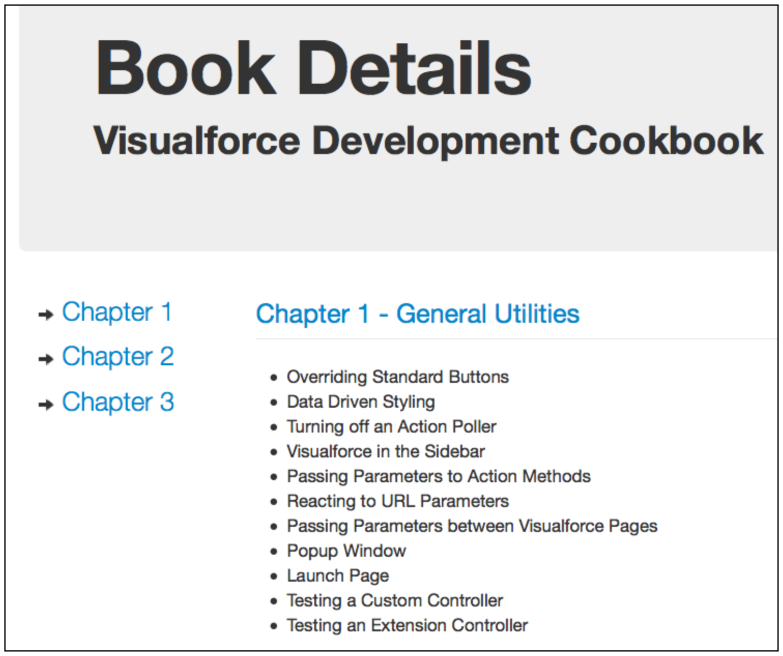
1. First, create the template; this is simply another Visualforce page. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CookbookTemplateV3` in the **Label** field.
4. Accept the default **CookbookTemplateV3** that is automatically generated for the **Name** field.
5. Paste the contents of the `CookbookTemplateV3.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the TableOfContents Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `TableOfContents` in the **Label** field.
9. Accept the default **TableOfContents** that is automatically generated for the **Name** field.
10. Paste the contents of the `TableOfContents.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
11. Next, create the Chapter1 Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Click on the **New** button.

13. Enter `Chapter1` in the **Label** field.
14. Accept the default **Chapter1** that is automatically generated for the **Name** field.
15. Paste the contents of the `Chapter1.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
16. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
17. Click on the **Visualforce Cookbook** link in the **Sites** section.
18. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
19. On the resulting page, **Enable Visualforce Page Access**, select **CookbookTemplateV3**, **TableOfContents**, and **Chapter1** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **Home** page: `http://<domain>/TableOfContents`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Book Details

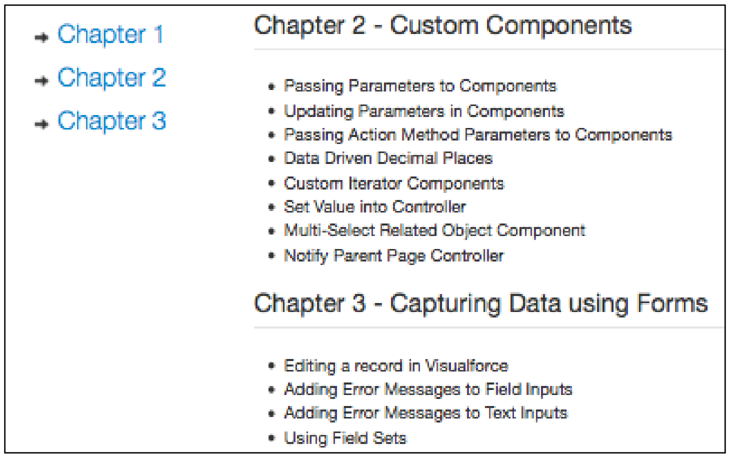
Visualforce Development Cookbook

- Chapter 1
- Chapter 2
- Chapter 3

Chapter 1 - General Utilities

- Overriding Standard Buttons
- Data Driven Styling
- Turning off an Action Poller
- Visualforce in the Sidebar
- Passing Parameters to Action Methods
- Reacting to URL Parameters
- Passing Parameters between Visualforce Pages
- Popup Window
- Launch Page
- Testing a Custom Controller
- Testing an Extension Controller

Scrolling down or clicking on one of the navigation links in the sidebar allows quick navigation to the particular section.



Chapter 2 - Custom Components

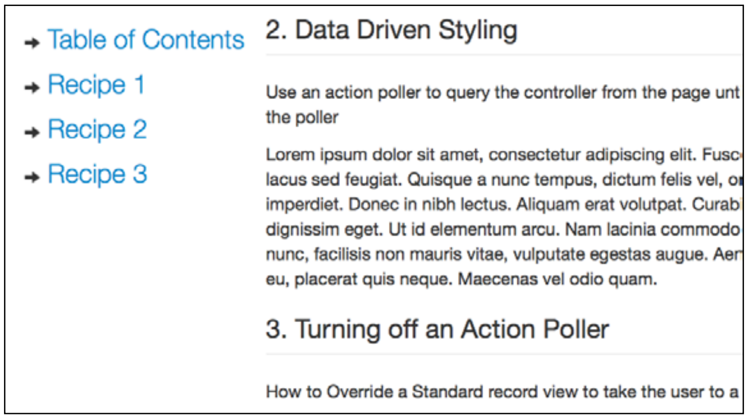
- Passing Parameters to Components
- Updating Parameters in Components
- Passing Action Method Parameters to Components
- Data Driven Decimal Places
- Custom Iterator Components
- Set Value into Controller
- Multi-Select Related Object Component
- Notify Parent Page Controller

Chapter 3 - Capturing Data using Forms

- Editing a record in Visualforce
- Adding Error Messages to Field Inputs
- Adding Error Messages to Text Inputs
- Using Field Sets

Note that even though the header has scrolled out of view, the navigation sidebar remains in view; this functionality is provided by the Bootstrap Affix JavaScript. For more information visit `http://twitter.github.io/bootstrap/javascript.html#affix`.

Clicking on the **Chapter 1 – General Utilities** link takes the user to the detail page for the first chapter, which also has a navigation sidebar.



2. Data Driven Styling

Use an action poller to query the controller from the page unt
the poller

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusc
lacus sed feugiat. Quisque a nunc tempus, dictum felis vel, o
imperdiet. Donec in nibh lectus. Aliquam erat volutpat. Curab
dignissim eget. Ut id elementum arcu. Nam lacinia commodo
nunc, facilisis non mauris vitae, vulputate egestas augue. Aer
eu, placerat quis neque. Maecenas vel odio quam.

3. Turning off an Action Poller

How to Override a Standard record view to take the user to a

The Bootstrap framework divides the page up into a 12-column grid. The template provides a 3-column container for the sidebar and a 9-column container for the details.

```
<div class="row-fluid">
  <div class="span3">
    <apex:insert name="sidebar" />
  </div>

  <div class="span9">
    <apex:insert name="body" />
  </div>

</div> <!-- row -->
```

Each page that utilizes the template provides the list of sidebar navigation elements and the content associated with each navigation element.

```
<apex:define name="sidebar">
  <ul class="nav nav-list" data-spy="affix" data-offset-top="270">
    <li><a href="#chapter1"><i class="icon-arrow-right"></i>
      <span class="lead">Chapter 1</span></a></li>
        ...
  </ul>
</apex:define>
<apex:define name="body">
  <fieldset id="chapter1">
    <legend><a href="/apex/Chapter1">
      Chapter 1 - General Utilities</a>
    </legend>
    <div class="fieldset-content">
      <ul>
        <li>Overriding Standard Buttons</li>
              ...
      </ul>
    </div>
  </fieldset>
</apex:define>
```

It is important to make sure that every navigation list item has associated content, or the user will experience viewing of content by clicking on links only in some instances.

See also

- The *Retrieving content from Salesforce* recipe in this chapter shows how to dynamically generate content for a Force.com site.
- The *Adding a header menu to a template* recipe in this chapter shows how to add a navigation menu to the header of a page template.

Conditional rendering in templates

Templating a website is an effective way to avoid repeated content and the associated maintenance overhead. There are occasions when this common content needs to be replaced for one or two exceptional pages; for example, a homepage may require slightly different header information than other pages in a site. This problem can be solved by the homepage not utilizing a template, but this then means that any common content that the homepage does require is repeated in the homepage and the template.

In this recipe we will create a Visualforce template that provides header and footer content. A page may override the header text provided by the template. We will then create two Visualforce pages that utilize this template: a **StandardHeader** page (that displays the standard header text) and a **CustomHeader** page (that provides its own custom text for use in the header). We will then make these pages available publicly available via an unauthenticated Force.com site.

Getting ready...

This recipe requires that you have already completed the *Creating a site* recipe, as it relies on the custom domain and Force.com site created in that recipe.

How to do it...

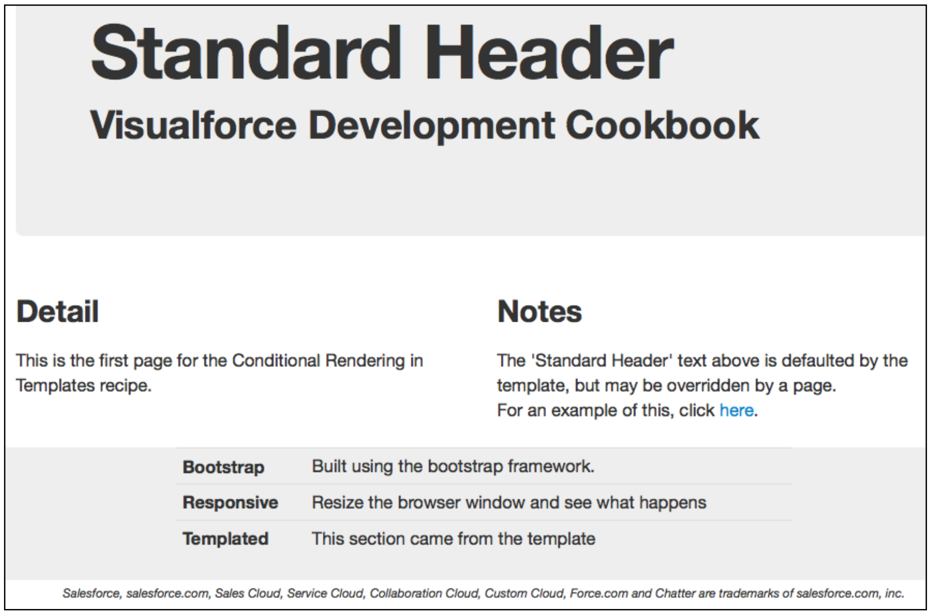
1. First, create the template; this is simply another Visualforce page. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `CookbookTemplateV4` in the **Label** field.
4. Accept the default **CookbookTemplateV4** that is automatically generated for the **Name** field.

5. Paste the contents of the `CookbookTemplateV4.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Next, create the StandardHeader Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Click on the **New** button.
8. Enter `StandardHeader` in the **Label** field.
9. Accept the default **StandardHeader** that is automatically generated for the **Name** field.
10. Paste the contents of the `StandardHeader.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
11. Next, create the CustomHeader Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
12. Click on the **New** button.
13. Enter `CustomHeader` in the **Label** field.
14. Accept the default **CustomHeader** that is automatically generated for the **Name** field.
15. Paste the contents of the `CustomHeader.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
16. Navigate to the Sites setup page by clicking on **Your Name** | **Setup** | **Develop** | **Sites**.
17. Click on the **Visualforce Cookbook** link in the **Sites** section.
18. On the resulting page, scroll down to the **Site Visualforce Pages** list and click on the **Edit** button.
19. On the resulting page, **Enable Visualforce Page Access**, select **CookbookTemplateV4**, **StandardHeader**, and **CustomHeader** from the **Available Visualforce Pages** list, click on the Add icon to add it to the **Enabled Visualforce Pages** list, and click on the **Save** button.

How it works...

Opening the following URL in your browser displays the **StandardHeader** page: `http://<domain>/StandardHeader`.

Here, `<domain>` is the Force.com domain name chosen when configuring the site, for example, `vfcookbook-developer-edition.na15.force.com`.



Standard Header

Visualforce Development Cookbook

Detail

This is the first page for the Conditional Rendering in Templates recipe.

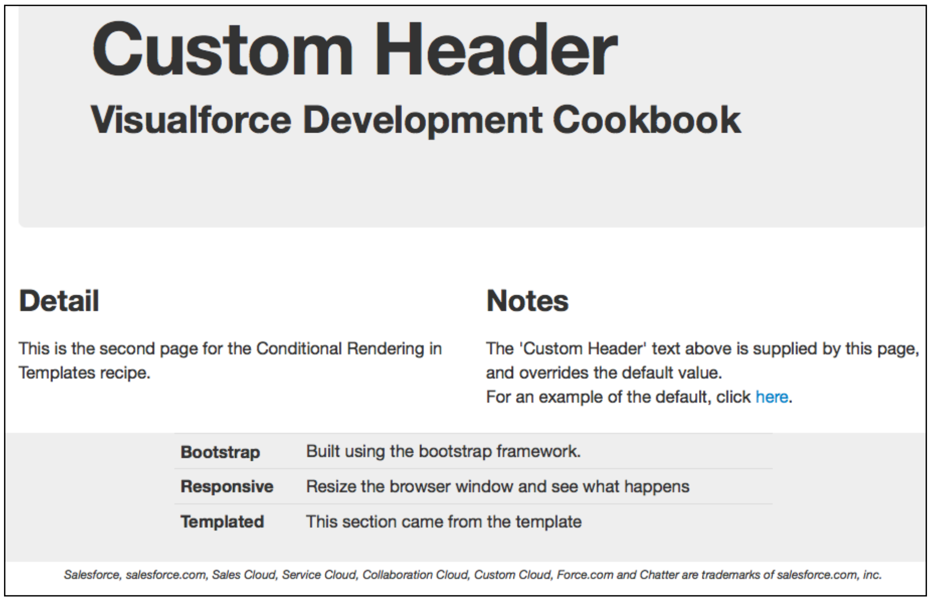
Notes

The 'Standard Header' text above is defaulted by the template, but may be overridden by a page.
For an example of this, click here.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens
Templated	This section came from the template

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

Clicking on the **here** link in the **Notes** section displays the **CustomHeader** page with a different text in the header section.



Custom Header

Visualforce Development Cookbook

Detail

This is the second page for the Conditional Rendering in Templates recipe.

Notes

The 'Custom Header' text above is supplied by this page, and overrides the default value.
For an example of the default, click here.

Bootstrap	Built using the bootstrap framework.
Responsive	Resize the browser window and see what happens
Templated	This section came from the template

Salesforce, salesforce.com, Sales Cloud, Service Cloud, Collaboration Cloud, Custom Cloud, Force.com and Chatter are trademarks of salesforce.com, inc.

The **CookbookTemplateV4** page defines a variable to indicate whether the standard header text should be used, and provides a mechanism for a page to inject Visualforce markup to override this.

```
<apex:variable var="header" value="standard" />
<apex:insert name="headerOverride" />
```

The standard header text is conditionally rendered based on the value of the header variable.

```
<apex:outputPanel rendered="{!header=='standard'}">
  Standard Header
</apex:outputPanel>
```

If the page has overridden the value of the header variable to `'custom'`, a mechanism is provided for the page-specific header text to be injected.

```
<apex:outputPanel rendered="{!header=='custom'}">
  <apex:insert name="customHeaderText" />
</apex:outputPanel>
```

A page may override the header text through the following markup:

```
<apex:define name="headerOverride">
  <apex:variable var="header" value="custom"/>
</apex:define>
<apex:define name="customHeaderText">
  Custom Header
</apex:define>
```

See also

- The *Retrieving content from Salesforce* recipe in this chapter shows how to dynamically generate content for a Force.com site.
- The *Adding a header menu to a template* recipe in this chapter shows how to add a navigation menu to the header of a page template.
- The *Adding a sidebar to a template* recipe in this chapter shows how to add a sidebar component to a template.

9 jQuery Mobile

In this chapter, we will cover the following recipes:

- Mobilizing a Visualforce page
- Navigation and transitions
- Adding a navigation bar
- Working with dialogs
- Listing records
- Mobile Visualforce forms
- Redirecting to the mobile page based on the browser
- Storing the user's location
- Scanning the QR code to access the page

Introduction

Users today expect mobile access to the same applications and data that they have on their desktop or laptop computer, and failure to provide this type of access can lead to a lack of adoption of an application.

There are three types of mobile applications:

- **Native**: These applications provide access to all the features on a device and have the potential for the slickest user experience and best performance. The downsides are that a separate application needs to be built for each platform that is supported using platform-specific tools and languages, and distributing and upgrading an application is often constrained by the platform (to distribute an iOS application, for example, requires membership of the Apple Developer Program).

- **HTML5**: These are web applications that are accessed via the device browser. They do not have access to many device features and have limitations around offline storage and session management. The key benefit to HTML5 mobile applications is that one application runs on any device and every user accesses the latest version of the application.
- **Hybrid**: These applications allow an HTML5 application to run inside a thin container on the mobile device. A container such as PhoneGap (Cordova) provides access to native device features via a JavaScript bridge. While hybrid applications are developed on a common framework, they still require a version of the application to be built for each device, and have the same distribution and upgrade challenges as native applications.

As this is a book about Visualforce rather than mobile development, we will focus on HTML5 using the jQuery Mobile framework to provide the user interface. jQuery Mobile is a cross-device, touch-optimized, mobile UI framework built on jQuery. It uses progressive enhancement to maximize device support, starting with regular HTML for older devices, then applying CSS and JavaScript for devices that support those technologies.

For more information on jQuery Mobile, visit `http://jquerymobile.com/`.

In this chapter, we will use Visualforce in conjunction with jQuery Mobile to produce mobile pages that access Salesforce data, apply animated transitions between pages, and add a navigation bar. We will then move on to more advanced techniques, including creating a record in Salesforce and updating a record with a user's current location.

The jQuery Mobile JavaScript and CSS files are included from the Microsoft Ajax Content Delivery Network (`http://www.asp.net/ajaxlibrary/cdn.ashx`) rather than being uploaded as Salesforce static resources, as this makes it straightforward to move to new versions simply by changing the URL of the included file.

This does introduce a dependency on the Microsoft Ajax Content Delivery Network and in the event that the site was unavailable or access blocked, the jQuery Mobile styling and functionality would be lost.

Mobilizing a Visualforce page

Mobilizing a Visualforce page using jQuery Mobile requires that the jQuery Mobile stylesheets are used rather than the standard Salesforce stylesheets. This means that standard Visualforce components controlling layout, such as `<apex:pageBlock />` and `<apex:pageBlock />`, cannot be used. Instead, the jQuery Mobile specific styles must be used to layout and organize data.

In this recipe we will create a mobile Visualforce page that displays the top 10 opportunities by value. This page will use a jQuery Mobile grid to lay out the information in two columns.

Getting ready

This recipe requires a custom controller, so this must be created before the Visualforce page.

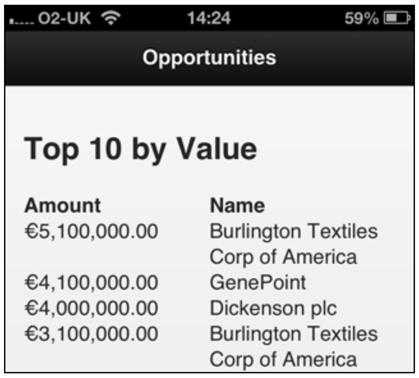
How to do it...

1. Navigate to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `MobileOppsController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the Visualforce page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `MobileOpps` in the **Label** field.
8. Accept the default **MobileOpps** that is automatically generated for the **Name** field.
9. Paste the contents of the `MobileOpps.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **MobileOpps** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the **MobileOpps** page: `https://<instance>/apex/MobileOpps`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Opportunities

Top 10 by Value

Amount	Name
€5,100,000.00	Burlington Textiles Corp of America
€4,100,000.00	GenePoint
€4,000,000.00	Dickenson plc
€3,100,000.00	Burlington Textiles Corp of America

Device browsers often assume that the user is viewing a desktop size page and size their display accordingly. Mobile Safari, for example, will assume a width of 980 px and zoom out to fit in all the content. To avoid this, the viewport is set through an HTML `meta` tag to match the dimensions of the device.

```
<meta name="viewport" content="initial-scale=1, maximum-scale=1,
      height=device-height, width=device-width" />
```

The `initial-scale` and `maximum-scale` tags ensure that the browser does not zoom out when initially rendering the page, and that the user cannot zoom the page in or out. It also allows the jQuery Mobile framework to scroll the page down by 60 pixels to hide the mobile Safari navigation bar.

A jQuery Mobile page is defined as `div` with a `data-role` attribute of `page`.

```
<div data-role="page" id="main">
```

HTML5 `data-` attributes are not standard attributes; they are a way to store additional data on an element that has meaning to the application. In earlier incarnations of HTML, the `class` attribute was often used to store this information.

Inside the page container, additional `<div>` elements are defined to specify the header bar (with a `data-role` attribute of `header`) and the main body of the page (with a `data-role` attribute of `content`).

```
<div data-role="header">
  <h1>Opportunities</h1>
</div> <!--  /header -->
<div data-role="content">
  <h2>Top 10 by Value</h2>
      ...
</div>
```

Further, the `<div>` elements generate the grid layout, while a standard Visualforce `<apex:repeat />` component iterates the opportunities from the controller and outputs them in individual grid cells.

```
<div class="ui-grid-a">
  <div class="ui-block-a"><strong>Amount</strong></div>
  <div class="ui-block-b"><strong>Name</strong></div>
  <apex:repeat value="{!opps}" var="opp">
    <div class="ui-block-a">
      <apex:outputField value="{!opp.Amount}"/>
    </div>
    <div class="ui-block-b">
      <apex:outputText value="{!opp.Name}"/>
    </div>
  </apex:repeat>
</div>
```

See also

- The *Adding a navigation bar* recipe in this chapter shows how to add a toolbar with navigation buttons to the footer of a mobile page.
- The *Listing records* recipe in this chapter recipe shows how to display a rich, filterable list of records on a mobile page.

Navigation and transitions

A Visualforce page leveraging jQuery Mobile can contain one or more application web pages. Each application page is demarcated by a `<div>` element with a `data-role` attribute of page, and additional application pages can be added to a single Visualforce page by stacking these elements.

When multiple application pages appear in a single Visualforce page, these are all stored in the Document Object Model (DOM) at load time and JavaScript is used to transition between the application pages. This can lead to faster application performance, as there is no round-trip to the server in order to access the next page, but does result in a larger DOM, so is not necessarily suitable for applications with many content heavy pages.

For more information about the Document Object Model, visit `http://en.wikipedia.org/wiki/Document_Object_Model`.

When a single Visualforce page contains a single application page, the default jQuery Mobile behavior is to load the new page into the DOM, use JavaScript to transition to the new page, and then to discard the previous page. This behavior can be overridden to force a full-page reload, which is useful when dealing with an application containing multiple Visualforce forms, as it ensures that any previous versions of the viewstate are discarded.

When jQuery Mobile navigates to an external page via Ajax, it only loads the content wrapped inside the first `<div>` element with a `data-role` attribute of page. Any additional inclusions or content/JavaScript outside of these elements will be discarded.

When page navigation makes use of Ajax, animated transitions may be applied to give more of a mobile application experience. The default transition is to fade out the previous page and fade in the new one, and there are a number of additional transition options.

In this recipe we will create a mobile Visualforce page that demonstrates each of these navigation types. This page will also contain links to demonstrate each of the transition types available.

How to do it...

This recipe does not require any controllers, so we only need to create the Visualforce pages.

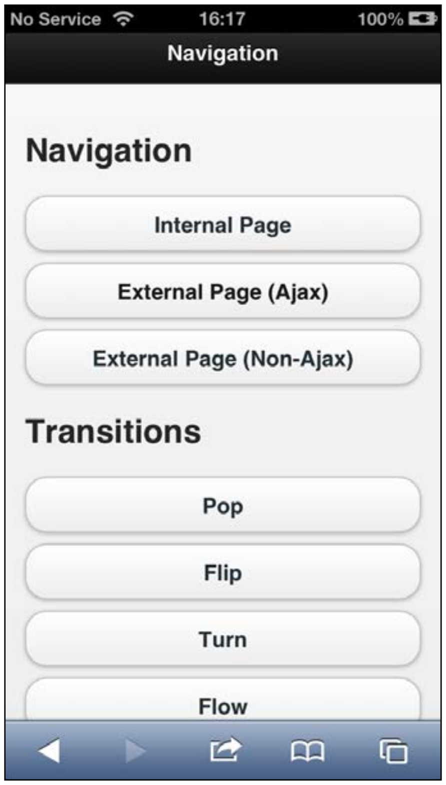
1. First, create the Visualforce page containing two application pages by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `MobileNavigation` in the **Label** field.
4. Accept the default **MobileNavigation** that is automatically generated for the **Name** field.
5. Paste the contents of the `MobileNavigation.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

7. Locate the entry for the **MobileNavigation** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.
9. Next, create the Visualforce page that is external but loaded via Ajax by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
10. Click on the **New** button.
11. Enter `MobileNavigation2` in the **Label** field.
12. Accept the default **MobileNavigation2** that is automatically generated for the **Name** field.
13. Paste the contents of the `MobileNavigation2.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
14. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Locate the entry for the **MobileNavigation2** page and click on the **Security** link.
16. On the resulting page, select which profiles should have access and click on the **Save** button.
17. Finally, create the Visualforce page that is external and accessed via a full page reload by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
18. Click on the **New** button.
19. Enter `MobileNavigation3` in the **Label** field.
20. Accept the default **MobileNavigation3** that is automatically generated for the **Name** field.
21. Paste the contents of the `MobileNavigation3.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
22. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
23. Locate the entry for the **MobileNavigation3** page and click on the **Security** link.
24. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the **MobileNavigation** page: `https://<instance>/apex/MobileNavigation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Each button is generated from an HTML anchor tag. The internal transition simply specifies the target anchor as the `href` attribute.

```
<a href="#internal" data-role="button>Internal Page</a>
```

The external transition via Ajax defines a target `href` attribute of the external page, and relies on the default jQuery Mobile behavior to provide Ajax navigation.

```
<a href="/apex/MobileNavigation2" data-role="button">
   External Page (Ajax)</a>
```

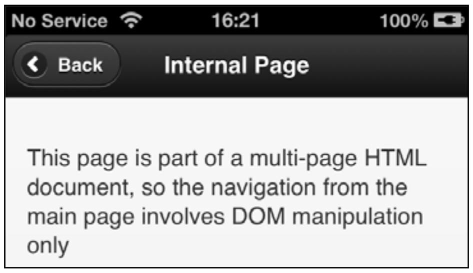
The external transition without Ajax defines a target `href` attribute of the external page, and turns off the Ajax navigation via the `data-ajax` attribute.

```
<a href="/apex/MobileNavigation3" data-ajax="false"
    data-role="button">External Page (Non-Ajax)</a>
```

The buttons in the **Transitions** section override the default transition behavior by specifying the desired transition through the `data-transition` attribute.

```
<a href="#internal" data-role="button" data-transition="pop">
  Pop</a>
<a href="#internal" data-role="button" data-transition="flip">
  Flip</a>
```

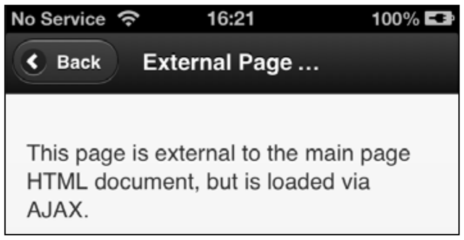
Clicking on the **Internal Page** link navigates to another application page inside the same HTML document using the default transition of fade in/out. As this is a JavaScript-only transition, there is no round-trip to the server.



This page is part of a multi-page HTML document, so the navigation from the main page involves DOM manipulation only

Clicking on the **Back** button returns to the main page via the same mechanism.

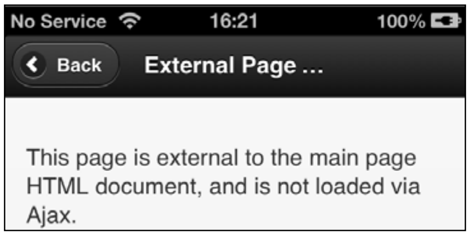
Clicking on the **External Page (Ajax)** button on the main page navigates to an external page via Ajax. A spinner is displayed to indicate that this is taking place, and the default transition of fade in/out is used.



This page is external to the main page HTML document, but is loaded via AJAX.

Clicking on the **Back** button returns the user to the main page via the same mechanism.

Clicking on the **External Page (Non-Ajax)** button on the main page navigates to an external page via a full-page reload. As this navigation does not take place via JavaScript, there is no animated transition or spinner image while the navigation takes place. Once the page is rendered, all HTML markup and JavaScript from previous pages has been discarded.



This page is external to the main page HTML document, and is not loaded via Ajax.

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains the how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Adding a navigation bar* recipe in this chapter shows how to add a toolbar with navigation buttons to the footer of a mobile page.

Adding a navigation bar

As HTML5 applications may either hide the mobile browser controls or navigate via JavaScript manipulation of the DOM, a different mechanism of navigating between pages must be used. jQuery Mobile provides a **navbar** widget that may be placed in the header, footer, or body of a page. This widget contains buttons to support navigation to other pages, and may be up to five buttons wide, after which it will wrap onto the next line.

In this recipe we will create mobile Visualforce pages for the **Home** and **About** elements of an application with a common navigation bar in the footer of the page. In order to avoid repetition of common content, we will use a template to generate the header and footer information, allowing each page to inject its content into appropriate areas of the page. The navigation bar will highlight the button for the current page. Finally, we will provide an additional button in the navigation bar to open a Facebook-style panel.

How to do it...

This recipe does not require any controllers, so we only need to create the Visualforce pages.

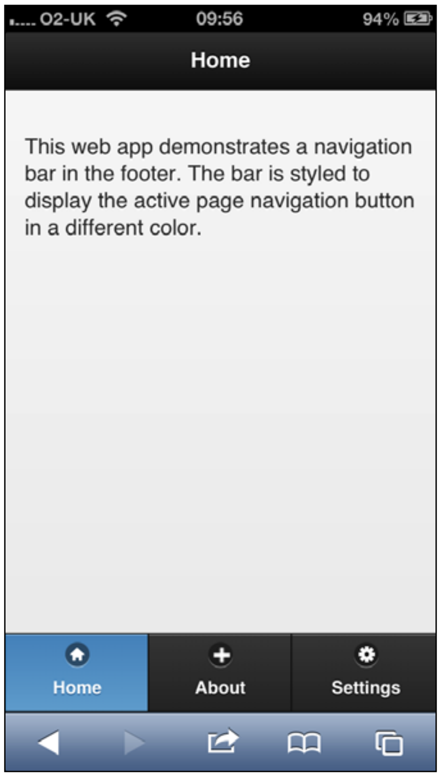
1. First, create the template for the pages by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.

3. Enter `MobileNavBarTemplate` in the **Label** field.
4. Accept the default **MobileNavBarTemplate** that is automatically generated for the **Name** field.
5. Paste the contents of the `MobileNavBarTemplate.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **MobileNavBarTemplate** page and click the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.
9. Next, create the Home page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
10. Click on the **New** button.
11. Enter `MobileNavBarHome` in the **Label** field.
12. Accept the default **MobileNavBarHome** that is automatically generated for the **Name** field.
13. Paste the contents of the `MobileNavBarHome.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
14. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Locate the entry for the **MobileNavBarHome** page and click on the **Security** link.
16. On the resulting page, select which profiles should have access and click on the **Save** button.
17. Finally, create the About page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
18. Click on the **New** button.
19. Enter `MobileNavBarAbout` in the **Label** field.
20. Accept the default **MobileNavBarAbout** that is automatically generated for the **Name** field.
21. Paste the contents of the `MobileNavBarAbout.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
22. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
23. Locate the entry for the **MobileNavBarAbout** page and click on the **Security** link.
24. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the **MobileNavBarHome** page: `https://<instance>/apex/MobileNavBarHome`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



This web app demonstrates a navigation bar in the footer. The bar is styled to display the active page navigation button in a different color.

The template page defines the header and allows the page to inject the text to be displayed.

```
<div data-role="header">
  <h1><apex:insert name="title"/></h1>
</div> <!--  /header -->
```

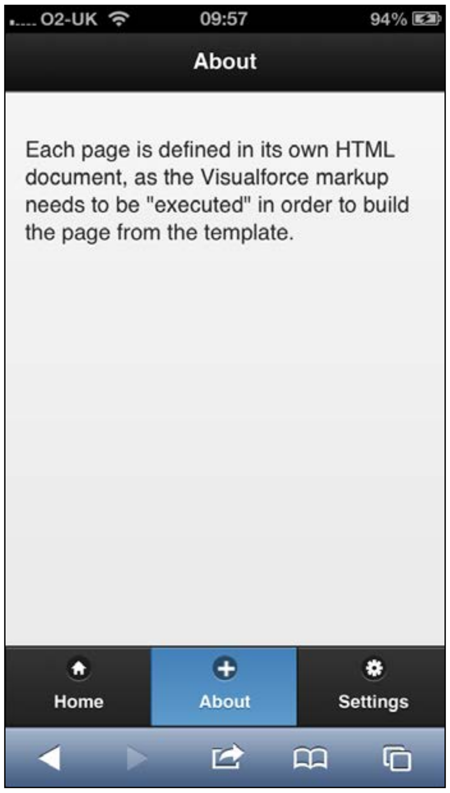
The navbar widget is defined as a `<div>` element with a `data-role` attribute of `navbar` in the footer, and relies on the page utilizing the template setting its name to a variable to highlight the active page. Each button in the navbar is generated from an anchor element inside a list item with the image displayed in the button specified by the `data-icon` attribute.

```
<div data-role="footer" data-position="fixed">
  <apex:variable var="page" value="home" />
```

```
    <apex:insert name="page" />
    <div data-role="navbar">
      <ul>
        <li><a href="/apex/MobileNavBarHome" data-ajax="false"
         data-icon="home" class="{!IF(page=='home', 'ui-btn-active',
                                     '')}">Home</a>
        </li>
           ...
      </ul>
    </div><!-- /navbar -->
  </div><!-- /footer -->
```

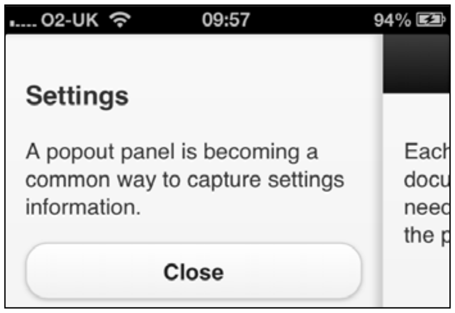
Note that the anchor tag specifies that Ajax navigation between pages should not be used via the `data-ajax="false"` attribute. This is to ensure that the browser interprets the entire page, rather than just the `<div>` element with a role of `page`, as detailed in the *Navigation and transitions* recipe.

Clicking on the **About** button navigates to the **About** page and updates the active navbar button.



Each page is defined in its own HTML document, as the Visualforce markup needs to be "executed" in order to build the page from the template.

Clicking on the **Settings** button pops out the Facebook-style panel, pushing the existing content off to the right.



Settings

A popout panel is becoming a common way to capture settings information.

Each
docu
need
the p

The panel is defined in the template as a sibling `<div>` element to the footer with a `data-role` attribute of `panel`. The close button is an anchor element with an `href` target of the `<div>` element for the page to return to.

```
<div data-role="panel" id="settingspanel">
  <h3>Settings</h3>
  <p>A popout panel is becoming a common way to capture settings
information.</p>
  <a href="#main" data-role="button" data-rel="close">Close</a>
</div><!-- /panel -->
```

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Navigation and transitions* recipe in this chapter explains the various types of page navigation available in jQuery Mobile and the animated transitions between pages.

Working with dialogs

While mobile applications can utilize the standard JavaScript `alert` function to create a dialog, this can be a jarring user experience, as the dialog will not be styled according to the application and the pop-up aspect may be at odds with the page transitions used elsewhere. jQuery Mobile provides the following two mechanisms for generating a dialog:

- A page that has a `<div>` element with a `data-role` attribute of `dialog` will only ever be rendered as a dialog and thus, is not as re-usable as the second method explained next.

- A link to a page with a `data-rel` attribute of `dialog` renders the target page as a dialog. This allows the target page to be rendered as a regular page or a dialog depending on the use case.

In this recipe we will create a Visualforce mobile page that displays a list of account names and a **View** button. Clicking on the **View** button will open an external page as a dialog and display details of the selected account.

How to do it...

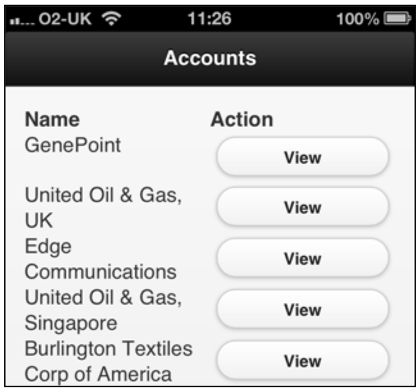
This recipe does not require any controllers, so we only need to create the Visualforce pages.

1. First, create the account list page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `MobileDialogMain` in the **Label** field.
4. Accept the default **MobileDialogMain** that is automatically generated for the **Name** field.
5. Paste the contents of the `MobileDialogMain.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **MobileDialogMain** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.
9. Next, create the dialog page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
10. Click on the **New** button.
11. Enter `MobileDialog` in the **Label** field.
12. Accept the default **MobileDialog** that is automatically generated for the **Name** field.
13. Paste the contents of the `MobileDialog.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
14. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Locate the entry for the **MobileDialog** page and click on the **Security** link.
16. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the **MobileDialogMain** page: `https://<instance>/apex/MobileDialogMain`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



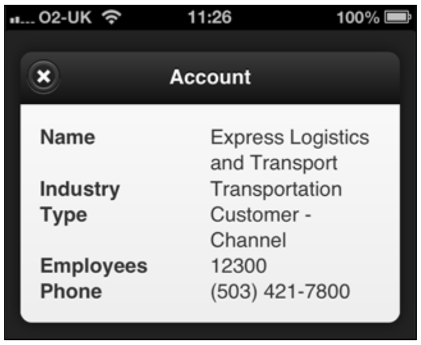
Name	Action
GenePoint	View
United Oil & Gas, UK	View
Edge Communications	View
United Oil & Gas, Singapore	View
Burlington Textiles Corp of America	View

A standard `<apex:repeat />` Visualforce component iterates the accounts from the controller and outputs the name and a button to view the detail.

```
<a href="/apex/MobileDialog?id={!acc.id}" data-mini="true"
   data-role="button" data-rel="dialog" data-transition="flip">
  View
</a>
```

The `data-rel` attribute of `dialog` specifies that the target page should be opened as a dialog, and the `data-transition` attribute specifies that the desired animated transition between the pages.

Clicking on the **View** button for any of the accounts displays the details in a dialog.



Name	Express Logistics and Transport
Industry	Transportation
Type	Customer - Channel
Employees	12300
Phone	(503) 421-7800

The detail page has no markup to indicate that it should be displayed as a dialog, and thus, could be used to display the account details as a regular jQuery Mobile page.

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Navigation and transitions* recipe in this chapter explains the various types of page navigation available in jQuery Mobile and the animated transitions between pages.

Listing records

In the previous recipes, records have been laid out in a grid format which do not have much space for record details. Using jQuery Mobile **listview** widget allows record details to be displayed in each element of a list, and also allows a number of different items of functionality to be made available declaratively through HTML5 `data-` attributes.

In this recipe we will create a mobile Visualforce page to render contact information in a jQuery Mobile listview widget. The listview will group the contacts into sections based on the first letter of the `LastName` field on the contact record. The listview will also be filterable by entering text into a search box at the top of the list.

How to do it...

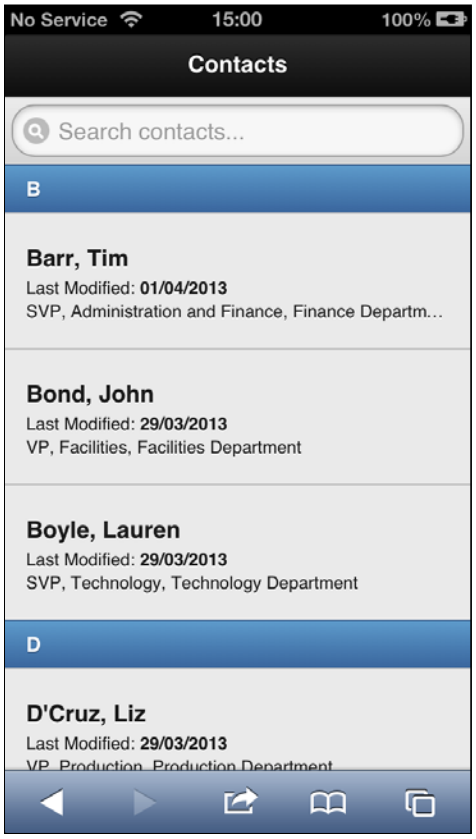
This recipe does not require any controllers, so we only need to create the Visualforce pages.

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `MobileFilteredList` in the **Label** field.
4. Accept the default **MobileFilteredList** that is automatically generated for the **Name** field.
5. Paste the contents of the `MobileFilteredList.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **MobileFilteredList** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the **MobileFilteredList** page: `https://<instance>/apex/MobileFilteredList`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



Barr, Tim
Last Modified: **01/04/2013**
SVP, Administration and Finance, Finance Departm...

Bond, John
Last Modified: **29/03/2013**
VP, Facilities, Facilities Department

Boyle, Lauren
Last Modified: **29/03/2013**
SVP, Technology, Technology Department

D'Cruz, Liz
Last Modified: **29/03/2013**
VP, Production, Production Department

The listview widget is generated from an unordered list element with the `data-role` attribute of `listview`.

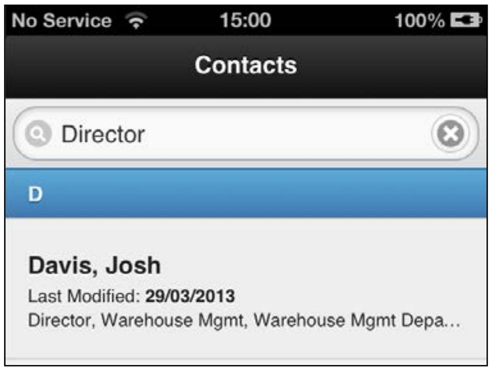
```
<ul data-filter="true" data-autodividers="true"
    data-filter-placeholder="Search contacts..."
    data-role="listview" data-theme="c" data-divider-theme="b">
   ...
</ul>
```

Setting the `data-autodividers` attribute to `true` groups the contacts by the first text item in each element; in this case, the `LastName` field from the contact record. The `data-filter` attribute set to `true` makes the listview searchable and adds the search box. Finally, the `data-filter-placeholder` attribute defines the placeholder text that the search box will initially be populated with.

The contacts from the controller are iterated by a standard `<apex:repeat />` component to generate a list item for each record.

```
<apex:repeat value="{!contacts}" var="cont">
  <li>
    <h2>{!cont.LastName}, {!cont.FirstName}</h2>
          ...
  </li>
</apex:repeat>
```

Entering text into the search box restricts the listview to those elements with matching text.



Davis, Josh
Last Modified: **29/03/2013**
Director, Warehouse Mgmt, Warehouse Mgmt Depa...

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Working with dialogs* recipe in this chapter shows how to open a detail dialog page from a list of records.

Mobile Visualforce forms

Capturing data from a mobile site can be achieved in a number of ways. The most straightforward, from a markup and code perspective, is to use a standard controller to manage the page and capture the information via a standard `<apex:form />` component. This mechanism requires the Visualforce viewstate to be used to maintain the state between the controller and the page, which is somewhat heavyweight for a mobile device and precludes use of Visualforce Ajax functionality, as this would interfere with the jQuery Mobile Ajax page navigation.

The other option is to use JavaScript to send the information back to Salesforce, either via the **REST API** or **JavaScript Remoting**. Using the REST API makes an application more portable, allowing it to be easily hosted outside the Salesforce platform, but does consume API calls and can lead to limits being exhausted. JavaScript Remoting allows methods in an Apex controller to be called from a Visualforce page via and does not consume API calls. It does, however, tie an application to Visualforce, which means that it can only be hosted on the Salesforce platform.

In this recipe we will create a mobile Visualforce page to capture lead information and store this in the Salesforce database via JavaScript Remoting.

Getting ready

This recipe makes use of a custom controller, so this must be created before the Visualforce page.

How to do it...

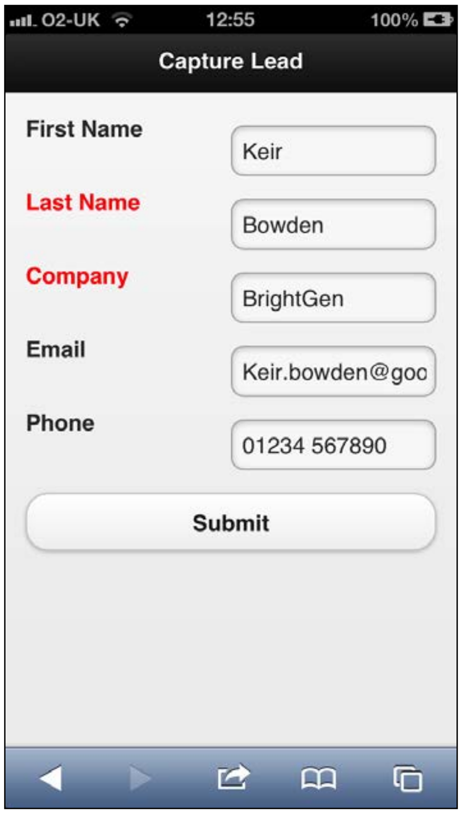
1. First, create the custom controller for the page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `MobileLeadCaptureController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the page that will capture the lead by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `MobileLeadCapture` in the **Label** field.
8. Accept the default **MobileLeadCapture** that is automatically generated for the **Name** field.
9. Paste the contents of the `MobileLeadCapture.page` file from the code download into the Visualforce Markup area and click on the **Save** button.

10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **MobileLeadCapture** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

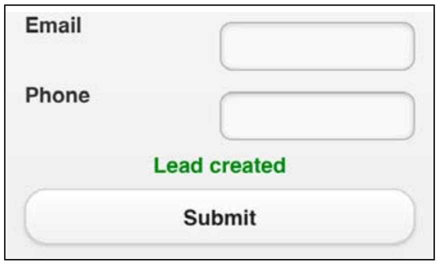
Opening the following URL in your mobile device browser displays the **MobileLeadCapture** page: `https://<instance>/apex/MobileLeadCapture`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.



First Name	Keir
Last Name	Bowden
Company	BrightGen
Email	Keir.bowden@goo
Phone	01234 567890

Filling out the form details and clicking on the **Submit** button saves the lead to the Salesforce database, clears the form, and displays a confirmation message.



In the event that an error has occurred or a mandatory field (with a red label) was not populated, an error message will be displayed in place of the confirmation message.

To make a controller method available to JavaScript Remoting, it must be decorated with the `@RemoteAction` annotation.

```
@RemoteAction
public static String CreateLead(String fName, String lName,
              String inCompany, String inEmail, String inPhone)
{
    ...
}
```

This method is invoked from the page using JavaScript after retrieving the lead details from the input elements.

```
var fname=$('#firstname').val();
var lname=$('#lastname').val();
...
MobileLeadCaptureController.CreateLead(fname, lname, company,
                email, phone, leadCaptured, {escape:true});
```

As JavaScript Remoting calls are asynchronous, a callback function must be provided to handle the controller response, in this case, it is the `leadCaptured` function, which checks the response from the controller and generates the appropriate message for the user.

```
function leadCaptured(result, event)
{
  ...
  if (event.status)
```

```
    {
      if ('SUCCESS'==result)
      {
        ...
        $('#msg').html('<span style="color:green;">Lead created</
span>');
    }
    else
      {
         $("#msg").html('<span style="color:red">An error occurred : ' +
result + '</span>');
      }
    }
    else if (event.type === 'exception')
    {
       $("#msg").html(event.message);
    }
    ...
}
```

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Storing the user's location* recipe in this chapter shows how to capture coordinates of the user's current location and store those in a record.

Redirecting to the mobile page based on the browser

While developers can provide mobile versions of a web page or site, users only access these if they are aware of the URL. When a user receives a link to a full site page in an e-mail or social media post, they are likely to follow this link rather than attempting to alter it to point to the mobile version. An application can improve the user experience by detecting that a mobile device is in use and sending the user to a mobile version of the page.

There are two ways in which this functionality can be provided in Visualforce. They are as follows:

- **Server side** by interrogating the `USER-AGENT` header of the request and matching this against a set of mobile devices. While this provides a faster experience for the user, as a server-side redirect can be used to send them to the destination page, it does introduce a maintenance overhead, as the list of mobile devices is ever increasing.
- **Client side** using JavaScript to determine the dimensions of the user's device and sending them to a different page if the width of the device is below a certain size. While this may involve two server round-trips: one for the full version of the site and another for the mobile version, it is future-proof in such a way that it is device-agnostic and purely concerned with the amount of screen estate available to display the page.

In this recipe we will create a Visualforce page that allows a user to view details of a contact. When the page is initially loaded, it will check the width of the user's device. If the width is less than or equal to 650 pixels, the user will be sent to a mobile page that renders the details using jQuery Mobile. If the width is greater than 650 pixels, the user will be redirected to a desktop version of the page that renders the details using standard Visualforce components.

Getting ready

This recipe does not require any controllers, so we only need to create the Visualforce pages.

How to do it...

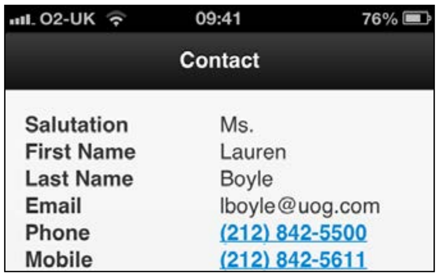
1. First create the desktop version of the page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `DesktopViewContact` in the **Label** field.
4. Accept the default **DesktopViewContact** that is automatically generated for the **Name** field.
5. Paste the contents of the `DesktopViewContact.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.

7. Locate the entry for the **DesktopViewContact** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.
9. Next, create the mobile version of the page by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
10. Click on the **New** button.
11. Enter `MobileViewContact` in the **Label** field.
12. Accept the default **MobileViewContact** that is automatically generated for the **Name** field.
13. Paste the contents of the `MobileViewContact.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
14. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
15. Locate the entry for the **MobileViewContact** page and click on the **Security** link.
16. On the resulting page, select which profiles should have access and click on the **Save** button.
17. Finally, create the page that will determine where the user should be redirected to by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
18. Click on the **New** button.
19. Enter `ViewContact` in the **Label** field.
20. Accept the default **ViewContact** that is automatically generated for the **Name** field.
21. Paste the contents of the `ViewContact.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
22. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
23. Locate the entry for the **ViewContact** page and click on the **Security** link.
24. On the resulting page, select which profiles should have access and click on the **Save** button.

How it works...

Opening the following URL in your mobile device browser displays the mobile version of the **ViewContact** page: `https://<instance>/apex/ViewContact?id<contact_id>`.

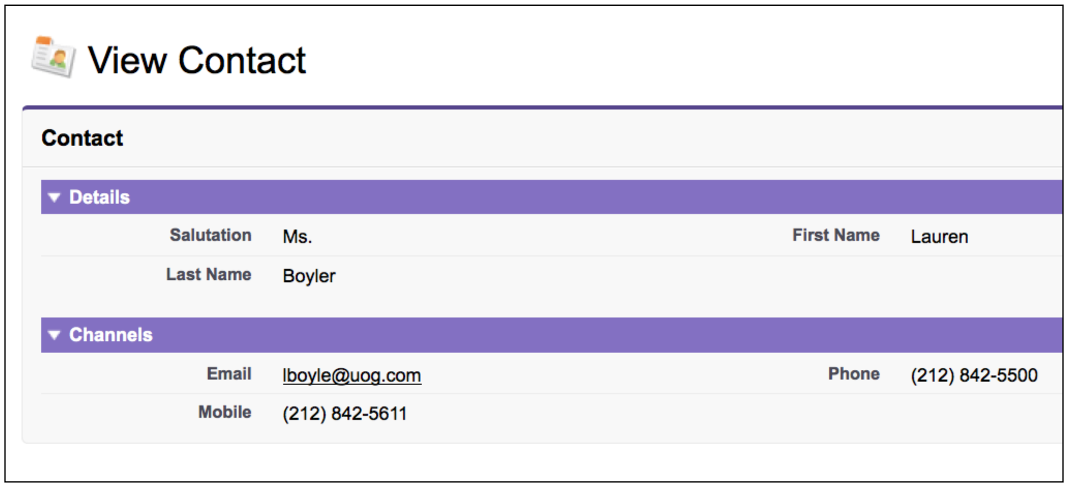
Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<contact_id>` is the record ID of a contact within your Salesforce instance.



Contact

Salutation	Ms.
First Name	Lauren
Last Name	Boyle
Email	lboyle@uog.com
Phone	(212) 842-5500
Mobile	(212) 842-5611

While accessing the same page in your desktop browser displays the desktop version of the page.



View Contact

Contact

Details

Salutation	Ms.	First Name	Lauren
Last Name	Boyler		

Channels

Email	lboyle@uog.com	Phone	(212) 842-5500
Mobile	(212) 842-5611		

Device browsers often assume that the user is viewing a desktop-size page and size their display accordingly, which would result in the user being redirected to the desktop version of the page. To avoid this, the viewport is set through an HTML `meta` tag to match the dimensions of the device.

```
<meta name="viewport" content="initial-scale=1, maximum-scale=1,
      height=device-height, width=device-width" />
```

A JavaScript fragment extracts the device width and redirects the user to the appropriate page.

```
<script>
  if ($(window).width()<=650)
  {
    window.location='/apex/MobileViewContact?id={!contact.id}';
  }
  else
  {
    window.location='/apex/DesktopViewContact?id={!contact.id}';
  }
</script>
```

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Mobile Visualforce forms* recipe in this chapter shows how to capture a lead on a mobile device and store this in the Salesforce database.

Storing the user's location

Mobile devices, by their very nature, are used on the move and applications often need to capture the location of the user in order to provide the best user experience; for example, showing proximity to a business or services, or allowing them to check-in at a destination.

In this recipe we will create a mobile Visualforce page to capture a lead and the location of the user. The location will be stored on the lead record in the Salesforce database.

Getting ready

This recipe requires a custom field on the lead sObject to capture the location.

1. Navigate to the lead fields setup page by clicking on **Your Name** | **Setup** | **App Setup** | **Customize** | **Lead** | **Fields**.
2. Scroll down to the **Lead Custom Fields and Relationships** section and click on the **New** button.
3. On the next page, **Step 1. Choose the field type**, select the **Gelocation** option from the **Data Type** radio buttons and click on the **Next** button.
4. On the next page, **Step 2. Enter the Details**, enter `Location` in the **Label** field, enter `10` in the **Decimal Places** field, accept the default name of **Location**, and click on the **Next** button.

5. On the next page, **Step 3. Establish field-level security for reference field**, leave all the fields at their default values and click on the **Next** button.
6. On the next page, **Step 4. Add to page layouts**, leave all the fields at their default values and click on the **Save** button.

> At the time of writing, in the Summer '13 release of Salesforce, the `Geolocation` field type is in **beta.** This means that it is a production quality feature with known limitations.

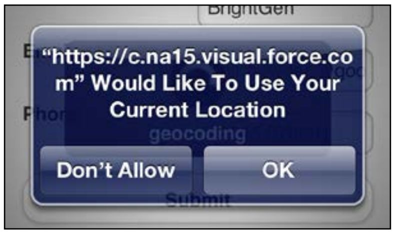
How to do it...

1. First, create the custom controller for the page by navigating to the Apex Classes setup page by clicking on **Your Name** | **Setup** | **Develop** | **Apex Classes**.
2. Click on the **New** button.
3. Paste the contents of the `MobileLeadLocationCaptureController.cls` Apex class from the code download into the Apex Class area.
4. Click on the **Save** button.
5. Next, create the page that will capture the lead by navigating to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
6. Click on the **New** button.
7. Enter `MobileLeadLocationCapture` in the **Label** field.
8. Accept the default **MobileLeadLocationCapture** that is automatically generated for the **Name** field.
9. Paste the contents of the `MobileLeadLocationCapture.page` file from the code download into the Visualforce Markup area and click on the Save button.
10. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
11. Locate the entry for the **MobileLeadLocationCapture** page and click on the **Security** link.
12. On the resulting page, select which profiles should have access and click on the **Save** button.

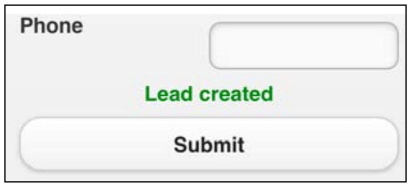
How it works...

Opening the following URL in your mobile device browser displays the **MobileLeadLocation** page: `https://<instance>/apex/MobileLeadLocation`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`.


"https://c.na15.visual.force.com" Would Like To Use Your Current Location
Don't Allow
OK

Filling out the form details and clicking on the **Submit** button executes JavaScript to retrieve the user's location after requesting permission. The lead is then saved to the Salesforce database, the form is cleared, and a confirmation message is displayed.



In the event that an error has occurred, a mandatory field (with a red label) was not populated, or retrieving the user's location has failed, an error message will be displayed in place of the confirmation message.

The controller method is made available for execution in JavaScript via the `@RemoteAction` annotation.

```
@RemoteAction
public static String CreateLead(String fName, String lName, String
inCompany, String inEmail, String inPhone,
           Double latitude, Double longitude)
{
  Lead newLead=new Lead(FirstName=fName,
                        LastName=lName,
                        Company=inCompany,
                        Email=inEmail,
                        Phone=inPhone);

  if (null!=latitude)
  {
    newLead.Location__Latitude__s=latitude;
    newLead.Location__Longitude__s=longitude;
  }
  ...
}
```

When the user clicks on the **Submit** button, the page uses the built-in geolocation functionality to attempt to determine the location, if available.

```
if (navigator.geolocation)
{
  navigator.geolocation.getCurrentPosition(
                  geoSuccess,
                  geoError,
                  {
                      maximumAge: 0,
                      timeout:30000,
                      enableHighAccuracy: true
                  }
              );
}
```

As the `getCurrentPosition` function is asynchronous, callback functions must be provided to handle error (`geoError`) and success (`geoSuccess`) results.

The `geoSuccess` function simply delegates to a common function process the lead, passing the location as coordinate parameters.

```
function geoSuccess(position)
{
  uploadLead(position.coords.latitude,
  position.coords.longitude);
}
```

The `uploadLead` function executes the controller method to store the lead record in the Salesforce database, which is also asynchronous and again requires a callback method to be specified.

```
function uploadLead(lat, long)
{
  var fname=$('#firstname').val();
  var lname=$('#lastname').val();
     ...
  MobileLeadLocationCaptureController.CreateLead(fname, lname,
        company, email, phone, lat, long,
        leadCaptured, {escape:true});
}
```

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Mobile Visualforce forms* recipe in this chapter introduces JavaScript Remoting and compares this with other mechanisms of saving records to the Salesforce database.

Scanning the QR code to access the page

Mobile devices, especially phones, have small keyboards and screens which can make entering page URLs difficult. If the device has a camera, scanning a code to navigate to a page can improve the user experience.

QR, or **Quick Response**, codes are 2-dimensional barcodes originally used to track automobiles during manufacture. For more information on **QR** codes see `http://en.wikipedia.org/wiki/QR_code`.

In this recipe we will create a mobile view page to display details of an opportunity. We will also create a Visualforce page to generate a **QR** code that links to this mobile view page, which we will then embed into the opportunity view page. When the code is scanned on a mobile device, this navigates a user to the mobile view page to allow them to view the opportunity details.

Getting ready

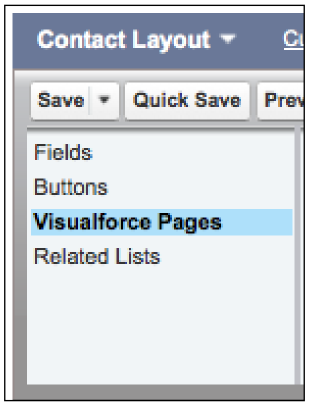
This recipe relies on the mobile device being able to scan the **QR code**.

This recipe also uses the **jquery.qrcode.js** (`https://github.com/jeromeetienne/jquery-qrcode`) JavaScript library to generate the QR code. As this is not available from a content delivery network, it must be present as a static resource:

1. Download the `jquery.qrcode.js` ZIP file by navigating to the GitHub page `https://github.com/jeromeetienne/jquery-qrcode /` and clicking on the **Download Zip** button at the bottom of the right-hand sidebar.
2. Navigate to the Static Resource setup page by clicking on **Your Name** | **Setup** | **Develop** | **Static Resources**.
3. Click on the **New** button.
4. Enter `QRCode` in the **Name** field.
5. Enter `QR code generator` in the **Description** field.
6. Click on the **Browse** button and select the `jquery.qrcode-<version>.zip` file downloaded in step 1.
7. Accept the default **Private** value for the **Cache Control** field and click on the **Save** button.

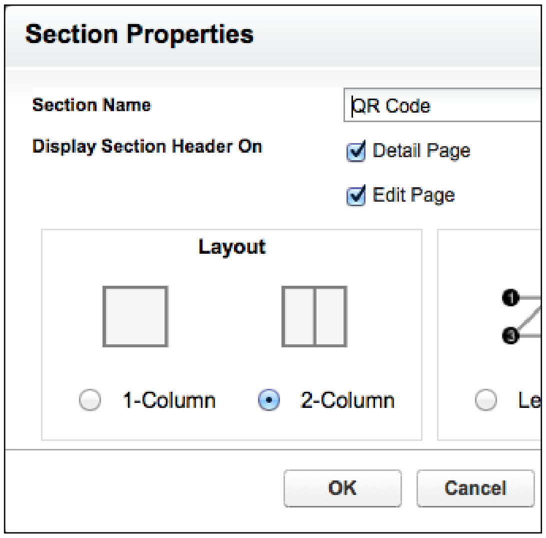
This recipe also requires Visualforce page to generate a QR code that is embedded into the opportunity record view page.

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `OppQRCode` in the **Label** field.
4. Accept the default **OppQRCode** that is automatically generated for the **Name** field.
5. Paste the contents of the `OppQRCode.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **OppQRCode** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.
9. Finally, add the page to the standard opportunity page layout. Navigate to the contact Page Layouts page by clicking on **Your Name** | **Setup** | **Customize** | **Opportunity** | **Page Layouts**.
10. Locate the first page layout to add the page to and click on the **Edit** link in the **Action** column.
11. On the resulting layout editor page, click on the **Visualforce Pages** link in the left-hand column of the palette.



12. Drag the **+Section** option from the right-hand side of the palette and drop this beneath the bottom section of the page.

13. In the **Section Properties** popup, set **Section Name** to `QR Code`, select the **1-column** radio button in the **Layout** section, and click on the **OK** button.



Section Properties

Section Name	QR Code
Display Section Header On	✓ Detail Page
	✓ Edit Page

14. Drag the **OppQRCode** page from the right-hand side of the palette and drop this beneath the **QR Code** section.
15. Click on the **Save** button to commit the page layout changes.
16. Repeat steps 10 to 15 to add the Visualforce page to additional page layouts as required.

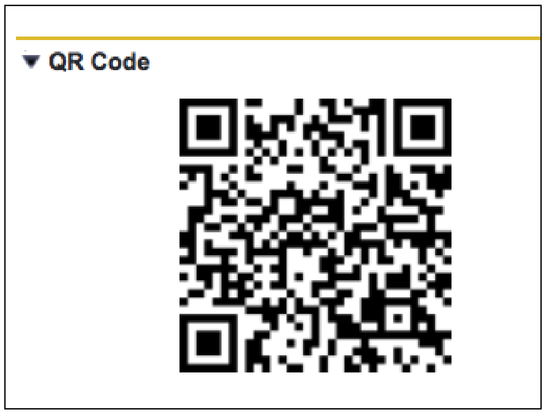
How to do it...

1. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
2. Click on the **New** button.
3. Enter `MobileOpp` in the **Label** field.
4. Accept the default **MobileOpp** that is automatically generated for the **Name** field.
5. Paste the contents of the `MobileOpp.page` file from the code download into the Visualforce Markup area and click on the **Save** button.
6. Navigate to the Visualforce setup page by clicking on **Your Name** | **Setup** | **Develop** | **Pages**.
7. Locate the entry for the **MobileOpp** page and click on the **Security** link.
8. On the resulting page, select which profiles should have access and click on the **Save** button.

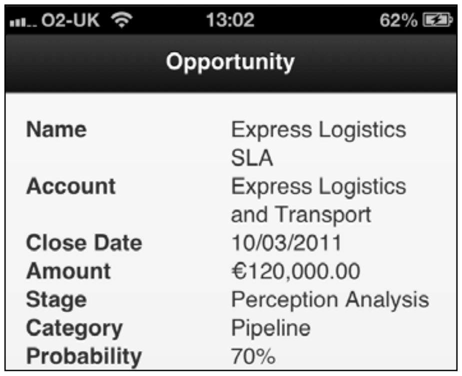
How it works...

Opening the following URL in your desktop browser displays the details of an opportunity, including the QR code: `https://<instance>/<opportunity_id>`.

Here, `<instance>` is the Salesforce instance specific to your organization, for example, `na6.salesforce.com`, and `<opportunity_id>` is the record ID of an opportunity within your Salesforce instance.



Scanning the QR code, with an application such as Semacode for the iPhone, opens the device browser and navigates to the **MobileOpp** page, passing the record ID as a parameter on the URL.



Opportunity

Name	Express Logistics SLA
Account	Express Logistics and Transport
Close Date	10/03/2011
Amount	€120,000.00
Stage	Perception Analysis
Category	Pipeline
Probability	70%

The **OppQRCode** page defines a `<div>` element to house the QR code.

```
<div id="qrcodeTable" />
```

This element is then populated via a JavaScript code.

```
<script>
  $('#qrcodeTable').qrcode({
                     render    : "canvas",
                     width     : 150,
                     height    : 150,
                     text      :
       "{!LEFT($Api.Partner_Server_URL_260, FIND( '/services',
  $Api.Partner_Server_URL_260)) +
      'apex/MobileOpp?id=' + Opportunity.Id}"
                          });
</script>
```

The `LEFT` function determines the base Salesforce URL by processing the partner web services API endpoint for your Salesforce instance.

See also

- The *Mobilizing a Visualforce page* recipe in this chapter explains how a Visualforce page utilizing jQuery Mobile is constructed.
- The *Mobile Visualforce forms* recipe in this chapter shows how to capture a lead on a mobile device and store this in the Salesforce database.

Index