Chapter 7

Protecting Your Data

In this chapter ...

- Force Platform Security
- Force Platform Security Framework
- Administrative Security
- Component-based Security
- Record-based Sharing
- Designing Security
- Summary

Data is the core of your information stack, and, more importantly, the repository of the business value of all of your systems. The value provided by your data justifies the very existence of your company's entire IT investment—including you!

You have to be sure that your data is safe and sound, protected from unauthorized access from outside your company, as well as safeguarded from inappropriate usage by your user community.

The Force Platform is built with security as the foundation for the entire platform. This foundation includes both protection for your data and applications, and the ability to implement your own security scheme, which must be able to flexibly reflect the structure and needs of your organization. The security features of the Force Platform provide both strength and flexibility.

In this chapter, you will learn how the platform itself is protected with built-in security guards and controls. More importantly for your task as a developer, you will come to understand how you can implement access limitations for your own data and applications to meet the specific requirements of your organization.

Important: If you have been following the examples in the previous chapter, your Force Platform organization should be ready for the exercises in this chapter. If you have not done the exercises, or want to start with a fresh version of the sample

Recruiting application, please refer to the instructions in the Code Share project for this book. You can find information on how to access that project in *Chapter 1: Welcome to the Force Platform*.

Force Platform Security

Justifiably or not, a multi-tenant, on-demand platform is the subject of more doubt in the area of security than an in-house platform. To guarantee the security of your own organization, the Force Platform includes a range of security defenses that are automatically used to guard your data resources.

Organization Security

One of the core features of a multi-tenant platform is the use of a single pool of computing resources to service the needs of many different customers. The Force Platform protects your organization from all other customer organizations by using a unique organization identifier that is associated with your Force Platform session.

Once you log into your Force Platform organization, your subsequent requests are associated with your organization, and only your organization, using this identifier. With this safeguard, all access to your organization is protected by the user authentication.

User Security

Users are identified to the Force Platform by their user name and password. You can specify the password policies through the **Setup ➤ Security Controls ➤ Password Policies** as shown below.

Figure 115: Password Policies

This page gives you the ability to set policies involving how frequently passwords must be changed, whether a user can re-use a recent password, password complexity requirements and how the platform treats invalid login attempts. These password domain settings are usually handled by administrators, but you can rest assured that your deployment environment can be secured with these settings.

Security settings and API access

The focus of this book is on applications running on the Force Platform, rather than applications accessing the platform through the Force Platform API. But you should be aware that some of the standard security features discussed in this section work somewhat differently for API access. For instance, if a user logs into an application running on the Force Platform and their password has expired, they are prompted to change the password. An expired password prevents access through the API, but applications that access the platform through the API have to detect this problem procedurally.

The session timeout discussed later in this section is sensitive to any user interaction through the standard Force Platform user interface, but the time limit on a session is enforced on an entire API session, regardless of user interaction. In addition, there is no warning given before an API session is terminated. Because of this, access through the Force Platform API should not assume the existence of an active session.

Administrators can force the expiration of passwords for one or more users through the **Setup ➤ Manager Users ➤ Users** page, or force the expiration of all passwords for all users with the Expire All Passwords option in the Security Control area of the Administrative Setup section of the Setup menu.

All users are assigned to a user profile, which is examined in more detail in the section on user-based permissions below.

User Authentication

The Force Platform has its own system of user authentication, but some companies would like to use an existing single sign-on capability to simplify and standardize their user security.

You have two options to implement single sign-on with the Force Platform: delegated authentication and Security Assertion Markup Language (SAML).

- **Delegated Authentication** - With this approach, a user logs into the Force Platform as usual, but the platform uses a web service callout to submit the user name and password to

an external authorization authority. Once that authority approves the logon, the approval is passed back to the Force Platform and the user can proceed. If you want to use delegated authentication, you will have to contact Salesforce.com to enable this feature for your organization and then create a Web Service callout to the authentication authority.

- **SAML** - Using SAML, your request goes to the SAML authority that validates your identity and returns a token. The token is passed to the Force Platform that verifies the user with the authority. This approach is typically used when your users are accessing your Force Platform applications through a portal, which would handle the initial authentication and avoid the need to log into the Force Platform environment again. You can configure SAML for your organization through the **Setup ➤ Security Controls ➤ Single Sign-On Page**.

Both of these single sign-on options are described in much more detail in the online documentation and in articles available at the developer.force.com site.

Network-based Security

To provide a level of network-based security, the Force Platform includes the ability to limit access to your organization based on the IP address of your client in two different ways.

You can use a whitelist to indicate the IP address ranges that are allowed to access an organization by default. You can define a whitelist for an entire organization clicking**Setup ➤ Security Controls ➤ Network Access ➤ New**, as shown in the following figure.

Figure 116: Network Access

You can define multiple IP ranges for your organization. You cannot use this feature to whitelist all IP addresses since there is a limit on the number of IP addresses that can be defined in a range; however, the limit is generous enough to allow for a broad range of specified IP addresses.

If you define a set of allowable IP addresses for your organization, all login attempts from those IP addresses are accepted by default. If a user tries to log in from an IP address that is not specified for the org, they are challenged by the Force Platform, which then sends them an

email. The user must click on a link in the email to allow access from the new IP address. Once this reply is sent, the user can log in from the current IP address in the future.

If no IP addresses are defined for the organization, a user must respond to this challenge the first time they log in from an IP address.

This challenge mechanism is fine if a user is actually attempting to log into the platform, but does not work if a user or application is trying to access the organization through the Web Services API, described in the next chapter. In this situation, a login request must include a security token appended to the password for the user. The user generates a security token through the **Setup ➤ My Personal Information ➤ Reset My Security Token** option that sends an email to the user with the security token. Some utilities, such as Data Loader, use the Web Services API to access the Force Platform, so use of these utilities from an unfamiliar IP address requires the use of a security token.

Profiles can also be used to define a range of acceptable IP addresses, although the IP addresses defined for a profile are restrictive, rather than acceptable defaults. If a profile has IP addresses defined, any user with that profile cannot log into the Force Platform from any other IP address.

Profiles are also used to restrict the hours that a user can log into the platform. You can set either of these restrictions for a particular profile from the bottom of the Profiles page under **Administrative Setup ➤ Manage Users**.

The limitations imposed on IP addresses are used to help protect against phishing attacks. A malicious attack cannot be triggered from outside your range of IP addresses, even if the attacker has a correct user name and password.

Session Security

The final area of security for the Force Platform revolves around an individual Force Platform session. The **Setup ➤ Security Controls ➤ Session Settings** page, shown below, allows you to require secure connections to the platform or to lock a session to the originating IP address.

Figure 117: Session Settings

You can also set the time limit for an individual session to between 15 minutes and 8 hours. Once a user's session is inactive for this length of time, the user has to respond to a warning popup. If the user does not respond, the session ends and the user has to log back into the Force Platform. You can also suppress the use of the warning popup.

The session timeout provides some protection from unauthorized access caused by leaving your computer while still logged into the Force Platform.

Auditing

Auditing features do not secure your Force Platform environment by themselves, but these features provide information about client usage of the platform that can be critical in diagnosing potential or real security issues. All objects include fields to store the name of the users who created the record and who last modified the record, providing some basic auditing information, but the Force Platform has features to extend the auditing capabilities of your application and data.

The **Setup ➤ Manage Users ➤ Login History** page displays the last 20 logins to your organization, as well as giving you the ability to download 6 months worth of logins in a comma separated variable file. This file includes session-specific attributes, such as IP address and browser type, which are not available in record and field auditing.

As mentioned earlier, you can turn on auditing for objects with a single click. Object-level auditing tracks changes in the overall object records, such as record creation. You can also enable auditing for individual fields, automatically tracking any changes in the values of selected fields. Although auditing is available for all custom objects, many standard objects do not allow auditing.

Under **Setup ➤ Security Controls**, administrators can use **View Setup Audit Trail** to monitor when meta-data definitions for objects have changed. With this feature, you can track the evolution of your application over time.

Force Platform Security Framework

The previous section of this chapter covered the built-in mechanisms used by the Force Platform to insure that your individual organization is protected from external access. The platform also has a framework you can use to offer different access permissions to authenticated users within your organization.

There are three tiers to the Force Platform security framework:

- Administrative permissions that grant overall security permissions to users or profiles
- Component-based permissions that control access to a range of components, including applications and objects
- Record-based sharing that limits access to individual records

The remainder of this chapter covers the use of each of these areas of the Force Platform personalized security scheme.

Administrative Security

The Force Platform includes a wide range of built-in capabilities. You usually do not want to give all the power of the complete platform to all users in the environment. Administrative permissions are used to grant or deny access to some areas of Force Platform functionality for particular users.

Profiles

Profiles are a way you can group users together for easier administration. A user can belong to one and only one profile, although administrators can change profile membership for a user.

You have already used profiles in previous chapters of this book – to assign access to objects, default record layouts and record types. Profiles are the basis for allowing administrative and component permissions.

Profiles are defined and edited through the **Setup ➤ Manage Users ➤ Profiles** page, which gives you access to detail pages for a profile by clicking on the profile name, as partially shown in the figure below.

Keyword	Keyword Layout [View Assignment]	Text Ad	Text Ad Layout [View Assignment]
Location	Location Layout [View Assignment]		

Field-Level Security

Standard Field-Level Security

Account	[View]	Lead	[View]
Asset	[View]	Opportunity	[View]
Campaign	[View]	Opportunity Product	[View]
Case	[View]	Product	[View]
Contact	[View]	Solution	[View]
Contract	[View]	Task	[View]
Event	[View]	User	[View]
Idea	[View]		

Custom Field-Level Security

Ad Group	[View]	Position	[View]
Candidate	[View]	Positions Tracker	[View]
Google Campaign	[View]	Recruiting Tracker	[View]
Interview	[View]	Search Phrase	[View]
Job Application	[View]	SFGA Version	[View]
Keyword	[View]	Text Ad	[View]
Location	[View]		

Custom App Settings

	Visible	Default		Visible	Default
Call Center	✓		Marketing	✓	
Google AdWords	✓		Recruiting	✓	

Figure 118: Profiles

This page allows you to assign page layouts, application and tab access and record type settings. Component-based permissions, described in the next section, can also be assigned from this page, along with various administrative and user permissions.

Your Force Platform environment comes with several predefined profiles. Administrators can create custom profiles to fit the needs of their organizations.

Administrative permissions

Administrative permissions allow users to manage higher levels of their Force Platform environment. The highest administrative permissions are granted when a user is assigned to the System Administrator profile, allowing access to all of the Administrative Setup choices.

Some of the other administrative permissions revolve around the use of Salesforce platform applications. The following table provides a summary of the relevant administrative permissions for Force Platform developers.

Table 8: Administrative permissions

Permission Type	Permission	Description
Security	Manage Users	Allows creation and modification of users for the organization, and access to profiles and sharing settings. This permission allows the owner to grant all other permissions to users, so should be assigned with care.
	Password never expires	Eliminates password policy requirements to expire passwords after a designated interval.
	API-enabled	Allows access to organizations through the Force Platform API. Without this permission, users cannot access the platform from outside of native applications.
	API-Only User	Only allows access through the Force Platform API.
	View Setup and Configuration	Gives users the ability to see the organization setup information, but not make any changes to this information.
Supporting Objects	Customize application	Allows user access to the complete Setup menu for Force Platform applications.
	Edit HTML Templates, Manage Letterheads, Manage Public Templates	Allows users to edit various components used by Force Platform components, such as messages sent from workflow and approval processes.
Reports	Manage Custom Report Types, Manage Dashboards,	Allows users to modify various components used in Force Platform reporting. A user

Permission Type	Permission	Description
	Manage Public Reports, Schedule Dashboards	with the Manage Custom Reports privilege can also create new folders for reports.
	Create and Customize Reports	Gives users the ability to create new reports or modify existing reports.
	Export Reports	Allows exporting of data from a report to Excel spreadsheets.
	Run Reports	Without this permission, the Reports tab is not available to the user.
Apex	Author Apex	Allows users to create Apex triggers and classes. Only available in editions that allow access to Apex code, and requires that the user also have the Modify All Data permission.
Data	Disable Outbound Messages	Prevents the use of outbound messages as a workflow activity.
	Edit Read-Only Fields	Overrides read-only limitations set in page layouts.
	Weekly Data Export	Allows users to run a weekly data export.
	View Encrypted Data	Allows users to see data in encrypted fields as plain text data. This feature is not turned on, by default—you can request the feature through salesforce.com.

Four permissions are extremely powerful, and deserve special discussion. The most powerful permission is `Manage Users`. When a user has this permission, they can grant any other

permission to themselves and other users. This permission makes a user into a super administrator who can grant any other permission.

The `Customize Application` permission grants a broad range of permissions that allow a user to control all aspects of an application from creating, editing, and deleting custom fields, to implementing workflow rules. Application developers need this permission, but you should be aware of the range of operations this permission grants. For a full description of these permissions, please refer to the online help.

The `View All Data` and `Modify All Data` permissions override any restrictions on data in any Force Platform objects. These privileges also circumvent the entire system of record-based sharing described later in this chapter. These permissions are granted to the System Administrator profile and can be granted to any custom profile.

Caution: `Modify All Data` grants full object-level privileges (create, read, update and delete) to a user who possesses this permission. A user with this permission also ignores sharing rules for data access. If the permission is revoked, the user will still have the full object-level privileges, but sharing rules will now be in effect.

`Modify All Data` is an extremely powerful permission. A user with this permission can not only edit all data, but also delete all data, and then empty the recycle bin to eliminate all traces of the data—certainly not something you would grant lightly. Any developer creating Apex code needs to have this permission.

Component-based Security

User-based permissions, granted to profiles, cut across the entire Force Platform environment. Profiles are also used to grant different levels of access to individual Force Platform components.

Once again, you have already seen that profiles can be used to define permissions when you create your application and custom objects. The following sections describe the different types of permissions granted to profiles for different types of components.

Application Permissions

You can grant access to an application by making the application visible, as shown in the drop-down list from the upper right corner of a Force Platform application, as shown below.

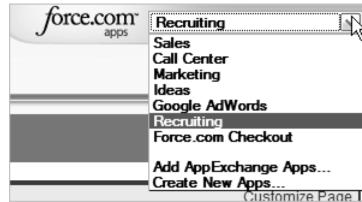


Figure 119: Application selection

If a user does not have access to an application, the name of the application does not appear in the picklist in the top right corner of the page. You can specify one application as the default application for a profile, causing the user to go to that application immediately after initially logging into the Force Platform. After the initial login, the default application becomes less important since the user's last application is retained the next time the user logs in.

Tab Permissions

Tab permissions allow you to show a tab by default with the setting of `Default On`. This setting also adds the object associated with the tab to the **Create New** picklist in the sidebar.

The `Default Off` setting suppresses the display of the tab in the tab set at the top of the page, but allows users to get to the tab with the right arrow at the right of the tab set, or to add the tab to their default display with the same **My Personal Information ➤ Change My Display ➤ Customize My Tabs** option.

If a tab is marked as Tab Hidden for a profile, users with that profile cannot access the tab. Preventing access to a tab eliminates access through the Force Platform tab set, but you must use object permissions, described later in this section, to circumscribe access to an object. Access can also be granted to a tab through the use of lookup fields and related lists that act as links to a record and its tab.

Voice of the Developer: Custom tabs are associated with an application, but restricting access to an application does not automatically restrict access to the tabs. If a user still has permission to access a tab, they can reach it through the additional tabs interface, even if the application which normally contains the tab is not available to them. This security implication flows directly from the fact that tabs are separate entities from applications. A user can add any tab they can access to any application which they can access—so security settings are likewise separate.

Record Type Permissions

You learned about record types in *Chapter 4: Expanding data options*. Record types are Force Platform features that allow you to assign different page layouts and picklist values to different profiles. If you have defined record types for an object, these types are listed in the Profile detail page.

Apex Class and Visualforce Page Permissions

In *Chapter 10: Apex* of this book, you will learn how to define Apex classes, which encapsulate procedural functionality defined with Apex code, and Visualforce pages.

You can give profiles permission to access individual Apex classes and Visualforce pages through the Profiles page.

Apex classes execute as the system user, so user permissions associated with profiles are not in effect for the execution of these classes. You can prevent users from accessing an Apex class to prevent users within a profile from using the functionality provided by that class.

Object Permissions

Object permissions are slightly more complex, since you can limit the type of action performed on an object through these permissions, rather than just allowing access or not.

You can edit object permissions for custom profiles from the detail page for a profile, as shown in Figure 7-6.

Standard Object Permissions

	Read	Create	Edit	Delete		Read	Create	Edit	Delete
Accounts	✓	✓	✓	✓	Ideas	✓	✓	✓	✓
Assets	✓	✓	✓	✓	Leads	✓	✓	✓	✓
Campaigns	✓	✓	✓	✓	Opportunities	✓	✓	✓	✓
Cases	✓	✓	✓	✓	Price Books	✓	✓	✓	✓
Contacts	✓	✓	✓	✓	Products	✓	✓	✓	✓
Contracts	✓	✓	✓	✓	Solutions	✓	✓	✓	✓
Documents	✓	✓	✓	✓					

Custom Object Permissions

	Read	Create	Edit	Delete		Read	Create	Edit	Delete
Ad Groups	✓	✓	✓	✓	Positions	✓	✓	✓	✓
Candidates	✓	✓	✓	✓	Postions Tracker Records	✓	✓	✓	✓
Google Campaigns	✓	✓	✓	✓	Recruiting Tracker Records	✓	✓	✓	✓
Interviews	✓	✓	✓	✓	Search Phrases	✓	✓	✓	✓
Job Applications	✓	✓	✓	✓	SFGA Version	✓	✓	✓	✓
Keywords	✓	✓	✓	✓	Text Ads	✓	✓	✓	✓
Locations	✓	✓	✓	✓					

Desktop Integration Clients

Figure 120: Object permissions in a profile

You can allow the following actions for any standard or custom object:

- `Read` allows read access, and is required for all other permissions. If this permission is removed, all other permissions are also removed
- `Create` and `Edit` permissions also grant `Read` permissions.
- `Delete` permission grants `Read` and `Edit` permissions.

If a user does not have at least Read access to an object, all associated components for that object, such as tabs and report types, are also inaccessible.

Field Level Security Permissions

You can also define permissions on individual object fields. A field can be limited to Read-Only access or hidden completely by removing the `Visible` permission.

You can access field-level security permissions from either the Profile page, to see and edit field level permissions for all fields in an object, or the Object page, to see and edit field level permissions for all profiles for a particular field.

Field-level security affects the display and access to fields through all standard Force Platform interfaces, including page layouts and reports. If a field is not visible to a user, the field display is suppressed on all page layouts for the user, and the field is not available for reports.

Record-based Sharing

The previous sections covered the standard security mechanisms that protect your Force Platform environment and the use of permissions to allow access to different pieces of Force Platform functionality, as well as the components you create. These permissions are fairly coarse—you have permission for a type of access to all the records in an object. The Force Platform gives you a way to implement different access to different data records stored in a single object. This type of security is based on individual rows of data, and implemented with a different set of tools and concepts. These tools and concepts are illustrated in the figure below, and the rest of this chapter is dedicated to explaining this chart in more detail.

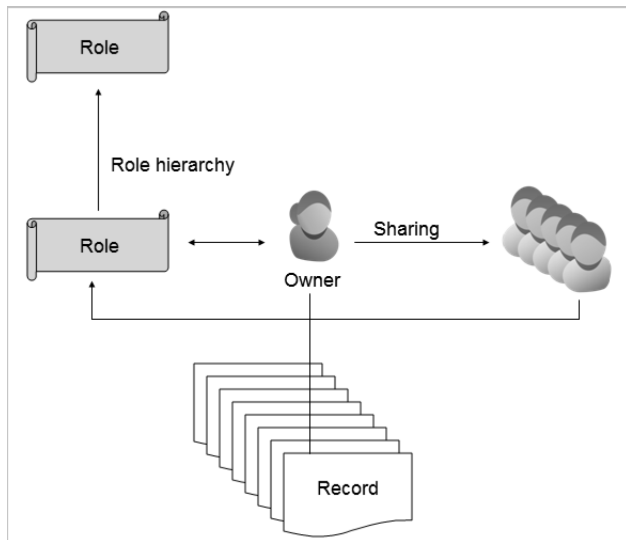


Figure 121: Record-based sharing

Object-based and record-based permissions are complementary, not alternate ways to implement access. Object-based permissions allow access to an object and all the records contained within the object. If a user has the permission to access data in an object, access permissions for individual records can be limited through the use of record-based sharing. If a user does not have access to an object, record-based sharing cannot grant access to the object.

Record Ownership

The core concept behind data-based permissions is record ownership. The owner of a record has all privileges for that record—the ability to delete the record, share the record with other users or even transfer ownership of the record.

The owner of a record created through the user interface is, by default, the user who created the record. A record owner always has read and write access to the record, as well as delete, transfer, and the ability to share the record with other users. A queue, which is a collection of users, can also be designated as the owner of a record. You can change the owner of a record, either through the standard Force Platform interface, as shown in the following figure, or with Apex code, the Force Platform procedural language that is the topic of Chapters 10 and 11.

Ownership Edit
POS-00014
Help for this Page
This screen allows you to transfer a position from one user to another.
Select New Owner
= Required Information
Transfer this position POS-00014
Owner
Send Notification Email
Save Cancel

Figure 122: Changing record ownership

A record can be owned by one or more users, or groups of users. Any owner of a record can transfer ownership of the record to another user by using the Change link next to the name of the owner in the detail page for the record. In addition, any user with the `Modify All Data` permission, discussed above, has full privileges for all data in all objects.

Tip: Later in this section you will learn about roles and the role hierarchy. The role hierarchy grants all record privileges for a user to all those who are above the user in the hierarchy, including ownership.

For the rest of this chapter, the term owner refers to anyone with ownership permissions, including users who have `Modify All Data` permission and users whose role is above the owner in the role hierarchy.

Organization-wide defaults

To properly implement record-based permissions, you have to follow a two-step process:

1. Lock down access to all records for an object based on the lowest level of permission that exists in your organization.
2. Open up access to particular records for particular users.

To lock down records, the Force Platform provides a concept known as organization-wide defaults, usually referred to as org-wide defaults. As the name implies, this specification defines the default access to all records in an object for all users.

There are three settings for org-wide defaults:

- `Public Read/Write` allows all users to read and write data to all the records in an object.

Caution: This org-wide default does not grant delete, transfer or share permissions. These are only available to owners of a record.

- `Public Read` allows all users to read all the records in an object.
- `Private` only allows the owner of the record, and users with the appropriate permissions, such as `Edit All Data` or `View All Data`, to view or edit a record in the object.

By default, all custom objects are created with an org-wide default setting of `Public Read/Write`. You can change org-wide defaults through the page accessed by **Setup ➤ Security Controls ➤ Sharing Settings ➤ Edit**, shown below.

Organization Wide Sharing Defaults Edit — Help for this Page

Edit your organization-wide sharing defaults below. Changing these defaults will cause all sharing rules to be recalculated. This could require significant system resources and time depending on the amount of data in your organization.

Save | Cancel

Object	Default Access	Grant Access Using Hierarchies
Lead	Public Read/Write/Transfer	☑
Account, Contract and Asset	Public Read/Write	☑
Contact	Controlled by Parent	☑
Opportunity	Public Read Only	☑
Case	Public Read/Write/Transfer	☑
Campaign	Public Full Access	☑
Activity	Private	☑
Calendar	Hide Details and Add Events	☑
Price Book	Use	☑
Ad Group	Public Read/Write	☑
Candidate	Public Read/Write	☑
Google Campaign	Public Read/Write	☑
Interview	Public Read/Write	☑

Figure 123: Editing org-wide defaults

The org-wide default you choose should be based on an analysis of the intended usage of the data. Since org-wide defaults are used to lock down data, select the setting that matches the least amount of access granted to the least privileged user of your organization.

If all users of your organization are allowed to edit all records in an object, then the `Public Read/Write` setting is appropriate for the org-wide default. If all users of your organization are able to read all records in an object, but not be allowed to edit some of the records, the

`Public Read` setting is appropriate. If any of the users of your organization are not allowed to read or write any of the records in an object, the `Private` setting is appropriate.

Tip: Remember, the sharing settings are used to grant differential access to individual records within an object. If a user is not allowed access to any records in an object, you can simply not grant their profile any permissions on the object. Record-based permissions cannot override component permissions.For instance, if you do not want a user to be able to see any Position records, deny them access to the Position object. If the user is able to see some Position records, allow them the appropriate access at the object level but then limit their access to records through sharing.

The diagram shown below explains the decision flow for selecting an org-wide default for an object.

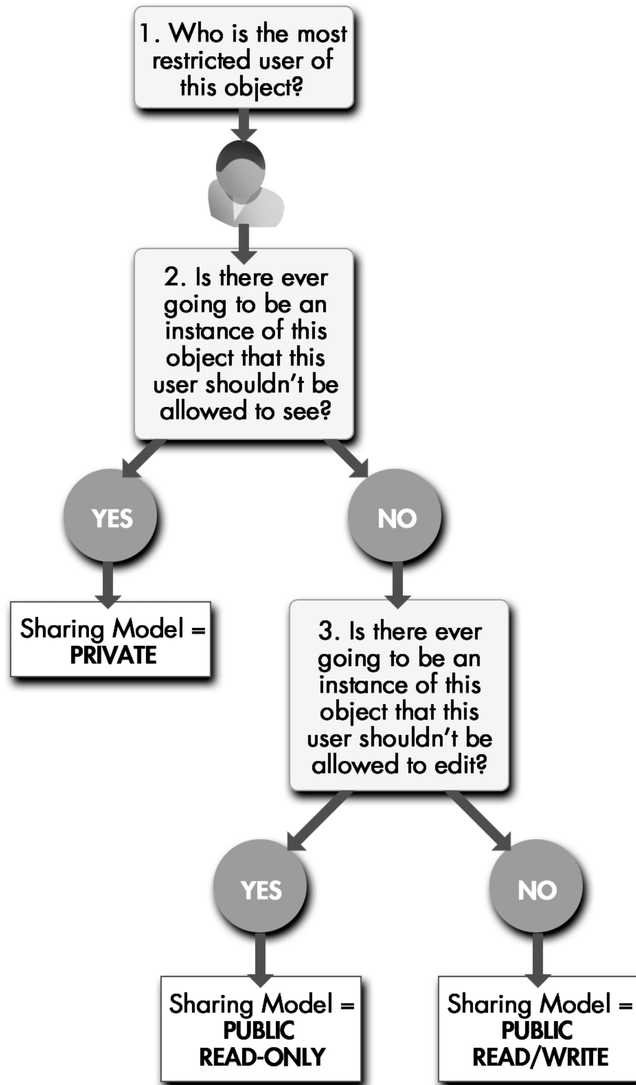


Figure 124: Decision flow for setting org-wide defaults

Sharing

You use org-wide defaults initially to lock down the data in an object, since the core goal of security is to prevent unauthorized access to critical data. You can think of org-wide defaults as adding a door to a room where your precious data is kept—a solid door that is locked (private), a door made of glass and locked (read-only) or a door that is left open (read-write

access). The Force Platform gives you a way to share a key to open that door with individual users, a process known as *sharing*.

When a record is shared with a user, the sharing privilege defines the type of access to that record. There are three levels of sharing access: `Private`, `Read Only` and `Read/Write`.

You use sharing to assign access rights that are greater than those assigned by org-wide defaults. You can only assign `Read Only`, and `Read/Write` sharing access to records in an object with a Private org-wide default setting and `Read/Write` to records with a `Public Read` org-wide default setting.

Ways to share

There are three ways that you can share a record:

- Manual sharing – If an object has an org-wide default other than `Public Read/Write`, a **Sharing** button displays on the detail page for each record. Clicking **Sharing** brings up the page shown in the following figure.

Figure 125: Sharing a record

Clicking **Add** brings up the page shown below.

Figure 126: Adding a share

On this page, you can choose to share a record with individual users or groups of users. You are given the choice of the type of sharing you wish to grant to the selected users, although you cannot grant sharing, delete or transfer permissions that are greater than the privileges you possess for the record.

Once you add sharing for users, those users show up in the list shown above. The owner of a record, or any user with Full Access permission for the record, can drop any shares added through this manual method.

- Sharing rules – In some scenarios, you want to automatically share records owned by one group of users with those in another group. You can implement this type of sharing by using sharing rules. The Sharing Settings page of the **Security Settings** menu, partially shown in the figure below, has a section for sharing rules for each object in your Force Platform database.

Candidate Sharing Rules New Recalculate Candidate Sharing Rules Help
No sharing rules specified.
Google Campaign Sharing Rules New Recalculate Google Campaign Sharing Rules Help
No sharing rules specified.
Interview Sharing Rules New Recalculate Interview Sharing Rules Help
No sharing rules specified.
Keyword Sharing Rules New Recalculate Keyword Sharing Rules Help
No sharing rules specified.
Location Sharing Rules New Recalculate Location Sharing Rules Help
No sharing rules specified.
Position Sharing Rules New Recalculate Position Sharing Rules Help
No sharing rules specified.
Positions Tracker Sharing Rules New Recalculate Positions Tracker Sharing Rules Help
No sharing rules specified.

Figure 127: Sharing Settings page

These rules automatically grant a type of sharing access for records owned by one group of users to another group. For instance, you might want to assign all users in the HR group read access to all Position records owned by users in the Recruiter group.

Note: The Winter '09 release of the Force Platform also supports criteria-based sharing, which allows you to define sharing rules based on logical criteria evaluated against the record. This feature is available as part of a Developer's Preview—please see developer.force.com for more information on this feature.

- Apex code - The end result of adding a sharing privilege is an entry into a sharing object. The Force Platform automatically creates a sharing object for every object in your database. Apex sharing uses Apex code to add entries into the sharing object. Apex code is a complete procedural language so you can implement sharing using virtually any type of logical and data-based conditions.

  The *Apex Language Reference* contains code samples to demonstrate how to implement sharing with Apex code.

Sharing Recipients

A record share is associated with either individual users or groups of users. The Force Platform comes with two ways to group users—roles, discussed later in this chapter, and public groups.

Both entities are groups of users, but roles have some special features. You can create a public group through the following steps.

1. Go to **Setup ➤ Manage Users ➤ Public Groups**, which brings up a standard list view of existing public groups.
2. Click the **New** button to define a new group on the page shown below.

Figure 128: Defining a public group

You can add individual users, other public groups, or two different choices for roles, which are discussed later in this chapter, to a public group.

To reduce the entries in a sharing object, strive to share with groups of users, rather than grant extensive sharing entries for individual users.

Changes Which Affect Sharing

Best practices suggest that you design your sharing scheme as part of your overall application design effort; however, there may be times when you will have to make a change that affects the sharing rules for a record.

Normally, changing a sharing rule automatically recalculates the sharing privileges affected by the change. If the Force Platform determines that the change affects a large number of users, you have to manually trigger a recalculation of shares. In the Sharing Settings page shown above, you can see the **Recalculate** button for each object.

If you change the org-wide defaults for an object, Force Platform recalculates shares on the object. Any shares granted to a record are dropped if they are now redundant, in that the new org-wide default encompasses the previous permission granted by the share. For instance, if you changed the org-wide default for an object from `Private` to `Public Read Only`, any read shares granted to records for the object would not be re-applied.

If you change the owner for an object, all shares for that object are dropped. New shares, based on sharing rules, are added as appropriate.

This final possibility brings up a potential problem. You might have created fairly sophisticated shares using Apex code, and if the owner of the record goes and transfers ownership to another user, all that sharing information is lost. The next section covers Apex-managed shares that address this and other issues.

Apex-managed Sharing

As mentioned previously, you can use Apex code to add shares for a record. Normally, these shares show up in the list of shares, as shown in the figure titled "Sharing a record" above. The owner of the record has the ability to delete any of these shares, and all of these shares are lost when the owner of the record changes. How can you create shares for a record and protect them from this type of destruction?

The solution is Apex-managed shares. You begin the process of creating an Apex-managed share by adding an `Apex Sharing Reason`, accessible from the main page for an object. The page for adding an `Apex Sharing Reason` is shown in below.

Figure 129: Apex Sharing Reason page

Once you create an `Apex Sharing Reason`, you can add shares to a record through Apex code, labeling them with the `Sharing Reason`.

Any shares that are associated with a `Sharing Reason` cannot be deleted by a user or a change of ownership. Additionally, you can use standard Apex data access statements to retrieve all the share records for a particular reason.

For more detail on Apex-managed shares, such as sample code and dealing with recalculation of these shares, see the *Apex Language Reference*. Apex-managed shares add a final bit of subtlety that can be crucial in designing sophisticated sharing schemes for your data.

Roles

You learned about using public groups to share records with more than one user earlier in this chapter. The Force Platform also includes a rich feature called a *role*. A role, like a public group, can include one or more users. But what makes a role different is the concept of a role hierarchy.

Each org has its own role hierarchy—you can see the sample hierarchy that comes with a default organization below.

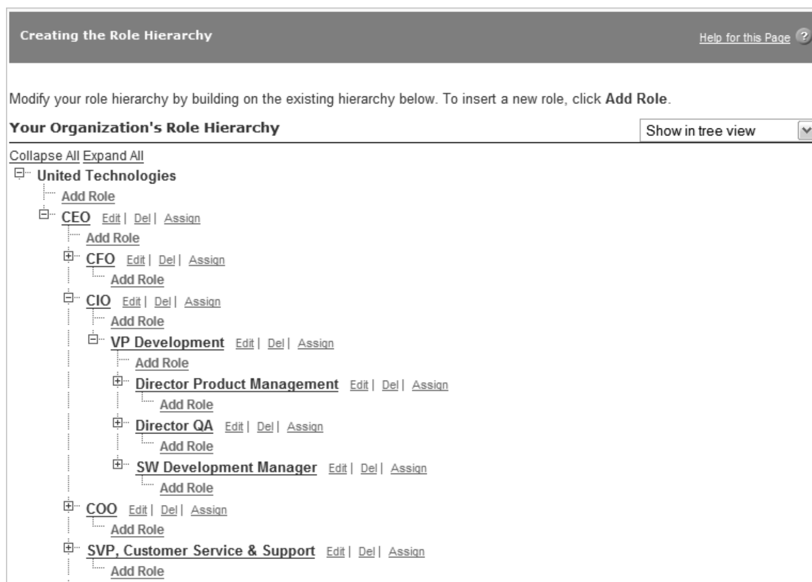


Figure 130: Role hierarchy

A role hierarchy is used to enforce a basic rule—all superior roles have all the sharing permissions granted to all roles below them in the hierarchy. In the hierarchy shown on the Roles page under Manage Users in the Force Platform environment in the figure above, anyone who is in the VP of Development role automatically has all sharing privileges assigned to anyone in the Director of Product Management role. These permissions include those of record ownership so that anyone in the VP of Development role has ownership permissions on all records owned by the Director of Product Management.

When you set org-wide defaults for a custom object, you can specify when the Force Platform should use hierarchies to grant access to users whose role is superior in the hierarchy to a specified user.

Since many organizations have a hierarchical reporting structure, using roles can help you to implement a sophisticated record-based sharing system without extensive rules or coding.

There are two main differences between roles and public groups. One is that a role hierarchy grants permissions in only one direction; your boss has all the permissions you have, but you do not have the same permissions that your boss owns.

You can have public groups that are members of other public groups, but the permission sharing is reciprocal. If Group A is a member of Group B, all members of Group A have all the permissions granted to Group B, and all members of Group B have all the permissions granted to Group A.

Secondly, a user can only belong to one role, and each Force Platform org only supports one role hierarchy, while a user can belong to many public groups. Due to these limitations on roles and role hierarchy, you might be in a situation where roles and a role hierarchy have already been defined for your org, and those definitions do not match the security scheme you would like to implement for your application. Because of this, there may be times when you have to use public groups with your sharing scheme.

Designing Security

This chapter began with a brief discussion of why security is so central to the well-being of your application. As the remainder of the chapter has demonstrated, you have an extremely flexible security model for components and data on the Force Platform.

The flexibility you have with component permissions and record sharing can deliver a sophisticated implementation scheme, but arriving at the proper definition of that scheme is, in most cases, non-trivial. You should create a robust security plan, which takes into account the eventual uses of your applications and data, as part of your design process. Although you can change the security implementation in your Force Platform org at any time, you should start to plan security as part of your initial design process. Creating a comprehensive sharing plan can help to reduce development overhead and implementation time, as well as eliminating the need for sharing recalculations.

Although this chapter has not included hands-on exercises, you can use the information from this chapter to design and implement a security scheme that is reflective of the needs and requirements of your own organization.

Summary

The Force Platform is designed to protect your data, with a host of security features built into the infrastructure of the platform.

The platform also gives you a flexible security scheme that you can use to provide differential access to your applications, data objects and other components. Additionally, you can limit access to individual records through sharing permissions.

The Force Platform provides all you need for the security of your applications and data, in its foundations and in the security tools it provides.

Chapter 12

Extended Visualforce Components and Controllers

In this chapter ...

- Visualforce Components
- Controller Extensions
- Conditional Display of Fields
- Visualforce Custom Controllers
- Summary

Back in *Chapter 9: Visualforce Pages*, you were introduced to Visualforce technology in the form of Visualforce pages that used standard controllers. You can expand the reach of your Force Platform applications even further by using Visualforce components, and both extensions to standard controllers and custom controllers, which require the use of Apex code.

In this chapter, you explore each of these areas and how they can increase your developer productivity and the productivity of your user community.

Important: If you have been following the examples in the previous chapter, your Force Platform organization should be ready for the exercises in this chapter. If you have not done the exercises, or want to start with a fresh version of the sample Recruiting application, please refer to the instructions in the Code Share project for this book. You can find information on how to access that project in *Chapter 1: Welcome to the Force Platform*. In addition, the project contains instructions on how to update your organization with changes required for this chapter.

Visualforce Components

You have already learned about Visualforce components—you used many standard components in your Visualforce pages in *Chapter 9: Visualforce Pages*. You can also create your own Visualforce components to use in the same way as standard components in a Visualforce page.

Visualforce components do not require the use of any Apex code, but you use a slightly different path to create these components, as well as some syntax that is only relevant for reusable components.

For our example, the particular Visualforce component you create addresses a particular need at Universal Containers. Universal Containers is fortunate in having extremely creative employees, but they are perhaps a bit too fortunate in this area. The company, as a whole, has had trouble deciding on a single slogan for their enterprise, so they have reached a compromise solution. All slogans displayed in their Force Platform applications have the same form, with the Universal Containers logo, and one of many mottos underneath the logo.

Further, the executives at Universal Containers like emphatic slogans, with a normal statement followed by one of increased emphasis. To account for this scenario, you will be creating a Visualforce component with this form, but with the option of assigning different slogans for each use. This section introduces you to creating and using components.

Creating a Visualforce Component

Visualforce components use all the same syntax as a standard Visualforce page, with several added tags and attributes to provide component-specific capabilities. The starting place for component creation is the Setup environment.

1. Go to **Setup ➤ Develop ➤ Components ➤ New**, which will take you to the page shown in below.

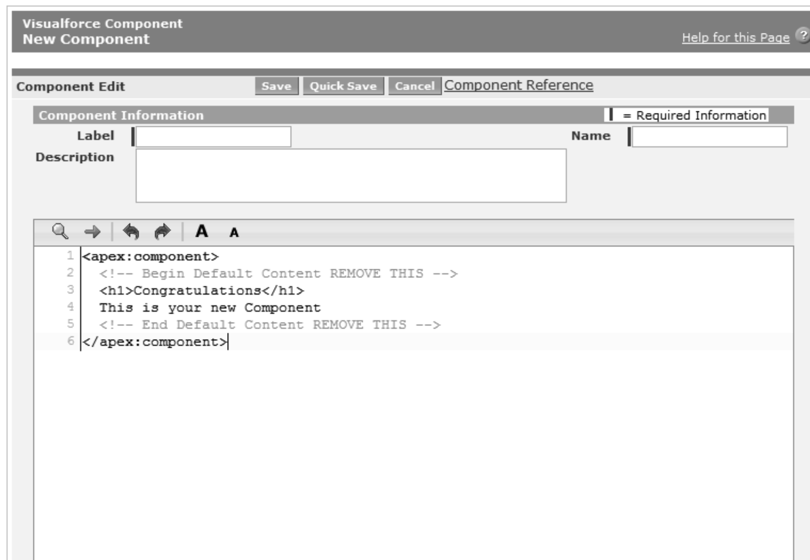


Figure 189: Creating a Visualforce component

2. Give the component the name of `slogo`, a combination of slogan and logo, and an appropriate description.
3. Modify the code for the page, eliminating the default code and adding an image component and an HTML page break, as shown in the code below:

```
<apex:component>
 <apex:image value="{!$Resource.Logo}"> </apex:image>
</apex:component>
```

You have just created your first component, The next step is to give this component more flexible capabilities.

Adding Attributes to a Component

You have created a basic component, with the Universal Containers logo. You now want to add a way for an instance of this component to receive words for the slogan.

You can create attributes to receive values passed from the Visualforce page that includes the component. An attribute, in a Visualforce component, is a way to pass values to the component. All attributes need a name, a description, and a data type, which is a primitive data type, such as string, number, or date. An attribute can also be specified as required or not.

1. Modify the code for the component by adding two attribute tag sets, as shown in the following code:

```
<apex:component>
<apex:attribute name="firstText"
       description="The first text string
         that is not italic"
       type="String" required="false"/>
<apex:attribute name="firstItalicText"
       description="The first italic text string."
       type="String"
       required="false"/>
<apex:image value="{!$Resource.Logo}"> </apex:image>
</apex:component>
```

   Now that your component includes attributes, you can use them in the display of the component.

2. Modify your component code to use the attributes within the component, as shown in the following code:

```
<apex:component>
  <apex:attribute name="firstText"
       description="The first text string
         that is not italic"
       type="String" required="false"/>
  <apex:attribute name="firstItalicText"
       description="The first italic text string."
       type="String"
       required="false"/>
  <apex:image value="{!$Resource.Logo}"> </apex:image>
  <p/>
  {!firstText} <i> {!firstItalicText}</i>
</apex:component>
```

As you can see, an attribute within a component is accessed just like any other field, through bind syntax.

Your component is now complete and ready to use.

Using a Visualforce Component

Now that your Visualforce component has been created, you can use it in any of your Visualforce pages.

1. Call up the Visualforce page you created in *Chapter 9: Visualforce Pages* through the URL of
   `http://c.instance_name.visual.force.com/apex/VisualforcePositions`

2. Add the component to the page, just below the opening page tag, with the following code:

```
<c:slogo firstText="Reaching the top - "
 firstItalicText="together"/>
```

The tag used to insert the component does not start with `apex:`, but with `c:`. The initial portion of any Visualforce tag indicates the namespace for that tag. A namespace is simply a way of separating Force Platform components to insure unique naming for the components.

Tip: Namespaces become important when you learn about packaging, discussed in *Chapter 15: Deployment*.

The `c:` namespace refers to any component in your org that has not been assigned to a specific namespace, such as the slogo component.

3. Save your page code to refresh your page and to display your new slogo component, as shown in the figure below.

Figure 190: Your Visualforce component at work

Although you have only used your new Visualforce component in a single page, you can easily see how this flexible entity is used across your entire application.

Visualforce components can also be linked to their own custom controllers, which you will learn about later in this chapter.

Controller Extensions

In *Chapter 9: Visualforce Pages*, you experienced the different granularity of various Visualforce components. For instance, a single set of tags can create an entire detail page. But if you want to change the way a detail page looks beyond the scope of page layouts, you must build the entire Visualforce page using more discrete components.

Visualforce controllers give you similar options, but with a difference. You can build a custom controller with Apex code that replaces all the functionality of a standard controller, and there are times when this approach may be desirable; however, Visualforce also gives you another option, in the form of controller extensions.

A controller extension uses all the functionality in a standard controller. The controller extension expands or modifies that functionality by adding new capabilities or replacing standard capabilities with modified versions of the same actions. With this approach, you do not have to recreate everything that a standard controller does to simply tweak or extend its operation.

In this section, you create a controller extension to handle dynamically populating a picklist based on the value of another picklist. You are, in effect, duplicating the operation of a standard dependent picklist, but in a way that allows you to extend that functionality in the next section.

The dependent picklist adds a value for a position type to each Position record. The values available for this field are dependent on the particular department. For instance, you might have a Developer position in the Engineering department, but not in the Finance department. Time to begin.

Modifying Your Force Platform Database

Your first step is to add an object to your Force Platform database. The Position Type object will contain two fields: the Position Type field, which is the name of a position, and the Department field, which associates the particular Position Type with that Department.

In addition, you need to create a lookup relationship between the Product object and the new Position Type object.

The initialization task described at the start of this chapter created both of these components for you.

Creating a Controller Extension

Your next step is to create a controller extension to provide the new functionality for the Visualforce page.

The controller extension is an Apex class, which you learned about in the previous two chapters.

1. Create a new Apex class through clicking on the **Setup ➤ Develop ➤ Apex Classes ➤ New** button.
2. Add the following code to begin the creation of your controller extension:

```
public class positionExtension {

}
```

The initial code for this class should be familiar to you from the previous two chapters. You simply begin the class by identifying it and creating the pair of brackets that enclose the code.

3. Add the following line of code to the class:

```
public class positionExtension {

private final Position__c position;
}
```

This line of code is one of the two ways that you will tie your controller extension to the standard controller. The code creates an sObject with the structure of the Position__c object, and gives that object the name of position. This object will only be accessible to methods inside this class, so it is defined as `private`

.

4. Add the highlighted code to the class:

```
public class positionExtension {

private final Position__c position;

public positionExtension(ApexPages.StandardController
       positionController) {
     this.position =
       (Position__c)positionController.getRecord();
     }
}
```

This second step creates a controller extension object and then loads the position record with the current record used by the standard controller for the Position__c controller. The position

variable is now tied to the same record as the standard controller, allowing the controller extension to operate in synch with the standard controller.

You have successfully created the shell of a controller extension. Your next step is to add the functionality you need to offer to your Visualforce page.

Populating a List of Options

The real work of your first controller extension is to provide a method to populate the picklist for the new Position_Type__c field.

1. Add the highlighted code to your controller extension:

```
public class positionExtension {

private final Position__c position;

public positionExtension(ApexPages.StandardController
       PositionController) {
    this.position =
      (Position__c)PositionController.getRecord();
    }

public List<selectOption> positionTypeOptions {get; {}
 private set;}

}
```

This property is declared with get and set accessors. When the variable is requested from a Visualforce page, the get accessor returns a value. Normally, when a page returns a value for the variable, the set accessor assigns that value. The `private` keyword before the set designates that only the code in this class can set the value of the property. If you do not define code for these properties, the Force Platform will take the default action.

The `positionTypeOptions` variable returns a list of values with a data type of `selectOption`. This data type contains a label that is displayed to the user and a value that can be bound to a column in a Force Platform object. The body of the `get` method populates this list.

2. Add the highlighted code to your controller extension:

```
public class positionExtension {

private final Position__c position;

public positionExtension(ApexPages.StandardController
```

```
       PositionController) {
    this.position =
      (Postion__c)postionController.getRecord();
    }

public List<selectOption> PositionTypeOptions {get {
    List<selectOption> positionTypes =
         new List<selectOption>();
    return positionTypes;
    }
    private set;}
}
```

These two lines of code create a list of `selectOption` fields, and then returns this list in response to a request from the Visualforce page.

3. Add the highlighted lines of code to your controller extension:

```
public class positionExtension {

private final Position__c position;

public positionExtension(ApexPages.StandardController
       postionController) {
    this.position =
      (Position__c)postionController.getRecord();
    }
public List<selectOption> PositionTypeOptions {get {
    List<selectOption> positionTypes =
      new List<selectOption>();
    for (Position_Type__c ptr :
        [select name
           from Position_Type__c pt
           where pt.Department__c =
           :position.Department__c order by Name ])
      positionTypes.add(new selectOption(ptr.id, ptr.name));
    return positionTypes;
    }
    private set;}

}
```

This simple `for` loop should be familiar to you from the previous chapters on Apex code. The `for` loop iterates through the result set returned from the SOQL statement. This statement selects a label for the selectList entry, the name of the record, that displays to the user. As with all SOQL statements, the unique Force Platform ID is also returned with the result set.

The SOQL code introduces a new data object, which was added to your organization with the initialization procedure at the start of this chapter. The Position_Type__c object contains a set of records which associate a particular type of position with the

department which offers that position. The Position object also contains a lookup field to this object. The net effect of this interaction is to create a list of position types which are appropriate for the department which is offering a position - similar to the work of a dependent picklist, but with the ability to extend those capabilities, as ou will see.

The `add()` method for `selectOptions` adds two values to the object: the value for the entry and a label that presents to the user in the selectList. You use the ID as the value for the entry since the Position_Type__c is defined as a lookup relationship, which always points to the ID of the parent field. The name of the Position_Type__c record is the label for the entry in the select list.

You have one more task to accomplish before you can save this first version of your controller extension code.

4. Add the highlighted code to your class to declare another property:

```
public class positionExtension {

private final Postion__c position;

public string positionTypeID {get ; set ;}

public positionExtension(ApexPages.StandardController
       postionController) {
    this.position =
      (Position__c)postionController.getRecord();
    }
public List<selectOption> PositionTypeOptions {get {
    List<selectOption> positionTypes =
      new List<selectOption>();
    for (Position_Type__c ptr :
        [select name
           from Position_Type__c pt
           where pt.Department__c =
           :position.Department__c order by Name ])
      positionTypes.add(new selectOption(ptr.id, ptr.name));
    return positionTypes;
    }
    private set;}

}
```

The `selectList` that you will have in your Visualforce page will bind to this positionTypeID property. Why can't you simply bind the component to the Position_Type__c lookup field in the Position__c object? You will understand the full reason for this in the functionality you will add to your controller extension later in this chapter. You will be giving your users a chance to add new dependent values

for the Position Type dynamically, so you will have to connect the selectList to an intermediate location—the positionTypeID property.

5. Save your code.

There you have it—the first version of your controller extension is complete. You can now create a Visualforce page to use your new functionality.

Your Visuaforce Page with Controller Extension

You need to use a Visualforce selectList component to implement the dynamic picklist operation for the new field, so you must have a Visualforce page to enter Position information. The load that you performed at the start of this chapter also added the basic Visualforce page with a name of `VisualforceExtension`. The code for this page is shown below.

```
<apex:page standardController="Position__c"
extensions="positionExtension" >
    <apex:form >
        <apex:sectionHeader title="Add New Position"/>
        <apex:pageBlock mode="edit" id="thePageBlock">
            <apex:pageBlocksection title="Information">
                <apex:inputField
                  value="{!Position__c.Location__c}"/>
                <apex:inputField
                  value="{!Position__c.Hiring_Manager__c}"/>
                <apex:inputField
                  value="{!Position__c.Status__c}"/>
                <apex:inputField
                  value="{!Position__c.Notification__c}"/>
                <apex:inputField
                  value="{!Position__c.Start_Date__c}"/>
            </apex:pageBlocksection>
                <apex:pageblocksection columns="1"
                  title="Department">
                    <apex:inputField
                      value="{!Position__c.Department__c}"/>
                </apex:pageblockSection>
            <apex:pageBlockSection title="Position Details">
                <apex:inputField
                  value="{!Position__c.Job_Description__c}"/>
                <apex:inputField
                  value="{!Position__c.Responsibilities__c}"/>
                <apex:inputField
                  value="{!Position__c.Programming_Languages__c}"/>
                <apex:inputField
                 value="{!Position__c.Educational_Requirements__c}"/>

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

Although there is more code than in the Visualforce pages you worked with in Chapter 9, almost all of this code is familiar. You have `inputField` tags for each of the fields in the Position__c object that requires user input, and you have several different `pageBlockSection` tags, with appropriate titles, to break up the page for formatting purposes.

The most important change is the new `extensions` attribute in the initial page tag. With this tag, you have made the functionality implemented in your PositionExtension controller accessible to this page. As the name of the tag implies, you can have more than one controller extension for a page.

There is a new attribute in the initial `pageBlock` tag that designates the mode of the page as `edit`. This attribute simply sets up the appearance of the page to model that of a standard Force Platform edit page, without lines between the different input fields.

There is also a new tag just above the `pageBlock` tag for a `sectionHeader`. The tag itself gives this page a header just like a standard page, with the same colors and icon associated with the object (or defined as part of the `apex:page` component) and the text listed as the title tag. You can also have a `subtitle` tag with an additional line of text. This page, in its current state, is shown below.

Figure 191: The initial version of your Visualforce controller extension page

Adding an apex:selectList

The page shown above presents an interface to most of the fields in the old version of the Position__c object. You need to add a new field to take care of the new Position_Type__c field in the object.

For all of the previous fields, you were able to simply use the `inputField` component within a `pageBlockSection`. This combination allowed your Visualforce page to simply use all the defaults associated with the field definition—yet another example of the metadata driven quality of the entire Force Platform environment.

This default will use a lookup field for the Position_Type__c object, but you want to add a little more functionality. As you saw in the definition of the Position_Type__c object, a user should not be able to see all position types for a position – just those associated with the department of the position. This functionality is very similar to that presented with a dependent picklist, but, as you will see later, can be expanded with additional capabilities.

The new field will be a `selectList` that is displayed as a picklist on the page, but the relationship to the Position_Type__c field is not as direct. Although you want to save the result of the user's selection into that lookup field, you populate the `selectList` with come of the code in the controller extension you created.

Because of this goal, you must use a slightly different set of tags for this field.

1. Bring up the new Visualforce page in your Force Platform environment, by adding `/apex/VisualforceExtension` to the base URL of your Force Platform org, explained above as `http://c.instance_name.visual.force.com`.
2. Open the Page Editor for your new page.
3. Add the highlighted code to your Visualforce page. The code is shown below, along with the code that precedes and follows it in the page.

```
<apex:pageblocksection columns="1" title="Department">
  <apex:inputField
    value="{!Position__c.Department__c}"/>
</apex:pageblockSection>
<apex:pageblockSection id="dependentPositionType"
  columns="1">
<apex:pageBlockSectionItem >
  <apex:outputLabel value="Position Type" for="pt"/>
  <apex:panelGrid columns="2">
    <apex:outputText
      value="{!Position__c.Position_Type__c}"
      rendered="false"/>
    <apex:selectList id="pt"
      value="{!positionTypeID}" size="1" >
      <apex:selectOptions
```

```
          value="{!PositionTypeOptions}"/>
      </apex:selectList>
   </apex:panelGrid>
   </apex:pageBlockSectionItem>
</apex:pageBlockSection>
<apex:pageBlockSection title="Position Details">
```

You have added a new `pageBlockSection` to enclose the new `selectList` item, a requirement since you want to use a partial page refresh to refresh the values in this picklist when the value for the Department field changes.

The first new thing to see in the code for this new item is the use of the `pageBlockSectionItem` tags. Since your new picklist is not bound to a particular field in a Force Platform object, you need to use these tags to force the Visualforce page to use the same formatting for the field listed within the tags. The `pageBlockSectionItem` tags use the `outputLabel` as the label of the field and the next Visualforce component as the input field, in this case the `selectList` component. The `outputLabel` is further associated with the `selectList` by the use of the `id` tag on the `selectList` and the `for` tag that points to that ID in the `outputLabel` tag.

The `selectList` component presents to users as a picklist. The size attribute indicates the number of entries initially for the picklist. With a `size="1"` attribute, the picklist is displayed as an entry field with a drop down list, like a standard picklist.

You can see that the `selectList` is bound to the `positionTypeID` variable in the controller extension field through the familiar value attribute. As you saw inn the previous section, the variable is used to hold the value for the lookup field Position_Type__c, as well as a value that you will be using later in this chapter.

The `panelGrid` component is here for a specific purpose, but one that is not yet apparent. You can see that the component has a `columns="2"` attribute, which allows two components to be placed in a single column pageBlockSection. Right now there is only one component, but you will see the use of the grid later in this chapter.

Why the hidden column?

You may have noticed that there is an `outputText` column bound to the Position_Type__c field, but hidden. You need this column as having a column bound to a field in a Visualforce page is the only way to include the column in the SOQL query for a standard controller. Since you will want to insert a value into this column,

having it referenced by field, but not shown, is the most efficient way to make sure the column is available to receive data.

You want to populate the values for this picklist with the values retrieved with the `get` method of the `postionTypeOptions` variable. The values in the list are populated with the `selectOptions` tag. The code behind that get method limits the values placed in the picklist through the selection condition in the SOQL statement.

4. Save the new version of your Visualforce page.

If you have modified the code properly, your new page should look like the figure below.

Figure 192: The next phase of your Visualforce extension page

Your new page does include a picklist for the Position Type field, but the values for that picklist are not being dynamically repopulated. Your next step is to add this key feature to your page.

Repopulating the selectList

In order to properly repopulate the Visualforce selectList, you can add AJAX functionality to your page, which is implemented automatically without having to write any Javascript code, whenever the user has changed the value in the Department list.

As you learned in Chapter 9, this interactive capability is easy to use with Visualforce.

1. Add the highlighted code to your Visualforce page.

```
</apex:pageBlocksection>
     <apex:actionRegion >
         <apex:pageblocksection columns="1"
           title="Department">
            <apex:inputField
              value="{!Position__c.Department__c}"/>
            <apex:actionSupport event="onchange"
                rerender="dependentPositionType"
                status="departmentStatus"/>
              <apex:actionStatus id="departmentStatus"
                 startText="Fetching position types..."/>
         </apex:pageblockSection>
          <apex:pageblockSection id="dependentPositionType"
```

   Adding the `actionSupport` component to the `inputField` for the Department__c field enables this field to react to user actions—in this case, the `onchange` event that fires whenever the value for the field changes.

   You have to include the `actionRegion` tags to insure that the only field values sent back to the controller are the Department and Position Type fields. If you send the entire page back, and the user does not enter a value for one of the required fields, the controller will return an error.

   In response to this event, you want to re-render the `pageBlockSection` that contains the `selectList` for Position Type, which will cause the component to call the `get` method for the field again.

2. Save the code for your Visualforce page.
3. Change the value in the Department picklist to see the dynamic refresh of the Position Type picklist at work.

This single component brings it all together—the AJAX capability of the Visualforce page and the additional functionality implemented in your controller extension.

Housekeeping

So far, so good—in fact, almost completely good. Your selectList is repopulating as you want, but remember, getting your application to do what you want is only the first step. You also have to insure that the appropriate values in your controller are properly initialized, and that your controller does not allow users to do what they are not supposed to do.

First to the initialization.

1. Add the highlighted line of code to your positionExtension class.

```
public positionExtension(ApexPages.StandardController
       postionController) {
    this.position = (Position__c)
     postionController.getRecord();
 positionTypeID = position.position_type__c;
    }
```

   Remember that the `positionTypeID` property is the intermediary between the Position_Type__c field and the input from the Visualforce page. This code initializes the value of that property to the initial value of the field in the current Position__c record.

   The second housekeeping task can have a significant impact on data integrity. Remember that any enforcement of the relationship between the Position Type and the Department must be handled by your implementation.

   When the value for the Department changes, the list of possible choices for Position Type changes. But the Position Type picklist is bound to the `positionTypeID` property and the value of this field does not change. You can have a scenario where the user selects a Department and then an appropriate Position Type. The user then changes the Department and saves the record. The value for the Position Type `selectList` can still be set to a position type for the previous department.

   You can adjust for this problem by simply resetting the value for the positionTypeID field to null whenever the value for the Department field changes. To do this, you need to add a new method to the controller extension and call that method when the AJAX change event occurs.

2. Return to the development environment for the controller extension class by clicking **Setup ➤ Develop ➤ Apex Classes** and then clicking the **Edit** link for your controller extension.

3. **3.** Add the following code to the end of your existing controller code, just before the final right bracket:

```
public void resetPositionType() {
    positionTypeID = null;
    }
```

   This function is very simple —it sets the value of the positionTypeID property to null.

   Now call this function in your Visualforce page, which is done in an even simpler fashion.

4. Save the code for the new version of the controller extension and return to your Visualforce page code in the Page Editor by entering the base URL, followed by `/apex/VisualforceExtension`.
5. Modify the actionSupport tag to include the action attribute, as shown in the highlighted code below.

```
<apex:outputText value="{!Position__c.Position_Type__c}"
   rendered="false"/>
<apex:selectList id="pt" value="{!positionTypeID}"
  size="1">
<apex:selectOptions value="{!PositionTypeOptions}"/>
<apex:actionSupport event="onchange"
  action="{!resetPositionType}"
  rerender="dependentPositionType"
  status="typeStatus"/>
</apex:selectList>
```

6. Save your changes and change the value in the Department picklist to insure that your page works properly.

   Although you modified the way your page works, you cannot really see the effects of the modification simply by the way the page acts. The value of the Position_Type__c field only becomes an issue once that value is stored, which you will address in the next section.

   But before you get to that section, you have one other scenario to address. Remember how dependent picklists operate when there is no value selected for the parent picklist? The dependent picklist is disabled, which makes the picklist incapable of accepting user input.

   You can accomplish the same result with a little more controller code and a single attribute on your Visualforce page.

7. Modify the selectList that receives the Position_Type__c values to add the highlighted new attribute:

```
<apex:outputText value="{!Position__c.Position_Type__c}"
  rendered="false"/>
<apex:selectList id="pt" value="{!positionTypeID}"
  size="1"
  disabled="{!ISNULL(Position__c.Department__c)}">
<apex:selectOptions value="{!PositionTypeOptions}"/>
<apex:actionSupport event="onchange"
  action="{!resetPositionType}"
  rerender="dependentPositionType"
  status="typeStatus"/>
</apex:selectList>
```

With this single attribute, you are disabling the component, based on whether the Department__c field is null. If the Department picklist is null, you want to disable the selectList that contains the Position Type.

Saving the Record

Your Visualforce page is now working just great. The dynamic repopulation of the Position Type picklist is swinging along splendidly, with data integrity enforced. But what do you do with this value once your user has selected it?

As mentioned at the beginning of this section, a controller extension can use all the functionality in a standard controller. In this case, that means you can use the standard Save and Cancel actions by simply adding buttons to your Visualforce page to call them.

8. Add the highlighted code to your Visualforce page:

```
<apex:pageBlock mode="edit" id="thePageBlock">
  <apex:pageBlockButtons >
    <apex:commandButton
      value="Save" action="{!save}"/>
    <apex:commandButton
      value="Cancel" action="{!cancel}"/>
  </apex:pageBlockButtons>
<apex:pageBlocksection title="Information">
```

The code shown here is added near the top of your page, but you could have added it in other places just as easily, as long as it is a direct descendent of a `pageBlock`. As you already have seen, Visualforce components that begin with `pageBlock` inherit their style and, in this case, functionality from the standard Force Platform controller. As described previously, the `pageBlockButtons` component indicates

that the `commandButtons` within these tags display just like standard Force Platform buttons, at the top and bottom of the page.

Similarly, all you have to do to call the standard Save and Cancel methods is to use them as the action attribute of the `commandButtons`.

9. Save your Visualforce page. The refreshed page should look like the figure below.

Figure 193: Your Visualforce controller extension page - with Position Type disabled

And there you have it—a new Visualforce page that delivers the functionality you added to your controller extension.

Integrating the page

Now that your Visualforce page is working properly, the final step is to integrate the page into your Force Platform application. You want your users to be able to use this page whenever they go to add a new Position or to edit an existing Position.

You achieve this result by overriding the default pages used for adding and editing a Position record.

1. Go to the **Setup** ➤ **Create** ➤ **Objects** ➤ **Position** page, and then to the Standard Buttons and Links area.

2. Click the **Override** link for the New action to bring up the page shown below.

Figure 194: Overriding a standard action

3. Select the `Visualforce Page` for the `Content Type`, and then select the page you just created in the Content Type picklist. Notice that you are only presented with Visualforce pages tied to the controller for this object.
4. Save your work.

   When you return to the properties page for the Position__c object, you see that the New standard action now has an additional link, **Reset** that returns the destination of this action to the standard page.

5. Override the Edit action in the same manner.
6. Click the Positions tab in your Force Platform application and then **New**. You can see your new Visualforce page in action.
7. Click **Cancel** to return to the list page for Positions. Select a current position and then click **Edit** to again call up your Visualforce page.

This final integration was a snap. You can replace any of the standard pages in your Force Platform application with a Visualforce page while retaining the use of others. In this way, you can use the power of Visualforce technology without sacrificing any of the default power of the Force Platform.

The Next Step

Is there a question running through your mind right now? Are you asking “Wait—isn't this exactly what a dependent picklist can accomplish?” Well, yes, it is.

And, if you are on your way towards being a great Force Platform developer, the next question in your mind should be “Then why don't I just use that declarative feature?” Another good question.

You always use the built-in power of the Force Platform if you can, and if all you want is a dynamic display in a picklist, based on the value of another picklist, you certainly should use a dependent picklist. Those perceptive Force Platform developers already gave you this capability, nicely delivered through a declarative interface.

In the next section, you start to understand that the purpose of using your own implementation is to give your users a capability that dependent picklists do not possess, and to learn a little more about the things your Visualforce page can do for you and your users.

Conditional Display of Fields

The previous page worked well, dynamically displaying different values in a picklist based on the value of another picklist. Yes, this functionality is a recap of what is available in standard dependent picklists—so why reimplement it?

The reason to use this version of a dependent picklist is that you can expand the capabilities of this set of picklists to address a common user need. Normally, the position type for a particular position is available in its picklist. But what about those times when a user is posting a new type of position, one that is not currently available?

For a dependent picklist, this scenario is resolved by someone with developer privileges going in and modifying the values available on the dependent picklist.

With this new implementation, you can add a field to the page that allows a user to add a new position type. Of course, you do not want the user to add a new position type until they have at least looked at the currently available options.

In this section, you learn how to add a value to the Position Type list that triggers the display of an entry field for a new position type, and how to modify your controller code to save this new position type in a new record in the Position Type object.

Adding a Value to the selectList

The key to the capability you want to add to your Visualforce page is a value. This value is used to trigger the display of fields that allow a user to add a new position type for a particular department.

For the purposes of this exercise, you can use the value of `Other` to indicate this scenario.

1. Return to the controller extension you created earlier in this chapter through **Setup ➤ Develop ➤ Apex Classes** and then the **Edit** choice for the positionExtension class.
2. Modify the code that returns the `selectOptions` for the Position Type to include the highlighted lines of the following code:

```
public List<selectOption> PositionTypeOptions {get {
    List<selectOption> positionTypes =
      new List<selectOption>();
    for (Position_Type__c ptr :
        [select name from Position_Type__c pt
          where pt.Department__c =
          :position.Department__c order by Name ])
      positionTypes.add(new selectOption(ptr.id, ptr.name));
      if (position.Department__c != null) {
          positionTypes.add(new
            selectOption('other', 'Other'));}
    return positionTypes;
    }
    private set;}
```

   This code adds another option to the list of position types, if a department has been selected. If a user selects this option, they have the opportunity to enter a new position type for the chosen department. You add the user interface for this new capability in the next section.

   With this new code, you can see why you had to create the `positionTypeID` property to hold the ID of the Position Type. Up until this point, all the values for the `positionTypeOptions` have been valid ID values. You can bind the value of a `selectList` with only those ID values to the Position_Type__c lookup field, which only accepts an ID.

   However, the value of `'other'` is not a valid ID. As soon as your code attempts to associate that string with the Position_Type__c field, you will get an error. Consequently, you cannot bind the `selectList` to the Position_Type__c field, requiring the use of the `postionTypeID` property.

   Before leaving this section of the code, you will add one option to the list returned to the page to help your users.

3. Add the highlighted code to your Apex class:

```
public List<selectOption> PositionTypeOptions {get {
    List<selectOption> positionTypes =
      new List<selectOption>();
    for (Position_Type__c ptr :
```

```
        [select name from Position_Type__c pt
          where pt.Department__c =
          :position.Department__c order by Name ])
      positionTypes.add(new
        selectOption(ptr.id, ptr.name));
      if (position.Department__c != null) {
          positionTypes.add(new
           selectOption('other', 'Other'));
          }
Else {
          positionTypes.add(new
           selectOption('', '
           Please select a department', true));
       }
   return positionTypes;
   }
   private set;}
```

This condition adds a user-friendly message to the `selectList` for the Position Type if there is no value for the Department `selectList`. The message appears in the `selectList` and gently guides the user towards their next action. The constructor method for this `selectOption` includes a third optional parameter which, if true, indicates that this option is disabled in the `selectList`.

Note: Limiting the power

This modification of the controller extension allows everyone to have the ability to add a new position type, a power you might not want to grant so lightly. You can easily add an `if` condition for adding that last value to the `selectListOptions` so that only certain users, or users with a certain profile, have that value in the Position type picklist. Since the presence of this value controls the rest of the operations, eliminating the `Other` value prevents users from accessing the ability to add a new position.

Conditionally Displaying Fields

Now that you have added a data value that allows a user to add a new position type, you have to add a field to accept the Name for the new type. However, you do not want this field to show up all the time —only when a user has chosen the `Other` choice.

Visualforce components have an attribute that controls their display, just as another attribute controls whether the component is enabled. With this attribute, and another re-rendering of the page, you can provide this dynamic interface to your users.

1. Return to the Page Editor for your current Visualforce page. Since the page is now tied into the standard New action, you can get to the page by clicking on the Positions tab and then **New**.
2. Add the highlighted code to the Visualforce page:

```
<apex:pageblockSection id="dependentPositionType"
  columns="1">
    <apex:pageBlockSectionItem >
      <apex:outputLabel value="Position Type" for="pt"/>
      <apex:panelGrid columns="2">
       <apex:actionRegion >
        <apex:outputText
           value="{!Position__c.Position_Type__c}"
           rendered="false"/>
        <apex:selectList id="pt"
          value="{!positionTypeID}" size="1"
          disabled="{!ISNULL(Position__c.Department__c)}">
            <apex:selectOptions
              value="{!PositionTypeOptions}"/>
              <apex:actionSupport event="onchange"
                rerender="dependentPositionType"
                status="typeStatus"/>
        </apex:selectList>
        </apex:actionRegion>
           <apex:actionStatus id="typeStatus"
              startText="updating form...">
            <apex:facet name="stop">
            <apex:inputField
               rendered="{!positionTypeId == 'other'}"
               required="true"/>
            </apex:facet>
            </apex:actionStatus>
      </apex:panelGrid>
    </apex:pageBlockSectionItem>
</apex:pageBlockSection>
<apex:pageBlockSection title="Position Details">
```

There are three different places you have added code. The first is to define an `actionRegion` again, as you did previously. The second is to add the `actionSupport`, which rerenders the `pageBlockSection` that contains the Position Type information.

The third area is the `actionStatus`. As before, you use `actionStatus` to display text indicated that a refresh of the form is taking place. The next set of tags define a facet. A facet is an area in a page that can contain other text or, in this case, a component. The `stop` facet of an `actionStatus` component is displayed when the action has completed – in other words, once the dependentPositionType has been refreshed. This facet will contain an `inputField` to receive the name of a new Position Type. But you only want to show this field if the value selected for Position

Type is `Other`. The `rendered` attribute checks to see if the `positionTypeId`, which is the variable bound to the `selectList` for Position Type, contains `Other`. The `inputField` will display only if this condition is true.

Note that you have not bound your new `inputField` to a value in the controller yet. You will define the destination for a new Position Type in the next section, and then come back and bind the field with the `value` attribute.

Your Visualforce page will save without an error, but if you run it now, you will get a runtime error if you picked `Other` to try and show the `inputField` for the new Position Type. The error comes because the inputField is not yet bound to a field in the controller, which cannot be done until you add an instance of the Position_Type__c object in the controller. The next section will address this issue.

Holding the New Value

You may think of your Visualforce controller as simply a way to interact with data objects, but your controller performs an equally important role as a place where values are maintained on the Force Platform. When a user adds a value for a new position type into the dynamically displayed input field you just defined, your Visualforce controller receives that value and keeps it until your user decides to save the value to persistent storage in a Force Platform data object.

The eventual destination for this new position type is the Name field of the Position_Type__c object, so you first modify your controller extension to create a record that can be used to hold the new value.

1. Go to edit your controller code through **Setup ➤ Develop ➤ Apex Classes** and select the **Edit** choice for your custom controller.

Voice of the Developer:

While actively working on Visualforce pages and controllers, you can increase your productivity by having two tabs open, with one displaying the Page Editor for the Visualforce page and another in the Setup menu for the platform to make it easy to switch between the two environments.

2. Add the following property declaration to the existing code:

```
public Position_Type__c newPositionType{
 get{
   if (newPositionType == null) {
     newPositionType = new Position_Type__c();}
   return newPositionType;
```

```
    }
 private set;
}
```

This property will be used to insert a new Position_Type__c record, if the user create a new position type. The `get` accessor checks to see if the property has been initialized with a new record instance—if not, the method creates the new instance.

3. Save the new version of your controller code.

   With this step, you have created an object to hold the name for a new Postion Type. To open the channel between the user interface of the Visualforce page and your Visualforce controller extension, you will have to go back to your Visualforce page and add the `value` attribute to the last inputField.

4. Return to edit your Visualforce page. Add `value="{!newPositionType.Name}"` to the `inputField` you added in the previous section.
5. Save your Visualforce code. Try changing the value of the Position Type `Other` and back to see the new fields appear and disappear from the page. The page, with the new components visible, should look like the figure below.

Figure 195: New field appearing on your Visualforce page

Your newly dynamic page is pretty nice, but you need to save the results of this pretty interface in your controller since you want to save the user's input into Force Platform objects.

Your last task to complete this implementation is to save this new value in the Force Platform database.

Saving the New Value

The last change you make to your controller extension is probably the biggest, in terms of its impact. A standard controller, by definition, is linked to a single Force Platform object. The functionality you have added to your Visualforce page calls for more than this. Once a user adds a new Position Type, you must add a record for this new value to the Position_Type__c table and then use the ID for this new record as the value for the Position_Type__c field in the Position__c table.

From the page point of view, you do this by replacing the `Save` method for the standard Position__c controller. This section covers the five basic logical steps for creating a successful save operation:

- Check for a new position type
- Add a new position type
- Set the new position type lookup
- Saves the Position record
- Catch errors

Checking For a New Position Type

If your user has created a new position type, you must add a record to the Position_Type__c object to represent the new type, but you only want to take this action if a new position type is indicated.

First, create a save method in your controller extension that makes this logical test.

1. Return to edit your controller extension class. Add the following code to the end of the existing code, just before the closing right brace:

   ```
   public PageReference save() {

           }
   ```

   This code defines the save method. This save procedure in your controller extension replaces the save procedure in the standard controller for the Position__c object, although your new save procedure will end up calling the default save for the object once you finish adding the specialized logic.

The save procedure returns a PageReference. The PageReference creates a URL that identifies the page sent back to the browser when the procedure completes. The default save procedure will take care of assigning a value to this return, but you will set PageReference variables explicitly later in the chapter.

2. Modify the save method to add the logical test shown highlighted below:

```
public PageReference save() {
       if (PositionTypeID == 'other') {
        }
       }
```

Why did you check for the other value in the PositionTypeID property instead of looking for a value in the newPositionType.Name field? Your user might have selected the Other choice for the position type, added a value in the inputField for the Name, and then changed the position type back to an existing position. The inputField would not be visible, but it would still have the previous value. Since the presence or absence of other value in the PositionTypeID property is the marker for this logical state, you should test that condition.

3. Click Quick Save to check your code for errors.

The Quick Save button parses and saves the Apex code in the edit window without sending you back to the list of Apex classes. Since you are still in the midst of developing your code, take this route to insure that you have entered the proper syntax.

Adding a New Position Type

If the user is adding a new Position Type, set the values for both fields in the record and save the record.

4. Add the highlighted code to the new save method in your Apex class:

```
public PageReference save() {
       if (PositionTypeID == 'other') {
         newPositionType.Department__c =
           position.Department__c;
         insert newPositionType;}
       }
```

The first line of code sets the Department__c field in the newPositionType record to the value of the Department__c field in the position record. You need to add the appropriate value for department, since this value populates the selectList used in your Visualforce page.

The second line of code inserts the values in the newPositionType record into their permanent home in the Position_Type__c object.

Setting the New Position Type Lookup Value

But simply storing the new position type is not enough to satisfy the needs of your Position__c record. Remember that the Position_Type__c field in the Position__c object is defined as a lookup relationship field. A lookup relationship field takes the unique ID of the related record, not the value of the Name of that record.

This unique ID is assigned when a record is created, as you are doing with the new Position_Type__c record now. Whenever you insert a record into a Force Platform data object, the in-memory representation of that object is updated to include the new ID.

5. Modify the code in your new save method add the highlighted line below:

```
public PageReference save() {
       if (PositionTypeID == 'other') {
       newPositionType.Department__c =
         position.Department__c;
       insert newPositionType;
       position.Position_Type__c = newPostionType.ID;
       }
       }
```

The code you currently have in your save method works for those cases when a user has added a new Position Type. What about those cases when a user has simply selected an existing Position Type?

6. Add the highlighted else code to your savemethod.

```
public PageReference save() {
       if (PositionTypeID == 'other') {
         newPositionType.Department__c =
           position.Department__c;
         insert newPositionType;
         position.Position_Type__c = newPostionType.ID;
       }
       else {
         Position.Position_Type__c = positionTypeID;
         }
       }
```

This condition simply transfers the value from the positionTypeID into the Position_Type__c field.

Saving the Position Record

Now, did you forget anything? Oh yes, you still have to save the Position__c record, the main focus of the page.

You don't want to do anything extraordinary with this record—simply take the standard actions to save the record. You can simply call the default `save` action for the standard controller to accomplish this task.

Except for one little problem. Remember, the `save` method in your extension overrides the `save` method for the standard controller. If you call the default `save` method, you find your code in an endless loop.

You can escape this problem by declaring another instance of the standard controller and then calling the `save` method on that instance.

7. Add the following code to the beginning of your class

```
private final ApexPages.standardController theController;
```

8. Add the highlighted line of code to your `save` method –

```
public PageReference save() {
  if (PositionTypeID == 'other') {
       newPositionType.Department__c =
        position.Department__c;
       insert newPositionType;
       position.Position_Type__c = newPostionType.ID;
       }
  else {
    Position.Position_Type__c = positionTypeID;
    }
  return theController.save();
}
```

This final line calls the save method on the new instance of the controller, which saves the current record and returns a `PageReference`, which is, in turn, returned to the calling page.

You have completed all of the proactive work for the new feature of your Force Platform application, but you still have to enable your application to react to any unexpected problems that might arise.

Catch Those Errors

Whenever your application attempts to write data to the Force Platform database, there is a chance that an error can occur during the process. For instance, even if you have created an exquisitely perfect data interaction, someone else's interaction with the underlying data might throw everything off.

For this reason, always include error handling code whenever your application writes to the database. This code gives you a way to handle the most difficult part of quality assurance for any application—the ability to fail (somewhat) gracefully.

9. Modify the code in your `save` method to match the code below by adding the highlighted code:

```
public PageReference save() {
       if (PositionTypeID == 'other') {
       try{
         newPositionType.Department__c =
              position.Department__c;
            insert newPositionType;
         position.Position_Type__c = newPostionType.ID;
         }
       catch (DmlException e) {
         ApexPages.addMessages(e)
         }
       }
 else {
  Position.Position_Type__c = positionTypeID;
  }
       return theController.save();
       }
```

The *try/catch method* on page 316 was described in the previous chapter on database interaction with Apex code. This syntax adds error handling to your Force Platform database interaction. If the database operation goes smoothly, no `DmlException` is thrown, so no message is received.

Without a **DmlException** handler, an error would cause a standard, generic error page to appear, which may be confusing to the user, and an email is sent to the owner of the trigger. With this exception handler, you will want to add the messages to the message collection, and then display that collection in an apex:messages component on the page.

The code above catches the errors and adds them to the `ApexPages.Message` variable. Your finishing step will be to add an `apex:message` component to your page to receive those messages.

10. Return to your VisualforceExtension page. Add the highlighted code near the top of the page:

```
<apex:pageBlock mode="edit" id="thePageBlock">
  <apex:pageMessages />
  <apex:pageBlockButtons >
```

As mentioned earlier, you should always include a `pageMessages` component in all your Visualforce pages, so hopefully this step is simply a reminder of something you have already done.

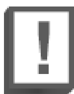
With this final important step, you have completed adding new functionality to your Force Platform application. You have explored using Visualforce pages with standard controllers and extending those controllers with some Apex code. For your last Visualforce task, you create your own custom controller that entirely replaces the functionality of a standard controller.

Visualforce Custom Controllers

You have accomplished a lot with Visualforce–shaping the look and feel of your Visualforce pages, using standard controllers to implement rich functionality, and extending those controllers when something a little extra is required. And your guiding philosophy for the Force Platform should be to always use as much default functionality as you can.

But Visualforce can take you even further. You can create your own custom controllers to back up your Visualforce pages. A controller extension extends or overrides standard controller functionality with a little bit of Apex code in a class. A custom controller completely replaces a standard controller with an Apex class.

In this final section on Visualforce, you create a custom controller to implement a wizard-like interface on the Force Platform. In this example, several different Visualforce pages access the same custom controller. Users navigate between pages, but the custom controller maintains the data over multiple Visualforce pages.



Important: Saving state

Normally, when a controller redirects the user to another page, the state for the controller is flushed. The only way to avoid this is to use the same controller for multiple pages, with that controller handling the forwarding to different pages.

The wizard you will implement gives users a fast path to creating an interview schedule for a candidate for a position. The user selects from a list of positions, sees the candidates for that

position, and then selects an interview schedule for that candidate, using three Visualforce pages and a custom controller.

Creating a Custom Controller

Your first step is to create the skeleton of the custom controller, which is an Apex class.

1. Go to **Setup ➤ Develop ➤ Apex Classes** and click **New**.
2. Enter the following code for the class, and do a **Quick Save** to valid your syntax and save the code:

```
public class customController {

}
```

   This basic code simply creates the Apex class identified as the controller for the Visualforce pages.

3. To support the first page, which you create in the next part, add the following code, which is also available from the Code Share project for this book:

```
public class customController {

public List<selectOption> openPositionList {
  get {
    List<selectOption> returnOpenPositions =
     new List<selectOption>();
    String positionInfo;
    for (Position__c p : [select name, job_description__c,
      department__c, position_type__c, status__c
      from Position__c where status__c = 'Open']) {
        positionInfo = p.name + ' - ' + p.department__c
         + ' - ' + p.job_description__c;
        returnOpenPositions.add(new selectOption(p.ID,
          positionInfo));
      }
    return returnOpenPositions;
    }
    set;
 }

public Position__c selectedPosition {
  get {
    if(selectedPosition == null) {
      selectedPosition = new Position__c();
      }
    return selectedPosition;
    }
    set;
}
```

```
public PageReference step2() {
  return null;
}

public PageReference step3() {
  return null;
}

}
```

The controller returns a list of `selectOptions`, with the value being the ID of the selected Position records and the label concatenated from several of the fields.. The SOQL query to populate this list has a `where` clause, limiting the positions displayed to those with a `Open` status, as the process flow at Universal Containers will only allow interviews to be set up for any open positions.

The final property defined in the current code, `selectedPostion` receives the position selected by the user in the first page of the wizard, which will be used to limit the selection in the second page of the wizard. The property uses an `if` statement to initialize itself, if this has not already been done, which is determined by checking for the property having a null value.

The class also contains two methods, `step2` and `step3`, used to navigate to the second and third pages of the wizard. These methods return a `PageReference`. When a `PageReference` is returned, the page referenced by the URL is returned to the browser from this function. The `PageReference` data type automatically uses the correct URL for the current Force Platform instance as the prefix for the URL, so returning null will return the user to the main page for the instance.

Right now, these methods cannot define the `PageReferences` to which the method navigates since these page references refer to Visualforce pages that are not yet created. You will modify these methods once you create the second and third pages of the wizard.

This small amount of code is all that you need to power the first page of the wizard. You are still be able to use all the default functionality in Visualforce page components, as you will see in the next section.

Displaying Positions

The first page of your wizard displays the Positions records retrieved by the SOQL statement you just created in your custom controller. Since you have had some experience defining Visualforce pages already, you will recognize almost everything in the code for this page:

```
<apex:page controller="customController">
    <apex:sectionHeader title="Interview Scheduler"
      subtitle="Step 1 of 3 - Select Position"/>
    <apex:form >
        <apex:pageBlock mode="edit">
            <apex:pageBlockSection columns="1">
                <apex:pageBlockSectionItem >
                    <apex:outputLabel
                      value="Select an open position: "
                      for="openPositionList"/>
                    <apex:selectList
                      value="{!selectedPosition.ID}"
                      id="openPositionList" size="1">
                        <apex:selectOptions
                          value="{!openPositionList}"/>
                    </apex:selectList>
                </apex:pageBlockSectionItem>
            </apex:pageBlockSection>
            <apex:pageBlockButtons location="bottom">
                <apex:commandButton
                  value="Get candidates" action="{!step2}"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

You have already encountered the `pageBlock`, `sectionheader` and `pageBlockTable` in this chapter and *Chapter 9: Visualforce Pages*. The only new aspect of this page is in the initial page tag. Instead of having a `standardController` attribute, the page has a `controller` attribute that points to the name of the Apex class acting as the custom controller for the class.

This page has been loaded into your organization as part of the initialization procedure for this chapter, as described at the beginning of this chapter, under the name of `wizard1`.

Run this `wizard1` page by appending `/apex/wizard1` onto the base URL for your instance, which should be `http://c.instance_name.visual.force.com`, where *instance_name* is replaced with the name of your Force Platform instance. You should see the page shown below.

Figure 196: The first page of your wizard

Your Visualforce page looks just like other Visualforce pages—the only person who knows that there is a custom controller at work is you. This result is nothing but positive. You can supplement or replace standard controller functionality without changing anything in your Visualforce page, aside from pointing the page to a different controller.

Displaying Candidates

The first page of your wizard for scheduling interviews is complete. This page presented a list of open job applications, and gave your user a way to select one of them. Your next step is to first add code to your custom controller for delivering data to the second page of your wizard, and then creating that second Visualforce page.

1. Return to edit the custom controller through the **Setup** ➤ **Develop** ➤ **Apex Code**, and then click **Edit** for the custom controller class.
2. Add the following code to the controller class:

```
public List<selectOption> jobApplicationCandidates {
  get {
    List<selectOption> candidates = new List<selectOption>();
    for (Job_Application__c JA :
      [select Candidate__r.ID, Candidate__r.first_name__c,
        Candidate__r.last_name__c, status__c
        from Job_Application__c
        where (Position__r.ID = :selectedPosition.ID)
        and (Status__c = 'Interviewing')] ) {
          candidates.add(new
            selectoption(JA.ID,
              JA.Candidate__r.First_Name__c
            + ' ' + JA.Candidate__r.last_name__c));
     }
     return candidates;
     }
     set;
}
```

```
public ID selectedJobApplication {get; set;}
```

This code creates a `List` object that is returned to the second Visualforce page of your wizard. You can return an array of Candidate__c records, as you did in the previous page, and use the same `pageBlockTable` structure as you did in that page. This example uses this second option to illustrate a different possible approach.

As part of this approach, you also need to define an ID variable, `selectedJobApplication`. The `selectList` in the Visualforce page binds to this variable, whose value is used to limit the records retrieved for the next page.

3. Save the new version of the custom controller code.
4. The Visualforce page for the second step of the wizard was loaded into your organization with the initialization script at the start of this chapter under the name `wizard2`. The code for this page is shown below:

```
<apex:page controller="customController">
    <apex:sectionHeader title="Interview Scheduler"
      subtitle="Step 2 of 3 - Select Candidates"/>
    <apex:form >
        <apex:pageBlock >
            <apex:pageBlockSection columns="1">
                <apex:selectRadio
                  value="{!selectedJobApplication}"
                  layout="pageDirection">
                    <apex:selectOptions
                      value="{!jobApplicationCandidates}"/>
                    <apex:actionSupport event="onchange"
                      rerender="buttons"/>
                </apex:selectRadio>
            </apex:pageBlockSection>
            <apex:pageBlockSection id="buttons">
              <apex:commandButton
                value="Schedule interviews"
                action="{!step3}"
               disabled="{!selectedJobApplication = NULL}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

You can see a new Visualforce component at work in this page. The `radioList` component provides a group of values with a radio button, limiting the choice to a single selection. The `selectList` on the first page of the wizard also limited the selection to a single choice, but there were potentially too many values to be easily displayed on the page. For the candidates on this page, a group of radio buttons is more appropriate.

A `radioList` does not have to have a value selected, so this page uses the same technique used in your previous page for avoiding user errors. The third page of this wizard lists the interviewers for a particular candidate, so you do not want your user to be able to move to that page until they select a candidate. You can stop the user from trying to perform this invalid action by disabling the `commandButton` that navigates to the third page until the value for `selectedJobApplication`, the ID variable that holds the value for the selected candidate, is not null.

This simple attribute prevents an error, but you also have to add the `actionSupport` code to the radio group so that any change in value for that group rerenders the `pageBlockSection` that contains the `commandButtons`. The buttons are redrawn whenever the value for the group of radio buttons changes, and if a radio button is selected, the `commandButton` for navigation to the final page of the wizard is enabled.

5. Run the page now, with `/apex/wizard2` appended to the base URL to see that it looks like the figure below.

Figure 197: The second page of your Visualforce wizard

6. Return to edit your custom controller code. Change the method called `step2` to return the name of this second page, following the class name of `Page`. For instance, if your second page is called `wizard2`, the correct code for the `step2` method will be as follows:

```
public PageReference step2() {
        return Page.wizard2;
    }
```

7. Save your controller code.

You can now test the first two pages of your wizard by loading the first page, selecting a position, and then clicking **Select Candidate**.

You have provided your users with two Visualforce pages that lead them through the process of selecting a closed position and then a candidate for that position. The final page in your wizard provides a way to schedule interviews on a particular day with a time assigned for each interviewer.

Displaying Interviewers

Once again, create the final page of your wizard in the controller code.

This time, add two new elements to your controller code, and handle them slightly differently in your Visualforce page.

1. Return to edit the custom controller through the **Setup ➤ Develop ➤ Apex Code** choice, and then click **Edit** for the controller class.
2. Add the following code to the controller class:

```
public Interview__c intDate {
  get {
    if(intDate == null) {
      intDate = new Interview__c();
    }
    return intDate;
    }
    set;
}

public List<Interview__c> jobApplicationInterviewers {
  get {
    if(jobApplicationInterviewers == null) {
      jobApplicationInterviewers =
        [select Interview__c.ID, Interviewer__r.Name,
        Interview__c.Interview_Date__c,
        Interview__c.Interview_Time__c
        from Interview__c
        where Job_Application__r.ID =
        :selectedJobApplication];
        }
      return jobApplicationInterviewers;
  }
  set;
}
```

Once again, you add two new variables to the controller code. The first is an instance of the Interview__c object. You use this to hold a value for the date of the scheduled interviews. Since the value entered for this date is added to all the interview records, you do not want to bind the `inputField` on the Visualforce page to the collection of Interview__c records. As with the earlier property, this property performs an `if` test to determine if initialization is needed.

The second variable declares and populates that collection of Interview__c records that are related to the candidate for the position chosen on the previous page. The ID for the Job Application record is the value of the ID chosen in the selectList on the second page of the wizard, as it should be.

With these changes, you are ready to add the final Visualforce page for your wizard. The code for this page is shown below, and was loaded into your organization under the name of `wizard3` with the initialization script for this chapter:

```
<apex:page controller="customController">
    <apex:sectionHeader title="Interview Scheduler"
      subtitle="Step 3 of 3 - Schedule interview times"/>
    <apex:form >
        <apex:pageBlock >
            <apex:pageBlockSection columns="1">
                <apex:pageBlockTable
                  value="{!jobApplicationInterviewers}" var="int">
                    <apex:column headervalue="Interviewer"
                      value="{!int.Interviewer__r.Name}"/>
                    <apex:column headervalue="Interview Time">
                        <apex:inputField
                          value="{!int.Interview_Time__c}"/>
                    </apex:column>
                </apex:pageBlockTable>
            </apex:pageBlockSection>
            <apex:pageBlockSection >
                <apex:pageBlockSectionItem >
                    <apex:outputLabel
                      value="Date for interviews:"/>
                    <apex:inputField
                      value="{!intDate.Interview_Date__c}"/>
                </apex:pageBlockSectionItem>
            </apex:pageBlockSection>
            <apex:pageblockButtons location="bottom">
                <apex:commandButton action="{!Save}"
                  value="Save"/>
            </apex:pageblockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

This Visualforce page is slightly more complex than the previous pages. There are two `pageBlockSections`: one to hold the display of Interviewers in a `pageBlockTable` and another to specify the date for al the interviews. This paricular implementation assumes that all interviews will be conducted on the same day, so a user can specify a date once for all the interviews. The code you create for your save method adds this value to all of the interviews listed in this page.

3. Return to your controller code and edit the `step3` navigation method to point to this third page. Once you save the third page as `wizard3`, the completed method looks like this:

```
public PageReference step3() {
        return Page.wizard3;
    }
```

4. Run your wizard from the first page to the last to see how it flows. The third page, with data, should look like the figure below.

Figure 198: The third page of your Visualforce wizard

You have now implemented the user interface portion of your wizard, along with the custom controller code to make them work nicely together. Your last task is to complete the work that your user specified with the wizard by saving the interviews with their date and time.

Saving the Data

The last piece of functionality for your custom controller is the most important. The save method for a standard controller saves the changed records back to the Force Platform database object. The save method for this custom controller must first set the date selected on the third page as the value for the Interview_Date__c field for all appropriate interview records, and then save those records.

The code for this save method is shown below:

```
public PageReference save() {
  for (Interview__c i : jobApplicationInterviewers) {
    i.Interview_Date__c = intDate.Interview_Date__c;
    }
  update jobApplicationInterviewers;
  return new ApexPages.StandardController(new
    Job_Application__c(id = selectedJobApplication)).view();
}
```

The code first walks through the array of Interviewer records to add the date from the Interview Date inputField at the top of the page to any records that contain a value for the Interview_Time__c field. If an Interview record does not have a time specified, there is no

reason to add a value for the Interview_Date__c field. Once you have properly set this value in the array, you can simply update the array.

Once the update is performed, the `return` sends back a page reference derived from the `ApexPages` class. The page returned is a view page for a standard controller for a Job Application record. The Job Application record displayed in the page is identified with the ID of the `selectJobApplication`, which was the focus of the update operation.

As usual, bracket your write operation with `try/catch` code, in the event that there is a problem with the update.

1. Open the Apex class for the custom controller.
2. Add the code for the save method to your custom controller.

The complete code for your custom controller class is included as a document in the Code Share project for this book and chapter under the name CustomController.txt, and is shown below:

```
public class customController {

public List<selectOption> openPositionList {
  get {
    List<selectOption> returnOpenPositions =
      new List<selectOption>();
    String positionInfo;
    for (Position__c p : [select name, job_description__c,
      department__c, position_type__c, status__c
      from Position__c where status__c = 'Open']) {
        positionInfo = p.name + ' - ' + p.department__c +
        ' - ' + p.job_description__c;
        returnOpenPositions.add(new
          selectOption(p.ID, positionInfo));
      }
    return returnOpenPositions;
    }
    set;
 }

public Position__c selectedPosition {
  get {
    if(selectedPosition == null) {
      selectedPosition = new Position__c();
      }
    return selectedPosition;
    }
    set;
}

public List<selectOption> jobApplicationCandidates {
  get {
    List<selectOption> candidates =
```

```
      new List<selectOption>();
    for (Job_Application__c JA : [select Candidate__r.ID,
      Candidate__r.first_name__c, Candidate__r.last_name__c,
      status__c
      from Job_Application__c
      where (Position__r.ID = :selectedPosition.ID) and
      (Status__c = 'Interviewing')] ) {
        candidates.add(new selectoption(JA.ID,
          JA.Candidate__r.First_Name__c + ' ' +
          JA.Candidate__r.last_name__c));
     }
     return candidates;
     }
     set;
}

public ID selectedJobApplication {get; set;}

public Interview__c intDate {
  get {
    if(intDate == null) {
      intDate = new Interview__c();
    }
    return intDate;
    }
    set;
}

public List<Interview__c> jobApplicationInterviewers {
  get {
    if(jobApplicationInterviewers == null) {
      jobApplicationInterviewers = [select Interview__c.ID,
        Interviewer__r.Name, Interview__c.Interview_Date__c,
        Interview__c.Interview_Time__c
        from Interview__c
        where Job_Application__r.ID = :selectedJobApplication];
        }
      return jobApplicationInterviewers;
  }
  set;
}

public PageReference step2() {
  return Page.wizard2;
}

public PageReference step3() {
  return Page.wizard3;
}


public PageReference save() {
  for (Interview__c i : jobApplicationInterviewers) {
    i.Interview_Date__c = intDate.Interview_Date__c;
    }
  update jobApplicationInterviewers;
```

```
  return new ApexPages.StandardController(new
    Job_Application__c(id = selectedJobApplication)).view();
}

}
```

Of course, you might want to implement even more logic in your custom controller. For instance, you might want to add code to check and make sure that none of the interviews occurred at the same time, or that every interview had a time scheduled.

Tip: An enhanced version of the custom controller code with this functionlaity is available in the Code Share project for this book.

This type of logic is simply more Apex code, which, as a developer, you should be able to implement to match your particular use cases.

Integrating Your Wizard

Your wizard should now be working properly. The custom controller maintains data values across multiple pages and handles the logic needed to properly apply the simplified user interface to the appropriate records. In this particular use case, your users want to be able to access this interview scheduling capability directly from a tab.

1. Go to **Setup** ➤ **Create** ➤ **Tabs**.
2. Click **Create** in the Visualforce Tabs section.
3. On the next page, specify a tab label and tab name. Give your tab a `Label` of `Interview Wizard` and accept the default tab name. Select a tab style and give the new tab a brief description. Click **Next**.
4. Accept the default security settings and click **Next**.
5. Uncheck the `Include Tab` checkbox for all applications except your Recruiting application and click **Save**.

The new tab appears as part of the tab set in your application. Clicking the tab brings you to the first page of your interview wizard, just the way your users wanted.

Summary

Over the course of *Chapter 9: Visualforce Pages* and this chapter, you have had a quick tour of the capabilities of Visualforce technology. You learned how you can implement flexible and

robust user interface pages with standard controllers, and how you can expand your options even more with extensions to standard controllers and custom controllers. You also had a brief look at creating your own Visualforce components, which can improve your productivity by eliminating multiple implementations of the same interface.

You can go even further with Visualforce and Apex, as you will learn in the next chapter, which covers how you can allow people to use email to communicate to your Force Platform data objects, and how to use Visualforce pages to create email templates and Adobe PDF documents.