Creating On-Demand Applications with AppExchange
An Introduction
Creating On-Demand Applications with AppExchange
An Introduction
Table of Contents

Preface ...........................................................................................................................................7
Welcome to On-Demand Computing ..............................................................................................8
Making the Most of the AppExchange Developer Network............................................................8
About This Book .............................................................................................................................10

Chapter 1: Introducing the AppExchange Platform ......................................................................11
Business Apps as a Service ...............................................................................................................12
Data-Centric Apps ........................................................................................................................12
Collaborative Apps .........................................................................................................................13
Technologies behind the AppExchange ........................................................................................13
Multitenancy ..................................................................................................................................14
Metadata-Driven Development .......................................................................................................15
Web Services ..................................................................................................................................16
AppExchange Directory ..................................................................................................................17

Chapter 2: About the Sample Recruiting App .............................................................................19
About Universal Containers .............................................................................................................20
Considerations for the Recruiting App ............................................................................................20
Building the App: Our Design .........................................................................................................20
Native Components ........................................................................................................................21
Composite Components .................................................................................................................23

Chapter 3: Reviewing Database Concepts ....................................................................................25
What’s a Database? ..........................................................................................................................26
What’s in a Database? .......................................................................................................................26
What’s a Relational Database? .........................................................................................................27
Summary of Database Concepts .....................................................................................................29

Chapter 4: Building a Simple App .................................................................................................31
Becoming Familiar with the Setup Area ..........................................................................................32
Introducing Apps ............................................................................................................................33
Try It Out: Defining an App .............................................................................................................33
Look at What We’ve Done .................................................................................................................34
Introducing Objects .........................................................................................................................35
The Position Custom Object ............................................................................................................36
### Table of Contents

Try It Out: Defining the Position Custom Object..............................................................36

Introducing Tabs.............................................................................................................................38

Try It Out: Defining the Positions Tab...........................................................................................39

Look at What We’ve Done................................................................................................................41

Becoming Familiar with Setup Detail Pages and Related Lists......................................................41

Introducing Fields...........................................................................................................................43

Try It Out: Adding Text Fields.........................................................................................................44

Try It Out: Adding Currency Fields.................................................................................................46

Try It Out: Adding a Checkbox Field...............................................................................................46

Look at What We’ve Done................................................................................................................46

**Chapter 5: Enhancing the Simple App with Advanced Fields and Page Layouts**.................49

Adding Advanced Fields................................................................................................................50

Introducing Picklists........................................................................................................................50

Introducing Field Dependencies.....................................................................................................52

Introducing Custom Formula Fields................................................................................................55

Introducing Page Layouts................................................................................................................59

Becoming Familiar with the Page Layout Edit Page.........................................................................59

Try It Out: Grouping Fields into a New Section.............................................................................60

Try It Out: Editing Field Properties...............................................................................................62

Look at What We’ve Done................................................................................................................62

**Chapter 6: Expanding the Simple App Using Relationships**.................................................65

Introducing Relationships................................................................................................................66

Introducing Lookup Relationship Custom Fields............................................................................66

Try It Out: Relating Hiring Managers to Positions.........................................................................67

Look at What We’ve Done................................................................................................................67

Adding Candidates to the Mix............................................................................................................68

Try It Out: Creating the Candidate Object.....................................................................................68

Try It Out: Adding Fields to the Candidate Object..........................................................................69

Try It Out: Modifying the Candidate Page Layout Properties.........................................................70

Look at What We’ve Done................................................................................................................71

Creating a Many-to-Many Relationship..........................................................................................71

Try It Out: Creating the Job Application Object.............................................................................73

Try It Out: Adding Fields to the Job Application Object.................................................................73

Look at What We’ve Done................................................................................................................74

Managing Review Assessments.......................................................................................................75

Try It Out: Creating the Review Object..........................................................................................76

Try It Out: Adding Fields to the Review Object.............................................................................77

Look at What We’ve Done................................................................................................................77
Chapter 7: Securing and Sharing Data

Controlling Access to Data in Our App
- Required Permissions for the Recruiter
- Required Permissions for the Hiring Manager
- Required Permissions for the Interviewer
- Required Permissions for the Standard Employee
- So Where Are We Now?

Data Access Concepts

Controlling Access to Objects
- Introducing Profiles
- Standard AppExchange Profiles
- Custom Profiles in Our Recruiting App
- Try It Out: Creating the Recruiter Profile
- Try It Out: Creating More Profiles

Controlling Access to Fields
- Introducing Field-Level Security
- Field-Level Security in Our Recruiting App
- Accessing Field-Level Security Settings
- Try It Out: Restricting Access to a Position's Minimum and Maximum Salary Fields
- Try It Out: Restricting Access to a Candidate's Social Security Number

Controlling Access to Records
- Introducing Organization-Wide Defaults
- Introducing Role Hierarchies
- Introducing Sharing Rules
- Introducing Manual Sharing
- Putting It All Together
- Try It Out: Creating Users for Our Recruiting App
- Try It Out: Verifying that Everything Works

Chapter 8: Using Custom Workflow

Introducing Workflow
- Introducing Workflow Rules
- Introducing Workflow Actions: Tasks, Field Updates, and Alerts
- Workflow in Our Recruiting App
- Try It Out: Creating Workflow Rules That Assign Tasks
**Chapter 9: Moving Beyond Native Apps**

- Introducing Web Services .............................................................. 132
- Introducing the AppExchange API .................................................. 132
  - Common API Calls .................................................................. 132
  - SOAP, XML, and the AppExchange API ......................... 133
  - Authenticating Calls to the AppExchange API ...................... 133
- Implementing a Composite Application Piece: The Candidate Mapping Tool ... 135
  - Introducing S-Controls ......................................................... 135
  - Try It Out: Defining the Candidate Mapping S-Control .... 135
  - Introducing Hooks and Targets .............................................. 140
  - Try It Out: Creating a Custom Link ................................... 141
  - Try It Out: Adding a Custom Link to a Page Layout .......... 142
  - Try It Out: Running Our S-Control .................................. 142

**Chapter 10: Learning More**

- Help and Training Options ............................................................ 146
- AppExchange Podcasts ................................................................. 146
- AppExchange Partner Program .................................................... 147

**Chapter 11: Things to Come: A Sneak Preview of Apex**

- Introducing Apex .......................................................................... 151
- Sample Schema Used in This Chapter ........................................ 152
- A Sample Apex Script ................................................................. 152
- Basics .......................................................................................... 154
- Statements .................................................................................. 162
- Methods ...................................................................................... 169
- Apex API Calls, Packages, Triggers, and Debugging .............. 177

**Glossary** ......................................................................................... 183
As users of the Internet, we're all familiar with the fascinating, innovative, creative, and sometimes silly ways in which it has changed how we work and how we play. Much of that excitement comes from the fact that how you create content for the Internet is as different from the "traditional" application development models as how you consume it. From websites, to wikis, to blogs, and more, it's exciting to watch the innovations taking place that are changing how we communicate and collaborate.

These changes have certainly impacted how we work with content. We can find a website or blog on almost any conceivable subject, at lightning speeds. A similar set of ideas and technologies is changing how we work with (and build!) business applications. These new business applications range from the customer databases and order systems common to most companies all the way to the unique programs that are only applicable to one company. Since business applications are so much easier to create, use, and run, we are starting to apply them to new kinds of business problems that, because of complexity or cost, have remained out of reach. Just like the changes that moved publishing technology from paper to bits made it possible for us to have information about anything in the whole world right at our fingertips, the changes in application technology make it similarly possible to imagine a robust, enterprise-class application for almost any business need. Sound pretty good? Then you're probably wondering: "What's the catch? What's the magic that makes this possible?"
Welcome to On-Demand Computing

These new ways of building and running applications are enabled by the world of on-demand computing, where you access applications, or apps, over the network as a utility, rather than as a piece of software running on your desktop or in the server room. This is already quite common for many consumer apps like email and photo sharing, and for certain business applications, like Customer Relationship Management (CRM). Because almost all apps these days are delivered via a Web browser, it’s increasingly hard to tell (from both an experience and a functionality point of view) which applications are "traditional software," and which are being run—and paid for—as a utility service across the Internet. As with the Internet, on-demand applications have grown so ubiquitous that almost every business user interacts with at least one, whether it's an email service, a Web conferencing application, or a sales system.

A new twist, the on-demand platform, is making the delivery of functionality even more interesting. Increasingly, on-demand applications and—even consumer services—are starting to look less like websites and more like platforms, meaning they are starting to sprout Application Programming Interfaces (APIs), code libraries, and even programming models. Collectively, these new kinds of development technologies can be thought of as on-demand platforms, similar to traditional platforms in that they are designed to be reused and built on to create new functionality, but accessed over the Internet rather than using a local platform or API. The possibilities presented by this new type of platform have emerged quickly, spurred by the popularity of mash-ups—the technique of combining multiple Internet services to create new functionality. These new capabilities include innovations like Google's search API, which allows developers to use the power of that search engine in their applications, eBay's APIs for auctions and listings, or Amazon's system for creating entirely new storefronts. For example, almost any real estate website or application these days is using Google or Yahoo maps under the hood, illustrating how these new APIs are now commonly running alongside the more traditional database, app server, or operating system platforms.

The new ways you can plug into websites are exciting and essential, but they are really just pieces of the much larger picture of on-demand platforms, or what people are now calling the programmable Web. Not only can you reuse or mash up Internet services in new and interesting ways, but these Web services themselves are becoming customizable and programmable. In a way, this is the database analogue to My Yahoo’s content model; just as you can easily configure a page to display only the headlines or feeds that you find interesting, so should you be able to modify the structure and behavior of an on-demand business application, or create new ones, using just a browser.

Making the Most of the AppExchange Developer Network

This book provides you step-by-step instructions on how to build your first AppExchange app. It will show you the power of the AppExchange platform, and we have the highest of hopes that it will become a trusted reference on your office bookshelf. But this book can give you only a glimpse of what is possible. There's a lot more to the platform! As you learn more about app building on AppExchange, and you become more ambitious about the sorts of apps you want to build, you’ll want more information, resources, and ongoing guidance. That’s where the AppExchange Developer Network (ADN) fits in. ADN is the place to be for AppExchange developers.

The centerpiece of the AppExchange Developer Network is the website at www.salesforce.com/developer. It provides a full range of resources including sample code, toolkits, an online developer community, and the test environments necessary for building apps. It includes an online version of this book (written by ADN staff members) and has information about the ADN@Dreamforce event that we hold every year for AppExchange developers. You should definitely bookmark the ADN home page. If you need more info, have a question to
ask, are seeking a toolkit or sample, or just want to dig a little deeper into AppExchange development, this is where it all comes together.

Let's take a quick tour:

• Get the latest AppExchange news.

  www.salesforce.com/developer is where you’ll find all the AppExchange developer community news, whether it’s the latest on platform technologies, a new toolkit release, an upcoming developer event, or a blog post from a salesforce.com AppExchange technology expert.

• Get a Developer Edition account.

Take your application-building skills to the next level. A Developer Edition account is a free, fully-functional salesforce.com account that allows you to get hands-on experience with all aspects of the platform in an environment designed for development. If you haven’t done so already, go to the ADN home page and create a Developer Edition account for yourself. Then use it to try out new application ideas and to test various AppExchange tools and technologies.

• Get the resources you need.

Of course, a big part of ADN’s mission is to ensure that you and every other AppExchange developer has the support necessary to successfully build great apps. To this end, you’ll want to know about the Documents section of the ADN site—www.salesforce.com/developer/docs.jsp is the place to turn for official, authoritative reference information for developers.

For more hands-on resources, check out www.salesforce.com/developer/projects_toolkits.jsp, the Projects & Toolkits section of the site. Here you’ll find tools, code samples, and other developer packages created by the ADN team for the express purpose of enabling development on the AppExchange platform. Use the AppExchange API and toolkits for .NET, Java, PHP, Ruby on Rails, and AJAX to combine AppExchange data and services with your own data and functionality to create unique new on-demand applications.

Additional resources such as white papers and webinars demonstrate how the AppExchange platform can be part of a successful service-oriented architecture that allows IT departments to create uniquely valuable solutions for their companies.

• Keep up with the ADN community at www.salesforce.com/developer/community/index.jsp.

Your fellow developers of all levels of sophistication are maintaining lively conversations every day on the ADN discussion boards. You’re welcome to join in by posting questions and comments of your own, or you can just read along and learn. The free monthly ADN newsletter provides a recap of the most interesting news and information, delivered right to your email inbox.

Check out the ADN blog at www.salesforce.com/developer/blogs.jsp. The ADN team shares some of what it’s learned in the process of creating and managing more than 300 apps on the AppExchange directory. This blog thrives on feedback and the cross-pollination of ideas. We welcome your comments.

• Take part in ADN events and activities.

ADN also sponsors various special events and activities, like the very popular AppExchange Seminars tour, and our biggest event of the year, ADN@Dreamforce—the conference within a conference for AppExchange application builders. Information on upcoming events is posted on the ADN site and is included in the ADN newsletter.
About This Book

To help illustrate the technologies and concepts of the AppExchange platform, and to demonstrate a data-centric and collaborative app well-suited to the platform, this book will walk you through the process of creating a recruiting application (the details of which are described in Chapter 2: About the Sample Recruiting App on page 19).

Beyond the features you’ll learn about in the following chapters, there are a few other ideas you'll hopefully keep in mind. First, you'll quickly experience how productive developing in this new model can be, and you'll be able to see the opportunities it creates. Second, in addition to being more productive, building on the AppExchange platform is designed to be engaging, rewarding, and most importantly, fun—and hopefully that will carry through not just how you build your apps, but also what you create in the end. Go to the AppExchange Developer Network at www.salesforce.com/developer and click Sign Up Now to access the free, fully-functional salesforce.com Developer Edition so you can use it to follow along with the exercises in this book, play around, and experience the features of the AppExchange platform directly.

This book focuses primarily on the building aspects of AppExchange development and will be easily understood by anyone from a business user to a professional developer. We're assuming that you have a basic familiarity with the salesforce.com Sales Force Automation app, as well as with basic relational database concepts. The final chapter will provide an introduction to the Web services API and how it can be used to build composite apps; to get the most out of that chapter, you should be familiar with HTML and JavaScript. However, even if you're not an experienced developer, you can still follow along to gain a deeper understanding of what can be done with the AppExchange platform. Check out the Glossary on page 188 if you're unfamiliar with any terminology.

Note: This book contains lots of screenshots. Since the AppExchange is a rapidly developing platform, the screenshots might vary slightly from what you see on the screen; these changes should be minor and shouldn't affect your understanding of the system.

An online version of this book is available on the AppExchange Developer Network website. Check here for additional content and updates: www.salesforce.com/developer/devguide.

We welcome your comments. The AppExchange Developer Network counts on your feedback and ideas. Go to the ADN discussion boards at www.salesforce.com/developer/community/index.jsp and let us know what you'd like to see in the next version!
Chapter 1

Introducing the AppExchange Platform

In this chapter ...

• Business Apps as a Service
• Technologies behind the AppExchange

AppExchange is a new service from salesforce.com that takes advantage of these new ways of delivering and developing services to make building, sharing, and running business applications simpler and more powerful than has previously been possible. At the heart of this service are two key capabilities: the AppExchange platform and the AppExchange directory. The AppExchange platform is the on-demand platform where you create and run your apps, and the AppExchange directory is where you share those apps.

While this book will focus on building apps on the platform, you might also want take a quick tour of the AppExchange directory at www.appexchange.com to see the kinds of applications that are possible with the platform, and even how some companies are creating commercial applications for distribution via that site.
Business Apps as a Service

So, what exactly are these business applications the AppExchange is designed to enable, and what kinds of problems are they designed to solve? One way to look at this is to examine the most popular application for the platform, salesforce.com's Sales Force Automation app, and generalize from there. If you haven’t used Salesforce before, it would be worthwhile to spend a bit of time clicking around the basics (like entering a contact or creating a report) to familiarize yourself with the underlying concepts, because they will have a lot in common with the app you are planning to build.

As you will see, there are a few key elements that form the foundation of the Salesforce app and of most applications created with the AppExchange platform. First, across the top of an app is a set of tabs that provides a means to segment the app into its different parts, and to navigate between them. As a concept, tabs are perhaps the most crucial part of using any AppExchange app. This is also true from the developer's perspective—most of the development work revolves around creating tabs and defining the data and behaviors that support them.

A second key element is the form that is displayed as part of a tab. Forms are a key part of not only AppExchange apps but almost all business apps, regardless of age or technology. From the earliest mainframe screens, to more recent client-server systems, and on to modern websites, forms are the primary means of entering, viewing, and navigating the information in these systems. And, just as the creation of forms and the corresponding data models are key aspects of most development tools and databases, so too are they central to how you create apps for this platform. Because they are delivered via a browser, a unique aspect of AppExchange forms is that they use hyperlinks extensively. These links provide navigation both within a set of data and also out to the Web in general—a technique you’ll likely find is simple and surprisingly effective.

To better understand what the AppExchange platform is well-suited for, let’s look beyond the core elements of tabs, forms, and links and into the types of applications they enable. Two huge benefits start to come into focus when you look across these AppExchange apps in general: they are almost universally data-centric and most often collaborative.

Data-Centric Apps

Data-centric apps are fundamentally built around structured information, like the information that is typically found in databases. The data in these apps describes different parts of a business function in a regimented
way—such as the details about a person, piece of equipment, or business transaction. We can find these kinds of data-centric apps everywhere, in small desktop databases like Microsoft Access or FileMaker, all the way to the huge systems running on database management systems like Oracle or MySQL. In contrast, a collection of Microsoft Word documents or HTML files is considered unstructured, because their contents are not described in a consistent way nor can they be queried or aggregated, as is common with data. A simple exercise such as trying to determine the total sales for a month from a series of Microsoft Word-based contracts versus querying a simple database for that information illustrates that difference.

And because the ability to control and manage data is a nearly universal requirement for companies of all types and sizes, data-centric apps are most often used for business apps. In our consumer lives, our requirements for structured data are typically limited to simple cases such as a contact list, a collection of photos, or maybe the contents of our music collection; whereas the AppExchange platform is designed for creating robust data-centric apps that can meet the needs of businesses of all sizes, from small businesses all the way to the largest companies.

**Collaborative Apps**

The collaborative characteristic of these apps is a powerful aspect of the AppExchange platform. It’s difficult to access a data app across a group or a company if it resides on one person’s computer, but AppExchange apps can always be shared. This benefit makes it easy for teams to collaborate on group activities like selling a product, managing a project, or hiring an employee. The many features of the platform, including the fact that any app that runs on it is available on demand, provide a far-reaching ability to collaborate. Apps are automatically available from anywhere in the world, with just a Web browser.

Achieving this kind of ubiquity comes automatically when you use the AppExchange platform, and as many developers have personally experienced, such ubiquity is often very challenging to otherwise obtain. Beyond the AppExchange platform’s deployment model, there are a number of development features native to the platform that facilitate sharing. The platform’s security and sharing model allows you to finely control the access and permissions to different data. The workflow model allows you to initiate business workflow processes and change data ownership based on a variety of business conditions. Collectively, these features provide the framework for sharing AppExchange apps across groups, divisions, and entire corporations yet retain the controls on data that sensitive scenarios require.

Now that you have an idea of what kinds of apps the platform is designed to build, it’s useful to review how they are built. As you’ll see, there are a few technologies that make this platform unique, not only in what it supports but in how it works and, in turn, what it means to access it as a developer.

**Technologies behind the AppExchange**

Since the AppExchange is an on-demand platform, you’ll interact with it through a Web browser or Web services API. Many aspects of the AppExchange platform will be familiar to developers of any kind of business app. Concepts like databases, user interfaces, forms, business logic, and workflow that collectively represent the platform’s development model all still apply.

However, the ways in which users and developers interact with those features, and how an application is represented through those concepts, might seem entirely new. These differences are expressed most clearly by these important AppExchange concepts: multitenancy, metadata-driven development, Web services, and the AppExchange directory. The architectural concept of multitenancy is related to offering systems on demand, while metadata-driven development and Web services are changes in application development technology in general. The AppExchange directory is the revolutionary way of deploying apps.
Table 1: AppExchange Key Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multitenancy</td>
<td>An application model in which all users and apps share a single, common infrastructure and code base.</td>
</tr>
<tr>
<td>Metadata-driven development</td>
<td>An app development model that allows apps to be defined as declarative &quot;blueprints,&quot; with no code required. Data models, objects, forms, workflows, and more are defined by metadata.</td>
</tr>
<tr>
<td>AppExchange Web services API</td>
<td>An application programming interface that defines a Web service that provides direct access to all data stored in AppExchange from virtually any programming language and platform.</td>
</tr>
<tr>
<td>AppExchange directory</td>
<td>A Web directory where hundreds of AppExchange apps are available to Salesforce customers to review, demo, comment upon, and/or install. AppExchange developers can submit their apps for listing on the AppExchange directory if they want to share them with the community.</td>
</tr>
</tbody>
</table>

Multitenancy

One of the key innovations of salesforce.com is the idea of multitenant apps. In contrast to their single-tenant counterparts, such as client-server enterprise applications or email servers, multitenant applications are designed so that users share the same physical instance and version of the application. Individual "deployments" of those applications occupy virtual partitions rather than separate physical stacks of hardware and software. Multitenant applications, such as Salesforce, and platforms, such as AppExchange, are similar to consumer applications like Google Mail and eBay, each of which runs a single code base and depends on an infrastructure shared by all users. This multitenant design makes possible the quick deployment, low costs, and rapid innovation that salesforce.com has become known for.
This multitenant architecture also impacts how developers use the AppExchange platform to create new applications. Specifically, it allows for a clear boundary between the platform and the applications that run on it. While it allows applications to have their own data objects, forms, layouts, and integration, the system manages these and other customizations as abstractions. This separation of platform and the apps that run on it is what ensures that a given application can’t "behave badly," meaning it can’t encroach on other users' applications or otherwise prevent low-level aspects of the platform from being upgraded, changed, or enhanced. The developer is shielded from having to know or worry about any aspect of multitenancy. Core development on the platform (such as adding an object) occurs at a much higher level of abstraction so that the platform can handle such details automatically. The good news, as you’ll see next, is that this abstraction allows for the platform to not only take care of much of the plumbing but also to provide sophisticated features for "free," without any developer effort.

**Metadata-Driven Development**

*Metadata-driven development* is essential to the AppExchange platform. It is being used by almost all platforms, from Ruby to Windows. With a metadata-driven approach, properties of an app are declared as XML or database entries, rather than being hard-coded. You can think of these XML or database entries as analogous to blueprints,
in that they describe the app rather than actually represent it. These blueprints are then rendered by the platform, either once at compile time or on request at run time, into the living and breathing applications we expect.

Although that may at first glance seem somewhat esoteric, this is of course exactly how Web browsers work—instead of hard coding the definition of a Web page in something like Java or C#, a developer defines the page as HTML, which in itself is just a kind of metadata. When the page is requested, the browser renders the content into the interface that the user experiences. The framework, in this case the Web browser, defines and constrains what is possible to build. Using HTML (the metadata), you can't make the browser (the framework) do something it wasn't explicitly designed to do. A metadata-driven development environment is much more constrained than an open-ended development environment, but it simplifies your work and increases your productivity. It's much easier to get stuff done.

This same principle is at work in the AppExchange development model. AppExchange apps—their data models, objects, forms, workflows, and more—are defined by metadata, which is in turn rendered on request into the application the user experiences. The framework in this case is the AppExchange platform itself. And like its HTML/browser counterpart, this model is much more constrained than an open-ended environment like Java or .NET, but far simpler and incredibly more productive.

Put this productivity gain together with the concepts of on-demand and multitenancy and you can begin to see how much promise this new kind of platform holds: 1) you don't need to worry about running or maintaining any of the infrastructure of these apps; 2) you don't need to worry about how that infrastructure will scale to support even tens of thousands of users; and 3) as a developer, you'll be extremely productive.

**Web Services**

*Web services* is another core development concept of the AppExchange platform, and like metadata, it is increasingly common across all kinds of applications and platforms. In simple terms, Web services allow programmers to access and manipulate applications in a way that is independent of either an application's physical implementation (what it is written in) or its location (where it resides). It's easy to understand why this technology is so important to on-demand systems: this kind of architecture and location independence enables the Web services to run outside of a company's server room while still allowing all the integration capabilities and functionally of being physically located in the same place. And since Web services are—not surprisingly—based on Web standards, they are well suited to traverse firewalls and leverage the rest of the Internet infrastructure already in place.

Developers benefit from the mature and near ubiquitous support for Web services in tools like Java, PHP, and .NET. This frees developers to use their tools of choice instead of proprietary systems when accessing these APIs.

In AppExchange terms, the significance of Web services is twofold. First, Web services provide a straightforward, powerful, and open way to programatically access the data and capabilities of any app running on the platform, which is useful for creating integrations and utility tasks such as bulk loading data. Second and more important in framing the platform's capabilities, Web services allow developers to go beyond the metadata or native model and create behaviors that fall outside of its constraints. This opens the door to leveraging the open-ended possibilities offered by traditional programming, with the caveat that it also typically requires more work. In most cases, apps that leverage the API to create advanced functionality do so in a way that builds upon the native capabilities. For this reason, apps that leverage the AppExchange Web services API as part of their functionality are known as composite.

With the AppExchange platform, it is radically simpler and easier for developers to create collaborative business apps than it has been before. It is now possible to bring the kind of productivity benefits, which you already know from traditional applications like CRM, to a whole new class of apps and users. The AppExchange platform has
already enabled successful development and delivery of these new kinds of apps, including smaller apps like vacation time requests and expense management, all the way up to full-scale project management systems and industry-specific solutions for real estate or other areas.

**AppExchange Directory**

In addition to being able to run multiple applications, AppExchange allows you to package and share any apps created on the platform. You can upload these apps and register them on the AppExchange directory, where other companies can browse, demo, review, and install them. You can always take a quick tour of the AppExchange directory, at [www.appexchange.com](http://www.appexchange.com), to see the wide range of innovative and exciting apps that exist today. Salesforce administrators can install on-demand applications for everything from handling expense management to tracking purchasing, monitoring recruiting, and beyond. Some of these apps have been created in-house at salesforce.com, but most are built by partners and individual developers who have chosen to take advantage of the AppExchange platform.
Chapter 2

About the Sample Recruiting App

In this chapter ...

• About Universal Containers
• Considerations for the Recruiting App
• Building the App: Our Design

The goal of this book is to show you how easy it is to create powerful, multifaceted AppExchange applications that solve common business problems. To do so, let’s walk through the steps of creating a simple application for a make-believe company called Universal Containers.

Like many companies that have grown rapidly, Universal Containers has been experiencing a few growing pains, especially in its Human Resources department. In this book, we’re going to build a Recruiting app for the company that allows it to move away from the Microsoft Word documents and Microsoft Excel spreadsheets that it has traditionally used to an application that's available on demand.

By the time we finish building the Recruiting app in this book, you should feel confident enough to build a custom on-demand application that suits your own company’s needs. So let's get started!

Note: It's not the intent of this book to provide you with a full-featured Recruiting application that meets all of your business needs. Rather, we'll be using this Recruiting app as a jumping-off point for learning about what's generally included in an app.

If you do want a full-featured Recruiting app, you can either use the app that we describe in this book as a good starting point or you can check out the Recruiting section of the AppExchange directory at www.appexchange.com.
About Universal Containers

First, let's learn a little more about our fictional company, Universal Containers.

Universal Containers is a rapidly growing international supplier of container products. The company produces every kind of container from simple overnight letter mailers to custom equipment packaging to large cargo shipping containers. In addition, Universal Containers develops and maintains its own proprietary software to facilitate the design of its various types of containers. As such, Universal Containers has a very diverse group of employees, including facilities and operations professionals, software and design engineers, financial accountants, and legal and human resources personnel.

Historically, the Human Resources department has used Microsoft Word documents and Microsoft Excel spreadsheets to manage the recruiting and hiring process for new employees. However, over the last two quarters it's become evident that unless this process is replaced by one that is more collaborative, reliable, and scalable, the department won't be able to meet its hiring goals for this fiscal year. Universal Containers needs a centralized application that can bring all of its recruiting and hiring processes together, and the company has hired us to solve this problem. Our approach will be to leverage their Salesforce account and build a recruiting application on the AppExchange platform. We're going to introduce Universal Containers to the world of on-demand, custom apps!

Considerations for the Recruiting App

After meeting with Megan Smith, Universal Container's vice president of Human Resources, we've drawn up a few requirements for the new Recruiting app. The app needs to:

• Track positions in all stages of the process, from those that are open to those that have been filled, deferred, or canceled
• Track all of the candidates who apply for a particular position, including the status of their application (whether they've had a phone screen, are scheduled for interviews, have been rejected or hired, or have passed on an offer that was presented)
• Allow employees to post reviews for candidates whom they've interviewed
• Provide security for the recruiting data so that it’s not mistakenly viewed, edited, or deleted by employees who shouldn’t have access
• Automatically inform the relevant recruiter about the next steps that should be taken when a decision has been made about an applicant
• Automatically inform all employees of new positions that have been posted
• Allow recruiters to map the locations of all candidates who are applying for a position, to better understand relocation expenses

Though modest, an app that includes this functionality is going to greatly increase the efficiency of Universal Containers' recruiting and hiring processes. Let's now take a look at how we'll implement these seven requirements.

Building the App: Our Design

As application developers, our job now is to figure out which AppExchange platform components are going to allow us to build all of these features.
One way to split up the work is to look at which requirements can be implemented by using just the point-and-click tools of the AppExchange platform and which requirements must be implemented by leveraging other Web applications. The former method, which uses native components, is typically fast and simple to use, requiring only point-and-click setup rather than more traditional coding. The latter method, which uses composite components, give us more control and flexibility in what we do but requires more work.

The following diagram shows how features implemented with each method are created, split out by their user interface layer, business logic, and data model.

![Figure 3: Native Versus Composite Components](image)

Fortunately, all but one of our Recruiting app’s requirements can be implemented using the native component method—an example of why the AppExchange platform is such a powerful development environment! Now let’s go into a little more detail about what we’re going to build.

**Native Components**

Natively, there are several AppExchange platform components that are going to help us implement our Recruiting app requirements. These include:

- Custom objects
- Security and sharing rules
- Workflow processes

We’re going to learn about all of these things in a lot more detail in later chapters, but for now, let’s get a quick preview of what’s in store.

**Custom Objects**

Custom objects are the native components that will model the data we need to store in our Recruiting app. Similar to a database table, a custom object is composed of several fields that store information such as a job
applicant's name, or the maximum salary for a particular position. However, unlike with traditional database
tables, we don't need to write any SQL in order to create custom objects. We can simply point and click in the
AppExchange platform to create as many objects as we need.

For our Recruiting app, we'll be creating four custom objects to track recruiting-related data:

- Position
- Candidate
- Job Application
- Review

Three of these objects, Candidate, Position, and Job Application, will be displayed as tabs in our application.
When a user clicks one of the tabs, he or she will have access to individual instances of that particular object, as
shown in the following screenshot.

One of the powerful features of a custom object is the fact that it can have relationships with other objects in the
system. For example, for every review written by an interviewer and entered into the system, we'll want to associate
it with the job application of the candidate who was being interviewed. Again, we won't need to write any SQL
to make this happen—thanks to the AppExchange platform, defining a relationship will be as simple as a few
clicks of the mouse.

Security and Sharing Rules

Another important function that we'll need to build into our app is the ability to restrict access to data that
particular users shouldn't see, without preventing other users from performing their jobs effectively. We're going
to implement this requirement with a group of AppExchange components that we've grouped under a single
term: security and sharing rules.

With security and sharing rules, we'll first specify which custom objects a particular user should be allowed to
create, view, or edit (for example, Candidate and Position), and then which instances of those objects should be
accessible (for example, the records for candidate John Smith or the Senior Sales Manager position). Controlling
our data either with the wide brush of object-level security or with the more detailed brush of record-level security
will give us a lot of power and flexibility in controlling what users can and can't see.
Workflow

Finally, two of our requirements involve automating business processes, such as triggering an alert email to a recruiter whenever a job application’s status has changed. Once again, the AppExchange platform makes this easy for us to implement natively with a built-in component called workflow.

Workflow allows us to create business logic based on rules. For our Recruiting app, we can create a workflow rule that triggers an event whenever the status of a job application has changed to Reject or Extend an Offer, as illustrated below.

![Workflow Diagram](image)

**Figure 5: Workflow When a Job Application's Status Has Changed**

When a hiring manager makes a decision to either extend an offer to or reject the candidate, changing the status of the application triggers the appropriate task to be assigned to the recruiter for that position. Based upon the hiring manager’s decision, the recruiter performs the appropriate follow-up task.

Composite Components

Although we’ll be able to use native AppExchange platform functionality to satisfy most of our Recruiting app use cases, there's still one use case that won’t be so easy to implement: the ability to map the locations of all candidates who are applying for a particular position.

At this point, we'll need to leave the relative comfort of the platform's native components and cover the gap by building a composite component, leveraging functionality from another website like Yahoo! maps. Although we’ll have to write a little code to make this work, the integrated component lives in a tab and looks just like any other part of our custom app.
Although we haven't yet gone into detail about how any of this stuff is going to work, you can probably see now just how flexible and powerful the AppExchange platform can be when you're creating a custom app.

In the next chapter, we'll start out by building our first custom object. We'll swiftly get a feel for how the platform interface works, and it won't be any time at all before you're building app components easily and quickly. When it's this easy, you can't help but become an AppExchange expert!
Chapter 3

Reviewing Database Concepts

In this chapter ...

• What's a Database?
• What's in a Database?
• What's a Relational Database?
• Summary of Database Concepts

Now that we've introduced the power of the AppExchange platform and learned about the requirements of the Recruiting app that we're going to be building, let's take a moment to talk about databases and why a simple understanding of database concepts can help you realize the full potential of the platform and make your app development a whole lot easier.

As you know, the underlying architecture of the AppExchange platform includes a database where your data is stored. This means that all of the information you enter is stored in that database and then retrieved from the database whenever you view it within your app.

Historically, companies were required to buy, build, and maintain their own databases and IT infrastructures in order to distribute and run their applications. On-demand computing on the AppExchange platform provides an alternative and makes it easy for you, as a company or as a sole developer, to build and deliver your app. Part of the simplicity of the on-demand model is that the technical responsibilities of maintaining and running all of the database hardware and software is handled by the hosting company (in this case, salesforce.com), so you can focus on developing your app.

It's worth pointing out that although your data is stored in a database and a simple understanding of database concepts is helpful, you don't need to be a database developer to build an app on the AppExchange platform. We won't be doing any traditional database programming in the course of developing our app.
What's a Database?

In simple terms, a database is an organized collection of information. Common examples include a phone book, a library catalog, an employee directory, a catalog of the MP3s you own, or in the case of our Recruiting app, information about the open positions at a company, the people who are applying for those positions, and the managers at our company who are in charge of hiring each position.

Typically, you use a database to collect information about people, things, or concepts that are important to you and whatever project you’re working on. In standard database language, the category of person, thing, or concept you want to store information about is referred to as an entity, although in standard AppExchange terminology, we refer to this as an object. In a database, each entity is represented by a table. A database table is simply a list of information, presented with rows and columns, about the category of person, thing, or concept you want to track. So in a phone book, you might have a table to store information about residences and another table to store information about businesses; or in a library catalog, you might have one table to store information about books and another to store information about authors.

In our Recruiting app, we'll have one table to store information about open positions, another table to store information about the candidates applying for the positions, and a table to store information about hiring managers. (Our Recruiting app will have more than just this, but we’ll get to that later.)

In very simplistic terms, an AppExchange object is similar to a database table in that you’ll have a separate object for each person, thing, or concept about which you want to collect information. In reality, an AppExchange object is much more than this because the full functionality of the platform is behind each object. Each object automatically has built-in features like a user interface, a security and sharing model, workflow processes, and much more that you’ll learn about in the rest of this book.

Note: As we introduce database concepts, "object" and "table" will be used interchangeably because they are similar, but just remember that an AppExchange object is much more than just a database table.

It’s important to understand that a single database table, or AppExchange object, should contain only one type of information. You don’t want to lump all of your information into one table, so you wouldn’t store positions, candidates, and hiring managers all in the same place. Not only is this not good database design, but it doesn’t allow you to relate objects to one another. For example, if all of our data were in one table, how would we ever know which candidates were applying for which positions, or which managers were in charge of hiring for which positions?

As we define our app, it's important for us to keep this in mind and ask ourselves questions like, "What kind of information do we want to store? Can we separate our information into distinct categories so that each object holds only one type of person, thing, or concept?" The answers to these questions will guide us as we design the structure of our application.

What's in a Database?

As we mentioned, a database table presents your information in rows and columns. Let’s take a look at how a table of positions might look:
Each row in the table represents the information about a specific instance of the object, for example, the Recruiter position or the SW Engineer position. In standard AppExchange terminology, we call this a record. For every object you want to track in your app, you’ll have multiple records to represent each individual item about which you’re storing information. It’s common for users who are new to the AppExchange platform to confuse the meanings of object and record. It’ll make your development a lot easier if you remember that an object is a category of information, such as a position or candidate, and the record is a single instance of an object, such as a SW Engineer.

**Note:** As a side note here, we’ll mention that the platform includes a set of built-in objects when you first start using it; we call these standard objects. One example of a standard object is the User object, which stores information about each person who is a user of the app, like our hiring managers. You can also build your own objects to store information that’s unique to your app; we call these custom objects. Both standard objects and custom objects are not really all that different—one kind is prebuilt for you, and the other you build yourself. We’ll talk more about these later as you start to build your app.

Now let’s look at the columns in the table. Each column lists a particular piece of information such as the Position Title or Max Pay. We refer to these as fields. Every object has a set of fields that you use to enter the information about a particular record. For each field in the table, a single item of data that you enter, such as the Functional Area of "Finance," is referred to as a data value.

Just like objects, fields come in two varieties: standard and custom. The standard fields are the ones that are built into the platform and automatically added for you. The custom fields are the ones you define to store specific pieces of information that are unique to your app. Fundamentally, there is no difference between standard and custom fields. Both are simply columns in the database table. We’ll talk more about standard and custom fields later when you begin building your app.

### What's a Relational Database?

Now you have some information stored in your database, but so what? You could easily make a list of positions using Microsoft Excel or some other spreadsheet software. For each position, you could even list the hiring manager in a field called Hiring Manager, like this:
But what if a hiring manager is responsible for hiring more than one position? You would need to have duplicate records for the same hiring manager so you could capture every position for which that hiring manager is responsible, like this:

This is not a good database design! Using this approach, data is repeated unnecessarily. In addition, there is really no way to capture additional information about our hiring managers, like their email addresses or phone numbers. And if we try to add information about which candidates are applying for each position, you can imagine that our simple table will quickly become extremely complex and unmanageable.

As we mentioned before, you want to create separate database tables, or AppExchange objects, for each person, thing, or concept you want to track. A better way to model our scenario here would be to create one object for positions, one object for candidates, and one object for hiring managers. (Luckily, the AppExchange platform has a standard object that we'll be able to use to represent our hiring managers—the User object.)

Once we have our data separated into discrete objects, we can easily relate objects to each other. This is what a relational database is all about! A relationship is an association between two or more tables. For example, we can relate positions to hiring managers so we know which positions each hiring manager is responsible for:
From a technical standpoint, each table in a relational database has a field in which the data value uniquely identifies the record. This field is called the primary key. The primary key is one part of what defines the relationship; the other part is the foreign key. A foreign key is a field whose value is the same as the primary key of another table. You can think of a foreign key as a copy of a primary key from another table. The relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another.

Primary and foreign keys are fundamental to the concept of relationships because they enable tables to be related to each other. As you begin building your app, you won't really need to think too much about primary keys and foreign keys. The important concept to understand here is that in a relational database, objects are related to each other through the use of common fields that define those relationships.

Summary of Database Concepts

At this point, we're ready to dive into the building of our Recruiting app. But first let's recap what we've learned about databases. Whether this was your first introduction to databases or whether you're already an experienced database developer who's new to the AppExchange platform, the important things to remember are:

- A database is an organized collection of information.
- A database table stores information about a single type of person, thing, or concept—such as a job position. In the AppExchange platform, we use the term object here (even though an AppExchange object is much more than this, as you'll see).
- A database row, or record in AppExchange terms, represents a single instance of an object—such as the SW Engineer position.
- A field stores a particular piece of information on a record.
- Relationships define the connection between two objects, and objects are related to each other through the use of common fields.

Now that we've got that all covered, let's get started building our first object!
Chapter 4

Building a Simple App

In this chapter ...

- Becoming Familiar with the Setup Area
- Introducing Apps
- Introducing Objects
- Introducing Tabs
- Becoming Familiar with Setup Detail Pages and Related Lists
- Introducing Fields
- Look at What We've Done

Just as traditional programming books first teach you how to write a simple "Hello World" program before getting into more complicated things, in this chapter, we're going to create a very simple Recruiting app to show you just how easy it is to get started with the AppExchange platform. Along the way we'll orient ourselves to the platform's user interface (where we'll be doing most of our work), and we'll learn how to create and configure our first custom object. Although easy and straightforward, the tasks we complete here will be the first step in developing a full-featured Recruiting app. So let's dive right in!
Becoming Familiar with the Setup Area

Since we're going to spend most of our time working in the Setup area of the AppExchange platform, let's first become familiar with what it is and how to navigate to it.

The Setup area is an administrative tool and user preferences area, all in one. We perform almost every task we need to create our app in the Setup area, so most of the "Try It Out" sections of the book are going to start with an instruction like, "Click Setup ➤ Build ➤ Custom Apps." This is a short way of saying:

1. Click the Setup link in the top-right corner of the page (shown in the figure below).
2. Go to the App Setup area on the left side of the page.
3. Click the + icon to expand the Build menu, or just click the Build link.
4. Click the Custom Apps link.

![Figure 11: The AppExchange Platform Setup Area](image)

The final link that you click (in this example, Custom Apps) will change depending on the task you're trying to perform, but you get the general idea.

Similar to the other parts of the application, the Setup area consists of a tab bar, a navigational sidebar, and a main window:

- The tab bar is made up of the same tabs that appear in the regular application. Just click on any one of the tabs to exit the Setup area and go to that tab in the main application.
- The navigational sidebar includes expandable lists of all the tools that are available in the Setup area:

  **Personal Setup**
  These tools control individual user preferences and are available to all users.

  **App Setup**
  These tools configure the standard objects, custom objects, and custom apps that are deployed and are typically used only by administrators.

  **Administration Setup**
  These tools configure the AppExchange platform as a whole and are typically used only by administrators.
• The main window is where the navigational links or a selected setup tool are actually displayed.

Now that we know what we're looking at, let's start creating our simple app.

Introducing Apps

What should we do first? If we were writing a computer program, the first thing we'd need to do is build a project where we could store all the code that we were going to write. With the AppExchange platform, the first thing we need to do is create a new app.

Like a programming project, an *app* is little more than a container for all of the objects, tabs, and other functionality that we're going to build as part of our Recruiting application. It consists simply of a name, a logo, and an ordered set of tabs. The simplest app contains only one tab—the Home tab—and a default logo. As we define more tabs in the remainder of this book, we can add them to the app later.

Let's start clicking through the process of actually creating a simple app now. Log in to the Salesforce Developer Edition account that you signed up for earlier so you can follow along!

Note: You might find that the screenshots you see in this book vary slightly from what you see on your screen. These changes should be minor and shouldn't affect your understanding of the app building process.

Try It Out: Defining an App

To create an app:

1. Log in to your Salesforce account using your username and password. Typically, we refer to your Salesforce account as an "organization" so be aware of that as we proceed through the rest of the book.
2. Click Setup ➤ Build ➤ Custom Apps.
3. If you see an introductory splash page, simply click Continue.

Welcome to the Custom Apps list page! Like many of the setup tools, the starting page for the Custom Apps tool consists of a list of all the apps that are currently enabled for your organization. Depending on what you've already purchased from salesforce.com or downloaded from the AppExchange website, you'll probably already have some standard apps listed here.

4. Click New.

The New Custom App wizard appears.

5. In the **Label** field, enter Recruiting.

The label is the name that will represent our new app in the AppExchange app menu that appears at the top right of all pages. Users can use this menu to switch back and forth between apps.

Notice that a vertical red bar appears just to the left of this **Label** field. This red bar indicates that you must provide a value for this field in order to save your work. If you don't enter a value here and try to proceed, an error message is displayed, as shown in the following screenshot.
6. In the Description field, enter "Manage positions, candidates, and job applications."
7. Click Next.

The next screen in the New Custom App wizard allows you to specify the image file that should be used for this app's logo. Whenever the app is selected in the AppExchange app menu, this is the logo that appears in the upper-left corner of all pages. Since we're just creating a simple app, let's accept the default Salesforce logo that's already provided. We can always change it later.

8. Click Next.

As we said before, an app is a container for an ordered collection of tabs, and this step of the New Custom App wizard allows us to specify which tabs we want to include in our new app. The Available Tabs list shows us the standard and custom tabs that are available for us to choose, and the Selected Tabs list shows us which tabs are already included, listed in the order that they should be displayed. You'll notice that one tab, the Home tab, is already included in our app by default. This is because the Home tab is required in every app, and must always be in the first position.

Again, since we're just creating a simple app, let's accept the default and move on. We'll add more tabs later.

9. Click Next.

Now that we've defined the label, logo, and tabs that should make up our app, you might be wondering what remains to be done in the New Custom App wizard—shouldn't we already be done? It turns out that one crucial step remains: we need to define the users who will be allowed to access our app.

In this step of the New Custom App wizard, we can choose which user profiles should have access to the app. We'll learn more about profiles in Chapter 7: Securing and Sharing Data on page 81. For now, just understand that every user is assigned to a profile, and profiles control which apps the users assigned to that profile can view.

10. Select the Visible checkbox next to the Standard User and System Administrator profiles.
11. Click Save.

That's it!

Look at What We've Done

Now that we've made it back to the Custom Apps list page, let's see what we've just done. First of all, we've got a new entry in the Custom Apps list—our Recruiting app! It shows up at the bottom of the list, signifying that
it's going to show up at the bottom of our AppExchange app menu. In fact, let's go look at the AppExchange app menu now.

Figure 13: AppExchange App Menu

Tip: If you want to change the position of our app in this menu, do so from the Custom Apps list page by clicking Reorder and arranging the available apps as you see fit.

Now select the Recruiting app from the menu and see what happens—our app is launched with a single Home tab! We've created the Recruiting app's Home tab, and we've added it to the AppExchange app menu. That's how easy it is to get started.

You'll notice that the approach we're taking here is iterative: we'll build part of the app, look at what we've accomplished, and then add to it. This sequence not only reflects the fact that we're leading you through the steps of building an app in this book, but you'll also find that in building AppExchange apps in general, this iterative process is common.

During the course of this book, you'll also notice that unlike with traditional coding projects, your app is always functional. There's no build or compile phase, and as a result, you'll almost never be chasing down syntax bugs or other typos. In fact, with this simple one-tab app, you can already utilize all of the built-in functionality that comes with the AppExchange platform, including search, calendar events and tasks, user preferences, and a familiar user interface.

Introducing Objects

Now that our app is functional (but rather boring), let's make it a little more interesting by introducing our first object.

As you might remember from the last chapter, an object is very similar to a database table in the AppExchange platform. While the platform already comes with a number of standard objects that support default apps like Sales and Service & Support (for example, contacts, accounts, and cases), we can also define custom objects that allow us to store information specific to our Recruiting app.

Whether they're standard or custom, AppExchange objects not only provide a structure for storing data but they also power the interface elements that allow users to interact with the data, such as tabs, the layout of fields on a page, and lists of related records. Because any object can correspond to a tab, and an ordered collection of tabs makes up an app, objects make up the heart of any app that we create with the AppExchange platform.

With custom objects being so important—they have lots to do with how our app will look, behave, and feel—what we do with custom objects and how we use them quickly becomes essential to creating a successful app. The design of the data model behind an app is typically the biggest factor in its success or failure.

That's enough talk about objects for now. Let's go define one!
The Position Custom Object

The first custom object that we'll create for our Recruiting app reflects a typical recruiting task: describing a position. Recruiters at Universal Containers need to keep track of all the positions they're hiring for, such as a Senior Developer, Sales Engineer, or Benefits Specialist. They'll need easy access to all positions in the system through a tab, and they'll need to include certain information for each position, such as its minimum and maximum salary range, the position's location, and its hiring manager. InAppExchange terms, we'll create a custom object, create a custom tab for that object, and then define some custom fields.

Try It Out: Defining the Position Custom Object

To create the Position object, we're going to stay in the Setup area.

1. Click **Setup ➤ Build ➤ Custom Objects**.
2. On the All Custom Objects page, click **New Custom Object**.

Unlike defining a custom app, which we did through the New Custom App wizard, defining a custom object is confined to just one page. You'll find that the AppExchange platform uses wizards or single pages depending on the amount of information that needs to be specified to fully define the object.

3. In the **Label** field, enter Position.
4. In the **Plural Label** field, enter Positions.
5. The **Object Name** field is defaulted to Position. Let's leave it as is.

The **Label** and **Plural Label** of a custom object are what users see in all of the object's related user interface elements, such as the object's tab or in search results headings. Object labels work best as nouns, and the plural label is always used to label a custom object's tab (if you create a tab for your object).

The value of a custom object's **Object Name** represents the unique name for the object when it's referenced in the AppExchange API. This value is helpfully autogenerated based on the value that you enter for the **Label**,
except with all spaces and punctuation replaced with underscore characters. We'll talk more about the API in Chapter 9: Moving Beyond Native Apps on page 131. For now, just keep in mind that the Object Name value must be unique across all objects defined in your organization.

6. In the Description field, enter "This object stores information about the open job positions at our company."

7. In the Record Name field, enter Position Title.

The Record Name is the label for the field that identifies individual Position records in the system. A custom object cannot be saved without this identifying field.

8. In the Data Type drop-down list, select Text.

The Data Type drop-down list allows you to select the type of value that should be used for this identifying field: either Text or Auto-Number. Some objects, like Positions or Accounts, can be identified with a text field, because there will always be a name for a position or account available. Other objects, like a Case (used in the standard Service & Support app) are harder to identify with a single text field, so we would assign them auto-numbers instead.

💡 Tip: Whenever possible, it's best to use text as the data type for an identifying field so that users can more easily identify a particular record when several of them appear together in a single list.

To illustrate how custom object and record name labels work together in the app, let's fast forward a bit to see where each label will appear once we've defined our Position custom object, its tab, and a single Sr. Developer position record.

Let's move on.

9. In the Optional Features area, select the Allow Reports, Allow Activities, and Track Field History checkboxes.

These three checkboxes actually enable some really robust functionality:

**Allow Reports**

Selecting this option makes the data in the Position records available for reporting purposes. The AppExchange platform comes with a large number of standard reports, and users can also create custom reports by using a simple yet powerful reporting wizard. (To find out more about reports, check out the online help by clicking Help & Training on any page.)
Allow Activities

Selecting this option allows users to associate tasks and scheduled calendar events with a particular position. For example, a user could create a task, such as "Update salary range for Sr. Developer position," and specify attributes such as priority, due date, and status. The user could then handle the task herself, or assign it to someone else. (Again, this is all covered in the online help.)

Track Field History

Selecting this option causes the platform to automatically track edits to position records, such as who changed the value of a field, when it was changed, and what the value of the field was before and after the edit. History data is available for reporting, so users can easily create audit trail reports when this feature is enabled.

In general, you should select these options if there's any chance that they might be useful for whatever custom object you're defining.

10. In the Deployment Status area, select Deployed.

Note: This step assumes that you're working in a development environment. If you're not, and if you don't want users to see the Position object after you click Save, select In Development. Setting the status to In Development will hide Position records from all users except those with the "Customize Application" user permission (that is, just about anyone who isn't a System Administrator).

11. In the Object Creation Options area, select the Add Notes & Attachments related list to default page layout and Launch New Custom Tab Wizard after saving this custom object checkboxes.

These two options are available only when you're creating a new custom object. If you later decide to go back and edit some of the details about your custom object, you won't see them. But what do they do?

• Enabling notes and attachments for an object means that you can attach external documents to any Position record, in much the same way that you can add a PDF or photo as an attachment to an email. It's handy functionality, so you should generally select it.

• Launching the New Custom Tab wizard does exactly what it says—it's a short cut to launching the tab wizard after we've saved our Position object and will save us a few clicks if we know that we need a tab.

All set? Let's go ahead and save our Position custom object now.

12. Click Save.

That's all there is to it! As promised, the New Position Tab wizard displays instead of the list of custom objects that we'd normally see. Let's take a moment to talk about why we should even be defining a tab for our Position object in the first place. What's so great about tabs, anyway?

Introducing Tabs

If you're familiar with the AppExchange platform, you know that clicking tabs is how you navigate around an app. Every tab serves as the starting point for viewing, editing, and entering information for a particular object. When you click a tab at the top of the page, the corresponding home page for that object appears. For example, if you click the Accounts tab, the Accounts tab home page appears, giving you access to all of the account records...
that are defined in your organization. Click the name of a particular account record, and you'll view all of the record's information in its associated detail page.

What's really powerful about building an app with the AppExchange platform is that you can create custom tabs that look and behave just like the tabs for standard objects that are already provided. From the perspective of your end users, any customizations that you make appear perfectly seamless, and as a developer, you don't have to do anything special to make it work that way! If only other platforms and programming languages made it so easy! Let's see how quickly we can create a tab for our Position object.

**Try It Out: Defining the Positions Tab**

To create a custom tab for our Position object, we're going to use the New Custom Object Tab wizard that was so helpfully launched for us when we clicked **Save** after defining the object. However, in case you forgot to select the **Launch New Custom Tab Wizard after saving this custom object** option or are coming back to work that you previously saved, have no fear! There's another way to launch the wizard.

1. Click **Setup ➤ Build ➤ Custom Tabs**.
2. In the Custom Object tabs area, click **New**.

Easy. Now that we're all on the same page, let's get started working through the wizard.

3. In the **Object** drop-down list, select Position.

If you launched the wizard directly after defining the custom object, the Position object is automatically selected for you.

4. Click the **Tab Style** lookup icon to launch the Tab Style Selector as shown in the following screenshot.

Every object that appears as a tab must have a unique color scheme and icon. This color scheme is what identifies the object, not only on its tab but also in different places in the user interface, such as in related lists and search results.
In the Tab Style Selector, you can choose a predefined color and icon, or you can create your own. To keep things simple, we're going to select an existing style.

5. Click the **Hide values which are used on other tabs** link to make sure you choose a unique style.

6. Click any colored box to choose a color scheme and icon.

Leave the **Splash Page Custom Link** drop-down list set to --None--. We'll learn more about custom links in Chapter 9: Moving Beyond Native Apps on page 131.

7. In the **Description** field, enter "A tab and color scheme for the Position custom object."

8. Click **Next**.

9. Click **Next** again to accept the default user profile visibility.

Just as we controlled access to our Recruiting app by selecting user profiles in the New Custom App wizard, we can also control access to our Positions tab by selecting user profiles here. We'll learn more about user profiles and what they do in Chapter 7: Securing and Sharing Data on page 81. For now, just know that accepting the defaults will make the tab visible to all users.

10. Deselect all of the **Include Tab** checkboxes except the one for our Recruiting app.

In performing this step, we're providing access to the Positions tab only when someone has access to our Recruiting app. Unless an employee is interested in recruiting, he or she probably doesn't need to see this tab.

11. Select the **Append tab to users' existing personal customizations** checkbox.

If you don't select this option, any users who have personalized their tab display will not immediately see the Positions tab. Also, if you've already created a new tab and didn't turn this option on, you have to first delete the existing tab and then re-create it with this option turned on to automatically push the tab to existing users. What a pain! Do yourself a favor and just always keep this option selected.
12. Click **Save**.

You'll notice when the page refreshes that the Positions tab has automatically been added next to the Home tab at the top of the page.

**Look at What We've Done**

To truly appreciate what we've just built with a few clicks, let's take a look at what we've done.

1. First, click the Positions tab to display the Positions tab home page, as shown below. Although the list is empty because we haven't yet created any records, you can see how this page will become the gateway to viewing, creating, editing, and deleting all of the positions that we create in our Recruiting app. It looks just like the tab home page of any other standard object.

![Figure 17: The Positions Tab Home Page](image)

2. Now, check out the contents of the **Create New...** drop-down list in the left sidebar. As promised, our custom object has been seamlessly incorporated into the platform with the other standard objects like Event and Task. An end user need never know that the Positions tab was created with a custom object, because it shows up alongside the standard objects as well.

3. Select Position from the **Create New...** drop-down list, or click **New** in the Positions tab home page. Voilà! It's the Position Edit page! Sadly, though, our position still doesn't have much room for data. At this point, all we have is a field for **Position Title** (the record identifier) and **Owner**, a default field that appears on every object to identify the user who created the object.

4. Click **Cancel**. It doesn't do to create a Position record with hardly any interesting data. We need more fields! And sure enough, that's what we'll get to next. First, though, let's revisit our Position custom object and orient ourselves to what else is available through a custom object detail page in the Setup area.

**Becoming Familiar with Setup Detail Pages and Related Lists**

You may recall when we first introduced the concept of objects that we learned: "Whether they're standard or custom, AppExchange objects not only provide a structure for storing data but they also power the interface elements that allow users to interact with the data, such as tabs, the layout of fields on a page, and lists of related records." If you've been following along closely, you might have been wondering why we didn't get to define any fields (other than the identifier field of Position Title) or user interface elements (other than the Positions tab) when we created our Position object. If fields and user interface elements are a part of the definition of what a custom object is all about, where do we get to define them?
It turns out that the AppExchange platform differentiates the initial creation of certain components from details related to those components. In other words, the information that we see when we define or edit a custom object is different from the information that we see when we view a custom object that's already defined. Let's go back to our custom object list page to see how this difference is reflected in the platform interface:

1. **Click Setup ➤ Build ➤ Custom Objects.**

Here we are back in the custom object list page. You'll notice in the row for Position there are three links that we can click:

- **Edit**
  This link takes us back to the custom object edit page where we originally defined our Position object.

- **Del**
  This link deletes the custom object, including any records, tabs, reports, or other components associated with that object.

- **Position**
  This link takes us to the custom object detail page for our Position object.

![Figure 18: Custom Object List Page: Edit, Delete, and Detail Links](image)

We're already familiar with the edit page from when we defined our Position object, and we certainly don't want to delete our object. Let's go ahead and open up the detail page to see what we can do there.

2. **Click Position.**

As you can see, the Custom Object Edit page that we filled out when we defined our Position object was just the tip of the iceberg. The top two areas of the Position detail page (see Figure 19: Position Custom Object Detail Page on page 43) include all of the information that we originally specified, plus a few standard fields that the platform includes with every object. Below those areas are several additional groupings of data that allow us to do more with our Position object.

In AppExchange terms, those groupings of data are called **related lists**, and they're a big part of what makes the AppExchange platform so powerful. A related list is a list of records or other components that are associated with whatever we're viewing. Related lists appear in both the main application and in the Setup areas and represent a relationship between the items that appear in the related list and the object or record that we're viewing in the detail area. We'll learn a lot more about relationships in **Chapter 6: Expanding the Simple App Using Relationships**.
on page 65, but for now, just understand that anything that appears in an object's related list is directly related to that object.

Now that we've found out where we can continue customizing our Position custom object, let's use the Custom Fields & Relationships related lists to create some more fields in our Position object.

---

**Introducing Fields**

We're ready to add more fields to our Position custom object, but first, let's talk briefly about what a field is and how it fits in to the world of the AppExchange platform.

As you might remember from the last chapter, a *field* is like a database column. The primary characteristic of a field is its data type—some fields hold text values, while others hold currency values, percentages, phone numbers, email addresses, or dates. Some fields look like checkboxes, while still others are drop-down lists or record lookups from which a user makes a selection.
The data type of a field controls the way the field is ultimately displayed in the user interface and how data entered into the field is stored in the platform. To get a better feel for how the fields will look, let’s take a sneak peak at what the Position object is ultimately going to look like, and the types of custom fields we’re going to create for it:

![Figure 20: Position Custom Object Fields](image)

There are a lot of fields here that we need to define, some more complicated than others. To keep things simple, let’s go through and create the simple text, currency, and checkbox fields. We can tackle the more complicated picklists and custom formula fields in Chapter 5: Enhancing the Simple App with Advanced Fields and Page Layouts on page 49.

**Try It Out: Adding Text Fields**

First let's define a few text fields. We already created a basic text field for **Position Title** when we defined our Position custom object. Looking at our screenshot, the only text fields that remain are the text fields under the **Description** heading. We’ll start by defining the **Job Description** field.

1. Click **Setup ➤ Build ➤ Custom Objects**.
2. Click **Position**.
3. In the Custom Fields & Relationships related list, click **New**.

Every time you create a custom field, you’ll first choose a data type from the field type selection page.

The AppExchange platform allows us to choose from three different types of text fields:

- Basic text fields allow users to enter any combination of letters and numbers on a single line, as many as 255 characters.
- Text area fields also have a 255 character limit but also allow carriage returns so the text can be formatted on separate lines.
- Long text fields allow as many as 32,000 characters, on separate lines.

Since job descriptions can be lengthy, let’s choose a long text area.

4. Choose the **Text Area (Long)** data type, and click **Next**.
Tip: Carefully consider the data type you choose for your custom fields, because once you set them, it isn’t always the best idea to change them later. Review the online help for details.

The second page of the custom field wizard allows us to enter details about our long text area field. The fields that appear in this step change depending on the data type that we selected in the previous page.

5. In the Field Label field, enter Job Description.

Like the other labels that we’ve seen in the platform so far, the Field Label specifies the text that should be displayed to users when the field is rendered in the user interface. Notice that when we enter a value for Field Label, Field Name is automatically populated with the same text but with all spaces and punctuation replaced by underscores. The value for Field Name is a unique name that is used to refer to the object when using the AppExchange API.

6. In the Length field, enter 32,000.

The Length field allows us to restrict the maximum number of characters that are allowed. Since we don’t get any benefit from this kind of restriction, leave this value set to the maximum allowed.

7. In the # Visible Lines field, enter 3.

This field allows us to specify how large our text box will appear on the page.

The remaining fields on this page have to do with formula fields. We’ll learn more about those in the next chapter. For now, just leave them blank.

8. Click Next.

The third page of the custom field wizard allows us to restrict access to this field from certain user profiles. We’ll learn more about profiles and field-level security in Chapter 7: Securing and Sharing Data on page 81, so for now, just accept the defaults.

9. Click Next.

The last page of the wizard allows us to automatically place our field on the Position page layout. Again, we’ll learn about page layouts in the next chapter, so for now, just accept the defaults.

10. Click Save & New.

Instead of clicking Save and returning to the Position object detail page, clicking Save & New saves ourselves a few clicks and allows us to finish up the other text area fields that we need. Here’s what you need to know to define them:

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Label</th>
<th>Length</th>
<th># Visible Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Area (Long)</td>
<td>Responsibilities</td>
<td>32,000</td>
<td>3</td>
</tr>
<tr>
<td>Text Area (Long)</td>
<td>Skills Required</td>
<td>32,000</td>
<td>3</td>
</tr>
<tr>
<td>Text Area (Long)</td>
<td>Educational Requirements</td>
<td>32,000</td>
<td>3</td>
</tr>
</tbody>
</table>

Now that we’ve wet our feet with text fields, let’s quickly create a few more fields of other types. You’ll find that with few exceptions, they’re all very similar to one another.
Try It Out: Adding Currency Fields

To keep track of a position's salary range, we need to add two currency fields: Min Pay and Max Pay. Note that unlike other fields, once we define these as currency fields, we won't be able to change them to any other type.

Defining a currency field is almost identical to defining a text field, with a few slight differences:

- The Length of a currency field actually corresponds to the number of digits to the left of the decimal point. An additional Decimal Places field handles the number of digits that should be displayed to the right.
- In the Details page of the wizard, a new checkbox called Required is displayed. We can select this option if we want to force our users to enter a value for this field when creating a new position.

Everything else should be familiar to you, so go ahead and use the custom field wizard to define the following fields:

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Label</th>
<th>Length</th>
<th>Decimal Places</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currency</td>
<td>Min Pay</td>
<td>7</td>
<td>2</td>
<td>Leave unchecked</td>
</tr>
<tr>
<td>Currency</td>
<td>Max Pay</td>
<td>7</td>
<td>2</td>
<td>Leave unchecked</td>
</tr>
</tbody>
</table>

Try It Out: Adding a Checkbox Field

Here's an easy one. The Position object requires a checkbox field to indicate if travel is required for the position. By default, this value should be unchecked. (Note that similar to currency fields, once you define a field as a checkbox, you can't change it to any other type.)

Use the custom field wizard one more time to define this field:

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Label</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkbox</td>
<td>Travel Required</td>
<td>Unchecked</td>
</tr>
</tbody>
</table>

Look at What We've Done

We've defined text, currency, and checkbox fields for our Position object. Let's take a look by going to the Positions tab and clicking New.
Check out all the fields we've just made! It's not the most elegant layout for all of our fields (each field got added to the page in the order that we created it), but it's definitely functional, and it looks just like any other AppExchange platform page. Wasn't that easy?

Once again, welcome to the power of the AppExchange platform. First we created a new recruiting app with a single Home tab, then we created a Position object and tab, and now we've just added a few fields, all in less than 15 minutes of clicking around. From start to finish we always had a fully functional app, and we never had to spend any time compiling or debugging our "code"!

In the next chapter, we'll enhance our simple Recruiting app even further by adding some more-complex fields and then moving the fields around in a page layout so that users can more easily find and enter the information they need. Let's keep going!
In the last chapter, we got our Recruiting app off to a quick start by defining the Position custom object, tab, and several simple fields. This simple version of our app had the same look and feel as any other page in the AppExchange platform, and we were able to whip it together in a matter of minutes.

In this chapter, we're going to enhance the Positions tab: first by defining a few more advanced fields, and then by moving these fields around within a page layout. These additions will help change the detail page of our Positions tab from a somewhat flat and inelegant user interface to something that users find powerful and intuitive to use. Let's get started!
Adding Advanced Fields

In this section, let’s revisit the custom field wizard to help us create fields with more-sophisticated data types: picklists, dependent picklists, and custom formula fields. We’ll see how the platform’s user interface helps guide us through the setup of these more complicated fields.

Introducing Picklists

When viewing the preview of what we wanted our Positions page to ultimately look like, there were several fields that were specified with drop-down lists. In AppExchange terms, these fields are called picklists, and they consist of several predefined options from which a user can select.

Picklists come in two flavors: a standard picklist, in which a user can select only one option, and a multi-select picklist, in which a user can select multiple options at a time. For the purposes of our Position object, we need to define standard picklists for a position’s location, status, type of job, functional area, and job level.

Try It Out: Adding Picklists

Let’s walk through the creation of the Location picklist field. Then, as in the previous chapter, we’ll give you the information that you need to create the others on your own.

1. Click Setup ➤ Build ➤ Custom Objects.
2. Click Position.
3. In the Custom Fields & Relationships related list, click New.
4. Select the Picklist data type and click Next.
5. In the Field Label text box, enter Location.
6. In the large text area box just below, enter the following picklist values, each on its own line:

   - San Francisco, CA
   - London, England
   - Austin, TX
   - Berkeley, CA
   - Oakland, CA
   - San Diego, CA
   - Boulder, CO
   - Dublin, Ireland
   - New York, NY
   - Sydney, Australia
7. Select the Sort values alphabetically checkbox.

Selecting this checkbox means that all of our locations will be sorted alphabetically, regardless of the order in which we entered them.

The Use first value as default value option allows us to populate the field with a default value. If you leave this deselected, the field defaults to None on all new Position records. Otherwise the field defaults to the first value that you specify in the list of possible picklist values. Since positions are randomly distributed among all of the office locations, there shouldn’t be a default value for this field.

8. Accept all other default settings for field-level security and page layouts.

9. Click Save & New.

Easy! Now specify the remaining picklists according to the table below:

**Table 5: Status, Type, Functional Area, and Job Level Picklist Values**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Label</th>
<th>Picklist Values</th>
<th>Sort Alphabetically?</th>
<th>Use First Value as Default?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picklist</td>
<td>Status</td>
<td>• Open • Approved • Closed - Filled • Closed - Not Approved • Closed - Canceled</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Picklist</td>
<td>Type</td>
<td>• Full Time • Part Time</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Picklist</td>
<td>Functional Area</td>
<td>• Finance • Human Resources • Information Technology • Retail Operations • Warehousing • Miscellaneous</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Picklist</td>
<td>Job Level</td>
<td>• FN-100 • FN-200 • FN-300 • FN-400 • HR-100 • HR-200 • HR-300 • HR-400 • IT-100 • IT-200</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Introducing Field Dependencies

Now that we’ve made all those picklists, answer this question: How many times have you clicked on a drop-down list and found far too many values to choose from? For example, maybe you were selecting Uruguay from a list of countries, and every country in the world was on the list. That meant that you had to scroll all the way down to the countries that started with the letter U. What a pain!

Fortunately, the folks who built the AppExchange platform have encountered that situation a few times themselves, and as a result, they’ve given us a tool to help us avoid this problem with our own picklist fields: field dependencies.

Field dependencies are filters that allow us to change the contents of a picklist based on the value of another field. For example, rather than displaying every value for Country in a single picklist, we can limit the values that are displayed based on a value for another field, like Continent. That way our users can find the appropriate country more quickly and easily.

Picklist fields can be either controlling or dependent fields. A controlling field controls the available values in one or more corresponding dependent fields. A dependent field displays values based on the value selected in its corresponding controlling field. In the example above, the Continent picklist is the controlling field, while the Country picklist is the dependent field.

Try It Out: Creating a Dependent Picklist

Looking at the picklists that we’ve created, it’s quickly obvious that our users might get frustrated with the length of our Job Level picklist. Let’s make our users happy by turning Job Level into a dependent field of the Functional Area picklist. Doing this will allow users to see only the three or four relevant job level values when a department is selected in the Functional Area picklist:

1. Click Setup ➤ Build ➤ Custom Objects.
2. Click Position.
3. In the Custom Fields & Relationships related list, click Field Dependencies.
4. Click New.
5. For the **Controlling Field** drop-down list, choose Functional Area.
6. For the **Dependent Field** drop-down list, choose Job Level.
7. Click **Continue**.

A field dependency matrix displays with all the values in the controlling field across the top header row and the dependent field values listed in the columns below. For each possible value of the controlling field, we need to include the values that should be displayed in the dependent picklist when that controlling value is selected. In the field dependency matrix, yellow highlighting shows which dependent field values are included in the picklist for a particular controlling field value.

![Field Dependency Matrix](image)

**Figure 23: Field Dependency Matrix**

To include a dependent field value, you simply double-click it. To exclude a dependent value from the list, double-click it again.

For example, let's try it out by including the values that should be displayed in the **Job Level** picklist whenever Finance is selected in the **Functional Area** picklist:

8. In the column labeled Finance, double-click FN-100, FN-200, FN-300, and FN-400.

Those four fields should now be shaded yellow in the Finance column.

Instead of double-clicking every **Job Level** value, we can also use Shift+click to select a range of values or Ctrl+click to select multiple values at once. Once those values are highlighted in blue, we can click **Include Values** to include them, or **Exclude Values** to remove them. Let's try it out.

9. In the column labeled Human Resources, single-click HR-100 and then press and hold the Shift key while clicking HR-400.
10. Click **Include Values**.

Now we have values selected for both the Finance and Human Resources columns!

11. Continue highlighting the appropriate values for all of the remaining columns, as described in the following table.
Tip: To get to all of the values that you need to modify for this step, you'll need to click **Previous** or **Next** to see additional columns.

### Table 6: Functional Area and Job Level Field Dependency Matrix

<table>
<thead>
<tr>
<th>Functional Area (Controlling picklist field)</th>
<th>Job Level (Dependent picklist field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finance</td>
<td>• FN-100</td>
</tr>
<tr>
<td></td>
<td>• FN-200</td>
</tr>
<tr>
<td></td>
<td>• FN-300</td>
</tr>
<tr>
<td></td>
<td>• FN-400</td>
</tr>
<tr>
<td>Human Resources</td>
<td>• HR-100</td>
</tr>
<tr>
<td></td>
<td>• HR-200</td>
</tr>
<tr>
<td></td>
<td>• HR-300</td>
</tr>
<tr>
<td></td>
<td>• HR-400</td>
</tr>
<tr>
<td>Information Technology</td>
<td>• IT-100</td>
</tr>
<tr>
<td></td>
<td>• IT-200</td>
</tr>
<tr>
<td></td>
<td>• IT-300</td>
</tr>
<tr>
<td></td>
<td>• IT-400</td>
</tr>
<tr>
<td>Retail Operations</td>
<td>• RO-100</td>
</tr>
<tr>
<td></td>
<td>• RO-200</td>
</tr>
<tr>
<td></td>
<td>• RO-300</td>
</tr>
<tr>
<td></td>
<td>• RO-400</td>
</tr>
<tr>
<td>Warehousing</td>
<td>• WH-100</td>
</tr>
<tr>
<td></td>
<td>• WH-200</td>
</tr>
<tr>
<td></td>
<td>• WH-300</td>
</tr>
<tr>
<td></td>
<td>• WH-400</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>• MC-100</td>
</tr>
<tr>
<td></td>
<td>• MC-200</td>
</tr>
<tr>
<td></td>
<td>• MC-300</td>
</tr>
</tbody>
</table>

12. Click **Preview** to test the results in a small popup window.
13. Click **Save**.
Look at What We've Done

Now that we've created all those picklists, let's revisit the Positions tab to see what we have so far.

1. Go to the Positions tab.
2. Click New.
3. In the Functional Area picklist, select Finance.
4. Open the Job Level picklist.

![Figure 24: Dependent Picklist Fields]

Our Recruiting app users are going to be very happy that they no longer have to deal with long, onerous picklists. Now let's go add a field that's even more powerful and complex than a dependent picklist: a custom formula field.

Introducing Custom Formula Fields

Up to this point, the fields that we've defined have all had one thing in common—they each require a user to give them a value. Fields like that are very helpful for storing and retrieving data, but wouldn't it be great if we could somehow define a "smart" field? That is, what if we could define a field that looked at information that was already entered into the system and then told us something new about it?

Fortunately, custom formula fields give us the ability to do just that. Just as you can use a spreadsheet program like Microsoft Excel to define calculations and metrics specific to your business, we can use custom formula fields to define calculations and metrics that are specific to our Recruiting app.

For example, on our Position object, we've already created fields for minimum pay and maximum pay. If Universal Containers gives out yearly bonuses based on salary, we could create a custom formula field that automatically calculated the average bonus that someone hired to that position might receive.

How would we perform this calculation if we were using a spreadsheet? The columns in our spreadsheet would represent the fields that we defined on our Position object, and each row of the spreadsheet would represent a different position record. To create a calculation, we'd enter a formula in a new column that averages the values of Min Pay and Max Pay in a single row and then multiplies it by a standard bonus percentage. We could then determine the average bonus for every position record row in our spreadsheet.

Custom formulas work in a very similar way. Think of a custom formula like a spreadsheet formula that can reference other values in the same data record, perform calculations on them, and return a result. However, instead of using cell references, you use *merge field* references. And, instead of typing characters in a box, you have a wizard to help you select fields, operators, and functions.

The net result is that anyone can quickly and easily learn to create formula fields. And, as with all AppExchange platform tools, the on-demand delivery model makes it easy to experiment. You can create a formula, view the
results, and change the formula again and again, as many times as you want! Your underlying data is never affected.

**Tip:** When defining your own custom formula fields, leverage the work of others. You can find more than a hundred sample formulas on the Successforce website at success.salesforce.com.

### Calculating How Long a Position Has Been Open

Let's now think about another custom formula field that we could create for our Position object—a basic custom formula field that calculates how many days a position has been open. To do this, let's first think about the logic that we should use to define the field, and then we can go through the process of creating it in our Recruiting app.

Let's think about the data that we need to make this calculation: we need to know the current date and the date that the position was created. If we could somehow subtract these two, we'd have the number of days that the position has been open. Fortunately, it's easy to get both of these values:

- For the current date, we can simply use the platform's built-in `NOW()` function. `NOW()` returns the exact date and time that the most recent page was loaded.
- For the date that the position was created, we can use the `CreatedDate` field. This field is automatically defined on all objects, and it represents the date that a record was created.

Now that we have our two dates, we want to subtract them: `NOW() - CreatedDate`. But what if the two dates span different months or years? Fortunately, the AppExchange platform is sophisticated enough to know how to handle all the intricacies of such a calculation behind the scenes. We just have to provide the dates, and the platform can do all the rest.

So far so good, but one problem still remains—for our field, we just want to know the number of days that the position has been open, not the number of days, hours, minutes, and seconds. Since `NOW()` returns a date/time field with that level of granularity, our subtraction operation will also return a field with that kind of granularity if we don’t do anything else. How can we just get an approximation of the number of days?

Once again, all we need to do is dip into the extensive library of AppExchange platform functions. The `ROUND()` function returns the closest value to a number that you specify, constraining the new number by a specified number of digits. The `ROUND()` function’s syntax looks like this:

```plaintext
ROUND(number, num_digits)
```

Replacing `number` with our subtraction operation, and `num_digits` with 0, our formula will now look like this:

```plaintext
ROUND(NOW() - CreatedDate, 0)
```

Great! Our formula calculates the number of days a position has been open, rounded to zero decimal places. Now, let's go define a field for it on our Position object.

### Try It Out: Defining a "Days Open" Custom Formula Field

We'll begin building the formula field the same way we created our other custom fields.

1. Click **Setup ➤ Build ➤ Custom Objects**.
2. Click **Position**.
3. In the Custom Fields & Relationships related list, click **New**.
4. Select the **Formula** data type, and click **Next**.
Step 2 of the New Custom Field wizard appears.

5. In the **Field Label** field, enter Days Open.

6. Select the **Number** formula return type.

In this case, even though we're subtracting two Date/Time fields, we want to end up with just a regular numeric value.

7. Change the **Decimal Places** value to 0, and click **Next**.

Now it's time to enter the details of our formula.

8. Click the Advanced Formula tab, as shown in the following screenshot.

![Figure 25: Custom Formula Field Wizard Step 2](image)

We want to use the Advanced Formula tab so we can access the platform's built-in functions through the Functions list on the right side.

9. From the Functions list, double-click **ROUND**.
Our formula now looks like this:

\[
\text{ROUND(number, num\_digits)}
\]

Let's go ahead and replace \text{number} with our \text{NOW()} function and the \text{CreatedDate} merge field.

10. Erase \text{number} from the formula, but leave your cursor there.
11. From the Functions list, double-click \text{NOW}.
12. Type a minus sign (-).
13. From the \text{Select Field Type} drop-down list, choose the object \text{Position\_c}.
14. From the \text{Insert Field} drop-down list, choose the field \text{Created Date}.

Our formula now looks like this:

\[
\text{ROUND(NOW() - CreatedDate, num\_digits)}
\]

Finally, let's replace \text{num\_digits}.

15. Replace \text{num\_digits} with 0.

Our formula now matches our original:

\[
\text{ROUND(NOW() - CreatedDate, 0)}
\]

Now that we've gone through those steps of the procedure, note that we could have just typed in the formula that we figured out in the last section. However, now you know how easy it is to use the Functions library and insert merge fields! Let's keep going and finish up this field:

16. Click \text{Check Syntax} to check your formula for errors.
17. In the \text{Description} text box, enter "Number of days position has been open."
18. Select \text{Treat blank fields as blanks}, and click \text{Next}.
19. Accept all remaining field-level security and page layout defaults.
20. Click \text{Save}.

\section*{Look at What We've Done}

Let's revisit the Positions tab to take a look at the custom formula field that we've just made.

1. Click the Positions tab.
2. Click \text{New}.

Our formula field doesn't show up on our Position Edit page—that's because it's a formula field and doesn't require any user input to display. In order to see it, we'll have to define our first position record. Let's do that now.

3. Enter any values you want to define a new position.
4. Click \text{Save}.

Your new position is now displayed in its own record detail page. At the bottom of the page, notice our \text{Days Open} formula field, just above the \text{Created By} field. It should show 0, since we just created the position.

The Positions tab is fully functional! But are the fields where we want them? Are the fields that must have values marked as required? In the next section, we'll fine-tune our Position custom object by modifying its page layout.
Introducing Page Layouts

After defining all those fields we now have a fully functional Position custom object. However, it doesn’t look all that nice—all of the long text areas appear at the top, and it’s hard to scan. Let’s move some things around to make this page easier for our users. We can do that by customizing the Position object’s page layout.

A page layout controls the position and organization of the fields and related lists that are visible to users when viewing a record. Page layouts also help us control the visibility and editability of the fields on a record. We can set fields as read-only or hidden, and we can also control which fields require users to enter a value and which don’t.

Page layouts are powerful tools for creating a good experience for our users, but it’s crucial that we remember one important rule: page layouts should never be used to restrict access to sensitive data that a user shouldn’t view or edit. Although we can hide a field from a page layout, users can still access that field through other parts of the app, such as in reports or via the AppExchange API. (We’ll learn more about security that covers all parts of the app in Chapter 7: Securing and Sharing Data on page 81.)

Now let’s see if we can organize the fields on our Position object in a way that’s more user-friendly.

Becoming Familiar with the Page Layout Edit Page

First let’s take a look at the Page Layout Edit page:

1. Click Setup ➤ Build ➤ Custom Objects.
2. Click Position.
3. In the Page Layouts related list, click Edit next to the Position Layout.

Welcome to the Page Layout Edit page! As you can see, this editor is different from the ones that we’ve already used in other areas of the platform. That’s because we’re designing a user interface and need to see a representation of how our page will look as we’re working. Before going any further, let’s give ourselves a quick orientation to how this page is set up.
On the left side of this page we can see a representation of what our page currently looks like. After an initial set of buttons, the Information section shows all of the fields we’ve created, plus an additional field for Owner that’s in its own column. Below it is another section for System Information, then Custom Links, and finally, Related Lists.

As we scroll down to view the entire page, the boxes on the right side move with us so that they’re always visible. These boxes include a legend for all of the icons we see on the left, plus a list of all of the Position fields, related lists, and other components that are available to put on our layout. Currently all of the fields in the box on the right are grayed out—that’s because they’ve all been placed on our page layout by default. If we were to move one of the fields that’s in the layout to the left back to the Position fields box on the right, that field would effectively be removed from the layout, and its name would no longer be grayed out.

Now that we know what we're looking at, let's rearrange the fields in the way a user might want to see them.

**Try It Out: Grouping Fields into a New Section**

Let's start modifying our page layout by first defining a new section for salary information. On a page layout, a section is simply an area where we can group similar fields under an appropriate heading. This makes it easy for our users to quickly identify and enter the information for a record, especially if our object has a large number of fields:
1. Click **Create New Section**.
2. In the **Name** text box, enter Compensation.

   The **Name** field controls the text that's displayed as the heading for the section.

3. In the **Columns** drop-down list, choose 2 (Double).

   This option allows us to choose whether we want the fields in our section to be arranged in two columns or one. The default is two columns and is the most commonly-used choice. However, if our section is going to contain text area fields, the one-column layout gives them more space on the page for display.

4. In the **Tab Order** drop-down list, choose Left-Right.

   This setting controls the direction that a user's cursor will move when using the Tab key to navigate from field to field.

5. Select the options for **Show Section Heading on Detail Page** and **Show Section Heading on Edit Page**.

6. Click **OK**.

   Voilà! We have a new section for Compensation. Let's move it above the System Information section and add the Min Pay and Max Pay fields:

7. Click the heading of the Compensation section and drag it above the System Information section.

8. Now use drag-and-drop to move the Min Pay and Max Pay fields from the Information section to the new Compensation section, as shown below.

   ![Figure 28: Dragging and Dropping Fields in a Page Layout](image)

Now that we've gone through the process for building one section, repeat the process to create a new one-column Description section below the Compensation section. Once it's created, drag these fields into it: Job Description, Responsibilities, Skills Required, and Educational Requirements.

**Tip:** You can use Shift+click or Ctrl+click to select all of these fields together and then drag them as one unit into the Description section.

That's much better—our fields are organized, and it's easy to locate all of the information we need. Now all we have left to do is make the Min Pay and Max Pay fields required when a user defines a new position. Once we do that, we'll be all done with our Position object!
Try It Out: Editing Field Properties

Let's make the Min Pay and Max Pay fields required:

1. On the Page Layouts Edit page, double-click the Min Pay field.

This popup window allows us to edit the Min Pay field's properties. We can set the field to read-only and/or required:

- If it's read-only, a user who views a Position record edit page won't be able to change its value.
- If it's required, a user won't be able to create a Position record without specifying a value.

If we didn't want a user to see the Min Pay field at all, we could simply drag it off the layout and onto the Position Fields box on the right side.

**Caution:** Don't forget our earlier warning! Page layouts should never be used to restrict access to sensitive data that a user shouldn't view or edit. That's because page layouts control only a record's edit and detail pages; they don't control access to fields in any other part of the platform.

2. Select the Required checkbox, and click OK.
3. Repeat these steps for the Max Pay field.
4. Click Save to finish customizing the page layout.

Hooray! We're all done with our Position object's page layout!

Look at What We've Done

Congratulations. We've just built ourselves a simple Recruiting app that tracks details about an organization’s open job postings. Let's check out what we've done by revisiting the Positions tab and clicking New. Because of the changes that we've made in our page layout, our Position edit page should now look like this:
We have an object with a tab, we've added custom fields, and we've arranged them in a page layout. We've finished our simple app, and now we're well on our way to creating the more-complex Recruiting app that we described earlier.

Now things are going to get even more interesting. In the next chapter, we'll add a few more custom objects to track things like candidates, job applications, and reviews, and then we'll enhance our Recruiting app even further by defining how our objects relate to one another. Before you know it, we're going to have ourselves an incredibly powerful tool, all implemented with a few clicks in the AppExchange platform.
Chapter 6

Expanding the Simple App Using Relationships

In this chapter ...

• Introducing Relationships
• Introducing Lookup Relationship Custom Fields
• Adding Candidates to the Mix
• Creating a Many-to-Many Relationship
• Managing Review Assessments
• Putting it All Together

So far we've accomplished a fair amount—we've created the Recruiting app and built out a fully functional Position custom object with a tab and several types of fields. It's a good start, but there's more to do.

Having just one object in our Recruiting app is like having a party with just one guest—not all that interesting! We need to invite more "people" to the party by building custom objects to represent candidates, job applications, and reviews, and, even more importantly, we need to create relationships between them. Just like a party isn't all that fun if you don't know any of the other guests, an app isn't all that powerful unless its objects have links to other objects in the app. That's going to be the focus of this chapter, so let's get started!
Introducing Relationships

So what is a relationship, and why are they important for our app? Just as a personal relationship is a two-way association between two people, in terms of relational data, a relationship is a two-way association between two objects. Without relationships, we could build out as many custom objects as we could think of, but they'd have no way of linking to one another.

For example, after building a Position object and a Job Application object, we could have lots of information about a particular position and lots of information about a particular candidate who's submitted an application for it, but there would be no way of seeing information about the job application when looking at the Position record, and no way of seeing information about the position when looking at the Job Application record. That's just not right!

With relationships, we can make that connection and display data about other related object records on a particular record's detail page. For example, once we define a relationship between the Position and Job Application objects we just talked about, our Position record can have a related list of all the job applications for candidates who have applied for the position, while a Job Application record can have a link to the positions for which that candidate is applying. Suddenly the "people" at our Recruiting app "party" know some of the other guests, and the app just got a lot more interesting.

Figure 30: Relationships Allow Information about Other Object Records to be Displayed on a Record Detail Page

Introducing Lookup Relationship Custom Fields

As we learned in Chapter 3: Reviewing Database Concepts on page 25, we can define a relationship between two objects through the use of common fields. On the AppExchange platform, we can define relationships between objects by creating a lookup relationship custom field from one object to another. A lookup relationship field is a custom field on an object record that contains a link to another record. When we place a lookup relationship custom field on an object, we're effectively creating a many-to-one relationship between the object on which the lookup relationship field is placed and the other object.

For example, if we placed a lookup relationship field on a Job Application object that referenced Position object records, many Job Application records could be related to a single Position record. This would be reflected both
with a new Position field on the Job Application record page and with a new Job Applications related list on the Position record detail page.

That's the sort of thing that we're going to do in this chapter. First, let's start with the really quick and easy example of putting a Hiring Manager field on our Position object—we'll create a many-to-one relationship between the Position object and the standard User object that comes with every AppExchange organization, reflecting the fact that a hiring manager can be responsible for several positions at a time. Then we'll build out a few more objects and implement a more complex relationship involving Positions, Job Applications, and Candidates.

Try It Out: Relating Hiring Managers to Positions

For our first relationship, let's associate a hiring manager with a position by putting a lookup relationship field on the Position object. The lookup field will allow users to select the hiring manager for the position by selecting from all the users of the Recruiting app.

For example, if Ben Stuart, our recruiter, wants to assign Anastasia O'Toole as the hiring manager for the Benefits Specialist position, he'll be able to do so by clicking the lookup icon ( disparities) next to the lookup relationship field that we create. Her name will then appear on the Position detail page.

To create the lookup relationship field that accomplishes this, we'll need to go back to the now familiar Position object detail page.

1. Click Setup ➤ Build ➤ Custom Objects.
2. Click Position.
3. In the Custom Fields & Relationships related list, click New.
4. Select Lookup Relationship, and click Next.
5. In the Related To drop-down list, choose User, and click Next.
6. In the Field Label text box, enter Hiring Manager. Once you move your cursor, the Field Name text box should be automatically populated with Hiring_Manager.
7. Click Next.
8. Accept the defaults in the remaining two steps of the wizard.
9. Click Save.

Look at What We've Done

Now return to the Positions tab, and click New. The Position Edit page includes a new Hiring Manager lookup field! If you click the lookup icon next to this field ( disparities), you can search through all of the users of the Recruiting app and select one as the hiring manager. That user's name now appears on the Position record:
As you can see, it was easy to set up this simple relationship between Positions and Users. And as a general rule, you'll find that relationships are pretty easy to set up.

What gets a little tricky is when we start wanting to create relationships that don't represent a simple one-to-one or many-to-one relationship. We'll see an example of one of those in a little bit. Right now, let's build a custom object for candidates so we'll be able to create some more relationships in our Recruiting app.

**Adding Candidates to the Mix**

Let's add a Candidate custom object to our app so we can manage the information about our candidates. We'll also add fields to the object, modify the page layout properties, and create a Candidate record. The process for creating the Candidate custom object is almost identical to what we followed to create the Position custom object, so we'll zip through this quickly.

**Try It Out: Creating the Candidate Object**

To create our Candidate custom object, navigate back to Setup ➤ Build ➤ Custom Objects, click New Custom Object, and fill out the page according to the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Candidate</td>
</tr>
<tr>
<td>Plural Label</td>
<td>Candidates</td>
</tr>
<tr>
<td>Object Name</td>
<td>Candidate</td>
</tr>
<tr>
<td>Description</td>
<td>Represents an applicant who might apply for one or more positions</td>
</tr>
<tr>
<td>Record Name</td>
<td>Candidate Number</td>
</tr>
<tr>
<td>Data Type</td>
<td>Auto Number</td>
</tr>
<tr>
<td>Display Format</td>
<td>C-{00000}</td>
</tr>
<tr>
<td>Starting Number</td>
<td>C-{00001}</td>
</tr>
<tr>
<td>Allow Reports</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow Activities</td>
<td>Yes</td>
</tr>
</tbody>
</table>
To create the Candidates tab, select a Tab Style in the first step of the wizard, and then accept all the defaults.

The Recruiting app now has three tabs: Home, Positions, and Candidates. Now let’s add some custom fields to the Candidate object.

**Try It Out: Adding Fields to the Candidate Object**

To create custom fields on the Candidate object, click Setup ➤ Build ➤ Custom Objects, and click Candidate to view its detail page. In the Custom Fields & Relationships related list, use the New button to create custom fields according to the following table. Where necessary, we’ve indicated some additional values you’ll need to fill in. Otherwise, you can simply accept all defaults.

**Note:** There are a lot of custom fields on the Candidate object. If you don't care about having your app match the screenshots in this book, you only need to bother with the first four fields in the table below.

### Table 8: Candidate Object Custom Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Label</th>
<th>Other Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>First Name</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Text</td>
<td>Last Name</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Phone</td>
<td>Phone</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td>Email Address</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>Street</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Text</td>
<td>City</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Text</td>
<td>State/Province</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Text</td>
<td>Zip/Postal Code</td>
<td>Length: 15</td>
</tr>
<tr>
<td>Text</td>
<td>Country</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Text</td>
<td>Current Employer</td>
<td>Length: 50</td>
</tr>
<tr>
<td>Number</td>
<td>Years of Experience</td>
<td>Length: 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decimal Places: 0</td>
</tr>
<tr>
<td>Number</td>
<td>SSN</td>
<td>Length: 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decimal Places: 0</td>
</tr>
<tr>
<td>Data Type</td>
<td>Field Label</td>
<td>Other Values</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Picklist</td>
<td>Education</td>
<td>Picklist values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HS Diploma</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• BA/BS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MA/MS/MBA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Ph.D.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Post Doc</td>
</tr>
<tr>
<td>Picklist</td>
<td>Gender</td>
<td>Picklist values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Male</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Female</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Currently Employed</td>
<td>Default: Checked</td>
</tr>
<tr>
<td>Checkbox</td>
<td>US Citizen</td>
<td>Default: Checked</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Visa Required</td>
<td>Default: Unchecked</td>
</tr>
<tr>
<td>Phone</td>
<td>Mobile</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>Fax</td>
<td></td>
</tr>
</tbody>
</table>

**Try It Out: Modifying the Candidate Page Layout Properties**

To finish up with this object, let’s organize all of our fields on the page layout and mark some fields as required. To do so, let’s go to the Page Layout Properties page.

1. Click **Setup ➤ Build ➤ Custom Objects**.
2. Click **Candidate**.
3. In the Page Layouts related list, click **Edit** next to the Candidate Layout.
4. Create three new double-column sections below the Information section: Address, Employment, and Additional Details. Drag the appropriate fields into them, as shown in Figure 32: Candidate Object Page Layout on page 71, and don’t forget to click **Quick Save** so you can save your work as you go.
5. Set the **First Name**, **Last Name**, and **Email Address** fields to required:
   1. Use Ctrl+click to select all three required fields.
   2. Click the **Edit Properties** button.
   3. Select the **Required** checkbox in the Select All row, and click **OK**.
6. Click **Save**.

Your Page Layout Properties page should now look similar to the following screenshot.
Look at What We've Done
Here's a quick way to verify that you did everything correctly.

1. Click the Candidates tab.
2. Click New.
3. Create a new record for a candidate named Ethan Tran.
4. Enter a value for each of the required fields.
5. Click Save.

How does the page layout look? Are the fields where you want them? If you were able to successfully create a new Candidate record, and everything looks okay, let’s move on—we have more relationships to create!

Creating a Many-to-Many Relationship

Now that we've got two custom objects in our Recruiting app, the only logical thing that remains for us to do is relate them, right?

Of course! However, unlike the simple relationship that we created between hiring managers and positions, the relationship between positions and candidates is going to be a little more complex.

Why? Let’s think about how these two objects relate to one another:

• A candidate can apply to many positions.
• A position can have many candidates apply for it.

Instead of a many-to-one relationship, such as the one that exists between positions and hiring managers, we've got a many-to-many relationship. We want a candidate to be able to apply for multiple positions and a position to have multiple candidates, but our lookup relationship field only allows us to create a one-to-many relationship. What would the "one" side be?

Here's where we get a little creative. Earlier in the chapter you might recall us talking about a Job Application object. A job application fits into the space between candidates and positions—one candidate can submit many job applications, and one position can receive many job applications, but a job application always represents a
request from a single candidate for a single position. In essence, the Job Application object has a many-to-one relationship with both the Candidate object and the Position object.

The Job Application object is the key to making a many-to-many relationship. Much like flour, yeast, salt, and water can be combined to form a loaf of bread, we can combine two lookup relationships and a junction object like Job Application to make a many-to-many relationship. Even though we don’t have a way of directly relating candidates and positions, the Job Application object brings them together.

For example, let’s look at a typical scenario at Universal Containers. Ethan is a candidate who applies for the Project Manager position and the Sr. Developer position. Bonnie is a candidate who applies for the Sr. Developer position. Every time Ethan or Bonnie apply for a position, a Job Application record tracks the exact position for which they're applying. As you can see in the diagram below, Ethan can apply to many positions, and Bonnie and Ethan (or many candidates) can apply to the same position.

![Figure 33: Using a Job Application Object to Create a Many-to-Many Relationship Between Candidates and Positions](image)

In relational database terms, each Job Application record is a row in the Job Application table, consisting of a foreign key to a Candidate record and a foreign key to a Position record. The following entity relationship diagram shows this relationship.

![Figure 34: Entity Relationship Diagram for the Candidate, Job Application, and Position Objects](image)

Consequently, in order to define a many-to-many relationship between the Candidate and Position objects, we'll need to create a Job Application object with the following additional fields:

- A Candidate lookup relationship
- A Position lookup relationship

Let’s go do that now.
Try It Out: Creating the Job Application Object

You should be a pro at this by now! To create our Job Application custom object, navigate back to Setup ➤ Build ➤ Custom Objects, click New Custom Object, and fill out the page according to the following table.

Table 9: Values for Defining the Job Application Object

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Job Application</td>
</tr>
<tr>
<td>Plural Label</td>
<td>Job Applications</td>
</tr>
<tr>
<td>Object Name</td>
<td>Job_Application</td>
</tr>
<tr>
<td>Description</td>
<td>Represents the junction object between a candidate and a position</td>
</tr>
<tr>
<td>Record Name</td>
<td>Job Application Number</td>
</tr>
<tr>
<td>Data Type</td>
<td>Auto Number</td>
</tr>
<tr>
<td>Display Format</td>
<td>JA-{00000}</td>
</tr>
<tr>
<td>Starting Number</td>
<td>JA-{00001}</td>
</tr>
<tr>
<td>Allow Reports</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow Activities</td>
<td>Yes</td>
</tr>
<tr>
<td>Track Field History</td>
<td>Yes</td>
</tr>
<tr>
<td>Deployment Status</td>
<td>Deployed</td>
</tr>
<tr>
<td>Add Notes &amp; Attachments related list to default page layout</td>
<td>Yes</td>
</tr>
<tr>
<td>Launch New Custom Tab Wizard after saving this custom object</td>
<td>Yes</td>
</tr>
</tbody>
</table>

To create the Job Applications tab, select a Tab Style in the first step of the wizard, and then accept all the defaults.

That was simple enough, but we're not quite done. We need to create the lookup relationship fields between the Job Application object and the Position and Candidate objects.

Try It Out: Adding Fields to the Job Application Object

Here's another procedure that we've done several times before, but this time we only need to define three custom fields instead of the nearly twenty that we built for the Candidate object. We'll need to add the two lookup relationship fields that will create relationships between the Job Application object and the Position and Candidate objects, and we'll also need to add a picklist field so that we can track the application's status.

Although these fields are almost identical to the ones we created earlier, you'll notice when you're defining the lookup relationship fields that there's a new step in the custom field wizard: Step 6: Add Custom Related Lists. This step of the wizard is where we can specify a heading for the Job Applications related list that will show up on both the Candidate and Position detail pages when we add it to their page layouts.
Why didn't we see this step earlier when we created our Hiring Manager lookup field? It turns out that because User is a standard object without a tab or a detail page of its own, we'll never have a need for a Positions related list. The platform knows this, so it leaves out the related list step whenever someone adds a lookup relationship field that references the User object.

Now that we're all squared away with that small difference, let's finish up these Job Application fields. Click Setup ➤ Build ➤ Custom Objects, and then click Job Application to view its detail page. In the Custom Fields & Relationships related list, use the New button to create custom fields according to the following table. Where necessary, we've indicated some additional values you'll need to fill in. Otherwise you can simply accept all defaults.

**Table 10: Add Custom Fields to the Job Application Object**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Label</th>
<th>Other Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup Relationship</td>
<td>Candidate</td>
<td>Related To: Candidate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Related List Label: Job Applications</td>
</tr>
<tr>
<td>Lookup Relationship</td>
<td>Position</td>
<td>Related To: Position</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Related List Label: Job Applications</td>
</tr>
<tr>
<td>Picklist</td>
<td>Status</td>
<td>Picklist values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Phone Screen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Schedule Interviews</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Extend an Offer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Offer Accepted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hired</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Rejected</td>
</tr>
</tbody>
</table>

**Look at What We've Done**

Voilà! If you click on the new Job Applications tab and click New, you'll see the Candidate lookup field, a Position lookup field, and a Status picklist field.
But there's more! Because we've built a couple of lookup relationships, our Candidate and Position record detail pages now each have a new Job Applications related list. And the Job Application detail page includes links to the Candidate and Position records that it references. All of our objects are now related and linked to one another!

Now we're nearly finished building all the objects and relationships that we need for our Recruiting app. We simply need to create one more custom object that'll provide our hiring managers and interviewers with a place to enter their comments about candidates.

Managing Review Assessments

Interviewers, recruiters, and hiring managers need to be able to create reviews so that they can record their comments about each candidate. They also need to see the reviews posted by other people, as well as make comments on them if they want to add details. To allow our users to perform these tasks, we'll need to create a custom Review object and relate it to the Job Application object.

The Review object has a many-to-one relationship with the Job Application object because one job application can have one or more reviews associated with it. Once again, we'll use a lookup relationship field on the object to create the relationship. A related list on the Job Application will show the associated reviews, representing the "many" side of the relationship.

Figure 36: Job Application Links to Position and Candidate Data
Try It Out: Creating the Review Object

For the last time, let’s create a new custom object. Navigate back to **Setup ➤ Build ➤ Custom Objects**, click **New Custom Object**, and fill out the page according to the following table.

**Table 11: Values for Defining the Review Object**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Review</td>
</tr>
<tr>
<td>Plural Label</td>
<td>Reviews</td>
</tr>
<tr>
<td>Object Name</td>
<td>Review</td>
</tr>
<tr>
<td>Description</td>
<td>Represents an interviewer’s assessment of a particular candidate</td>
</tr>
<tr>
<td>Record Name</td>
<td>Review Number</td>
</tr>
<tr>
<td>Data Type</td>
<td>Auto Number</td>
</tr>
<tr>
<td>Display Format</td>
<td>R-{00000}</td>
</tr>
<tr>
<td>Starting Number</td>
<td>R-{00001}</td>
</tr>
<tr>
<td>Allow Reports</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow Activities</td>
<td>Yes</td>
</tr>
<tr>
<td>Track Field History</td>
<td>Yes</td>
</tr>
<tr>
<td>Deployment Status</td>
<td>Deployed</td>
</tr>
<tr>
<td>Add Notes &amp; Attachments related list to default page layout</td>
<td>Yes</td>
</tr>
<tr>
<td>Launch New Custom Tab Wizard after saving this custom object</td>
<td>No</td>
</tr>
</tbody>
</table>

Notice that we didn’t launch the tab wizard this time. Reviews don’t need a tab of their own, because they can be accessed via a related list on the Job Application detail page. When you create an object with a tab, the platform provides access to that object’s records in various places other than just the tab, such as in search results and the Recent Items list in the sidebar area of every page. Because most Recruiting app users won’t need to see reviews unless it’s in the context of a job application, we don’t need to create a separate tab for them.
Now let's finish up the custom fields on the Review object.

**Try It Out: Adding Fields to the Review Object**

For our Review object we only need to build two fields: a lookup relationship field to the Job Application object and a text area field for entering the reviewer's assessment.

Click **Setup ➤ Build ➤ Custom Objects**, and then click **Review** to view its detail page. In the Custom Fields & Relationships related list, use the **New** button to create custom fields according to the following table. Where necessary, we've indicated some additional values you'll need to fill in. Otherwise, you can simply accept all defaults.

**Table 12: Add Custom Fields to the Review Object**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Label</th>
<th>Other Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup Relationship</td>
<td>Job Application</td>
<td>Related To: Job Application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Related List Label: Reviews</td>
</tr>
<tr>
<td>Text Area (Long)</td>
<td>Assessment</td>
<td>Length: 32,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td># of Visible Lines: 6</td>
</tr>
</tbody>
</table>

**Look at What We've Done**

Terrific! Let's go see what we've made:

1. Click the Job Applications tab and select a record, or create one if you haven't already. Since we related the Review object to the Job Application object, the Reviews related list now appears on the Job Application detail page.
2. In the Reviews related list, click **New** to create a review.

Do you see how the AppExchange platform automatically filled in the job application number in the Review's edit page? That's one of the small, but important benefits of using the platform to build an application like this—not only is it easy to create links and relationships between objects, but the platform anticipates what we're doing and helps us accomplish our task with as few clicks as possible.

**Putting it All Together**

We just created several objects and a lot of relationships. The following simple diagram shows us what we've accomplished so far.
If you're interested, you can take a look at the entity relationship diagram shown here.

We've now built all of our Recruiting app objects and tabs, and we've defined lots of custom fields—everything from text fields and picklists to more complex formula fields and lookup relationship fields. We've created a
robust user interface so that our recruiters and hiring managers can enter and retrieve data about positions and related candidates, job applications, and reviews, and we did all of this without writing a single line of code!

Remember when we assigned Anastasia O'Toole as the hiring manager for the Benefits Specialist position? Let's look at what Anastasia can do now: She can create and update her positions, and she can view other positions. She can look at details about any candidates who've applied for the Benefits Specialist job, and she can review their related job applications. She can also check the status of the job applications. She no longer has to go to Human Resources to search through Microsoft Word documents and spreadsheets to manage her tasks in the hiring process. The Recruiting app is well on its way to becoming a fully-functional and useful application!

However, before we leave this chapter behind, let's get ourselves prepared for the rest of this book by creating and importing some real data. It'll help us when we get to our next chapter on security and sharing if we have some records that we can work with.

Try It Out: Downloading Sample Data

In addition to entering data via our tabbed pages, we can also use the handy Import Wizard to import multiple records at a time. The ability to easily import data into your custom objects is one of the AppExchange platform's key benefits. Let's download some sample data so we can add more records to our custom objects without tons of typing.

1. Download the zip file containing the sample CSV (comma-separated values) import files from www.salesforce.com/developer/devguide.
2. Extract the zip file to C:\dev\recruiting (or any directory on your computer).
3. Go to C:\dev\recruiting. This directory contains three CSV files: Positions.csv, Candidates.csv, and JobApplications.csv. (The directory also contains one other file that you'll use later when we build composite components for our app in Chapter 9: Moving Beyond Native Apps on page 131.)

Before we import anything, we need to make a modification to the import file for Positions. The sample Positions.csv you downloaded contains fictional users in the Hiring Manager column. The names of these users most likely won't match any user in your organization, and if you import the file "as is," the Import Wizard won't be able to find any matching users, and the Hiring Manager field on each Position record will be left blank. So let's go ahead and make that change.

4. Go to C:\dev\recruiting, and open Positions.csv in Excel, a text editor, or any other program that can read CSV files.
5. In the Hiring Manager column, replace the fictional users with the first and last name of a user in your organization.
6. Save the file, making sure to maintain the CSV format.

Try It Out: Using the Import Wizard

Now let's walk through the process of importing Position records using the Import Wizard and the Positions.csv file you downloaded.

1. Click Setup ➤ Data Management ➤ Import Custom Objects.
2. Click Start the Import Wizard! The Import Wizard appears.
3. Select Position for the type of record you're importing, and click Next.
4. To prevent duplicate Position records from being created as a result of this import, choose Yes. Accept the other defaults for matching, and click **Next**.

5. Select None for the record owner field. We didn't include a User field in the CSV file to designate record owners. The Import Wizard will assign you as the owner of all new records.

6. Choose the Hiring Manager lookup relationship field so you can link Position records with existing User records in the Recruiting app, and click **Next**.

7. Select Name as the field you want to match against as the Import Wizard compares Hiring Manager names in your import file with User names in the system, and click **Next**.

8. Click **Browse**, and find `C:\dev\recruiting\Positions.csv`. Click **Next**.

9. Use the drop-down lists to specify the Salesforce fields that correspond to the columns in your import file. For your convenience, identically matching labels are automatically selected. Click **Next**.

10. Click **Import Now!**.

Now you can repeat this import process to add Candidate records using the `Candidates.csv` file you downloaded. As you’re going through the Import Wizard, there are two differences for this object: 1) in step 2 of the wizard, select No when prompted to prevent duplicates; and 2) you’ll notice the wizard skips the two steps about lookup relationship field matching. Since the Candidate object doesn’t have any lookup relationship fields, the Import Wizard automatically leaves these steps out.

Let’s do it once more to add some Job Application records using the `JobApplications.csv` file. There are a few differences this time too: 1) in step 2 of the wizard, select No when prompted to prevent duplicates; 2) select both the Candidate and Position lookup fields in step 3; and 3) select the Candidate Number and Position Title fields to use for matching the lookup fields.

We’ve just added a bunch of data to our app without a lot of work. In the next chapter, we’ll take a look at all the ways we can control access to this data using the built-in tools of the AppExchange platform. We’ll get into the nitty-gritty about security, sharing rules, permissions, roles, and profiles.
Chapter 7

Securing and Sharing Data

In this chapter ...

• Controlling Access to Data in Our App
• Data Access Concepts
• Controlling Access to Objects
• Controlling Access to Fields
• Controlling Access to Records
• Putting It All Together

Now that we’ve got a fully functional Recruiting app with all of our object relationships in place, it's time to start thinking about who’s actually going to be using the app and how much access they should have to all of the data that it's going to contain.

As with many apps, our Recruiting app exposes sensitive pieces of data such as social security numbers, salary amounts, and applicant reviews that could really come back to haunt us if they fell into the hands of the wrong people. We need to provide security without making it harder for our recruiters, hiring managers, and interviewers to do their jobs.

Here we're going to see another one of the huge benefits that the AppExchange platform has to offer—simple-to-configure security controls that easily allow us to restrict access to data that users shouldn't see, without a lot of headaches. Similar to Access Control Lists or Windows folder permissions, the AppExchange platform allows us to specify who can view, create, edit, or delete any record or field in the app. In this chapter, we’ll see how we can use the AppExchange platform to implement those rules.
Controlling Access to Data in Our App

As we’ve already seen, there are three types of users who will need to access the data in our Recruiting app: recruiters, hiring managers, and interviewers. To these three, let’s add a fourth type of user—a standard employee who doesn’t perform any interviews and who never needs to hire anyone. (This employee will help us determine the default permissions that should apply to all of the new recruiting objects in our app.)

One by one, let’s take a look at the kinds of access that each one of these users needs and, more important, the kinds of access they don’t need in order to do their jobs. Once we’ve compiled a set of required permissions, we’ll figure out how to implement them in the rest of the chapter.

Required Permissions for the Recruiter

For our first set of required permissions, let’s take a look at Mario Ruiz, a recruiter at Universal Containers. To do his job, Mario needs to be able to create, view, and modify any position, candidate, job application, or review that’s in the system. Likewise, Mario needs to view and modify the recruiting records that all of the other recruiters own, since all of the recruiters at Universal Containers work together to fill every position, regardless of who created it.

Although Mario has the most powerful role in our Recruiting app, we still can’t give him complete free reign. State and federal public records laws require that all recruitment-related records must be saved for a number of years so that if a hiring decision is questioned, it can be defended in court. Consequently, we need to make sure that Mario will never accidentally delete a record that needs to be saved to fulfill the law.

But how will he keep the number of positions, candidates, job applications, and reviews in check if he can’t delete? Won’t the app become swamped with old data? Not if we’re smart about it—instead of having Mario delete old records, we can use the Status field on a record as an indication of whether it’s current. We’ll filter out all of the old records by using a simple list view.

Here’s a summary of the required permissions that we need to implement for a recruiter:

Table 13: Summary of Required Permissions: Recruiter

<table>
<thead>
<tr>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recruiter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
</tr>
<tr>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
</tr>
<tr>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
</tr>
</tbody>
</table>
Position

First of all, Ben likes to post his own positions so that he can publicize them as fast as possible, but in our app, Mario the recruiter ultimately needs to take ownership of the record to make sure the position gets filled. As a result, Ben needs the ability to create positions, but then we’ll need to find a mechanism to make sure that they ultimately get transferred to Mario for ownership. (Hint: as you’ll see in *Chapter 8: Using Custom Workflow* on page 115, we’ll tackle that problem with a workflow rule that transfers position ownership to a recruiter when a new position is created by a hiring manager. For now just assume that this already works.)

Ben should also be able to update and view all fields for positions for which he’s the hiring manager, but he should only be able to view other managers’ positions.

Candidate

Ben sometimes wants to poach a prime candidate who's applying for a position under another manager, but this is a practice that Universal Containers frowns upon. As a result, Ben should only be able to view those candidates who have applied for a position on which he’s the hiring manager. Also, since Ben has no reason to see a candidate’s social security number, this field should be restricted from his view.

Job Application

Because Ben is restricted to viewing only those candidates who have applied for a position on which he’s the hiring manager, he should likewise view only the job applications that relate those candidates to his open positions. As the hiring manager, he needs to be able to update the status of those job applications to specify which candidates should be selected or rejected.

Review

To make a decision about the candidates who are applying, Ben needs to see the reviews posted by the interviewers, as well as make comments on them if he thinks the interviewer was being too biased in his or her review. Likewise, Ben needs to be able to create reviews so that he can remember his own impressions of the candidates he interviews.

Here’s a summary of the required permissions we need to implement for a hiring manager:

<table>
<thead>
<tr>
<th>Table 14: Summary of Required Permissions: Hiring Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Position</strong></td>
</tr>
<tr>
<td><strong>Hiring Manager</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

* Only for those records that are associated with a position to which the hiring manager has been assigned

Required Permissions for the Interviewer

For our third set of required permissions, let’s take a look at Melissa Lee’s role as an interviewer. Ben, her manager, likes Melissa to interview candidates for highly technical positions, but doesn’t want her speaking with folks who are applying for roles on the user interface team. As a result, Melissa should be able to view only the candidates

83
and job applications to which she’s assigned as an interviewer. No such restriction needs to exist on the positions that are out there, but she shouldn’t be able to view the minimum and maximum salary values for any of them. Likewise, she shouldn’t see the social security number of any candidate, since it’s sensitive information that has nothing to do with her job.

Melissa must be able to create and edit her reviews so that she can record her comments about each candidate, but she shouldn't be able to see the reviews of other interviewers—reading them might sway her opinion one way or the other. As with hiring managers and recruiters, Melissa also shouldn't be allowed to delete any records to ensure that public records laws are fulfilled.

Here’s a summary of the required permissions we need to implement for an interviewer:

**Table 15: Summary of Required Permissions: Interviewer**

<table>
<thead>
<tr>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
</table>
| Interviewer  | • Read (No min/max pay) | • Read* (No SSN) | • Read* | • Read**  
|              |                 |                 |        | • Create  
|              |                 |                 |        | • Edit**  |

* Only for those records that are associated with a position to which the interviewer has been assigned

** Only for those records that the interviewer owns

**Required Permissions for the Standard Employee**

Employees, such as Manny Damon on the Western Sales Team, are often the best resources for recruiting new hires, even if they are not active hiring managers or interviewers. For this reason, we need to make sure that employees like Manny can view open positions, but that they can’t see the values for the positions' minimum and maximum salary fields—otherwise they might tip off friends to negotiate for a position’s maximum salary value! Manny also shouldn’t be able to view any other records in our Recruiting app.

Here’s a summary of the required permissions we need to implement for a standard employee:

**Table 16: Summary of Required Permissions: Standard Employee**

<table>
<thead>
<tr>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Employee</td>
<td>• Read (No min/max pay)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**So Where Are We Now?**

Now that we've gone through the required permissions for each of our four users, let's organize our thoughts by summarizing them in the table below. In the rest of this chapter, we'll figure out how we can use the AppExchange platform to implement these rules in our Recruiting app.
**Table 17: Summary of Required Permissions**

<table>
<thead>
<tr>
<th></th>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recruiter</strong></td>
<td>Read</td>
<td>Read</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td>Create</td>
<td>Create</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>Edit</td>
<td>Edit</td>
<td>Edit</td>
<td>Edit</td>
</tr>
<tr>
<td><strong>Hiring Manager</strong></td>
<td>Read</td>
<td>Read* (No SSN)</td>
<td>Read*</td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td>Edit*</td>
<td>Create</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>Edit*</td>
<td></td>
<td>Edit*</td>
<td>Edit</td>
</tr>
<tr>
<td><strong>Interviewer</strong></td>
<td>Read (No min/max pay)</td>
<td>Read* (No SSN)</td>
<td>Read*</td>
<td>Read**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read*</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read*</td>
<td>Edit**</td>
</tr>
<tr>
<td><strong>Standard Employee</strong></td>
<td>Read (No min/max pay)</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* Only for those records that are associated with a position to which the hiring manager/interviewer has been assigned

** Only for those records that the interviewer owns

**Tip:** When implementing the security and sharing rules for your own organization, it's often useful to create a required permissions table like the one above to organize your thoughts and make sure you don't forget to restrict or grant access to a particular user. You'll see that we're going to refer back to this table again and again as we go through this chapter.

## Data Access Concepts

Before we get started implementing our security and sharing rules, let's quickly take a look at all the ways that we can control data in the AppExchange platform:

**Object-Level Security**

The bluntest way that we can control data is by preventing a user from seeing, creating, editing, and/or deleting any instance of a particular type of object, like a Position or Review. Object-level access allows us to hide whole tabs and objects from particular users, so that they don't even know that type of data exists.

In the AppExchange platform, we set object-level access rules with object permissions on user profiles. We'll learn more about profiles in a little bit.

**Field-Level Security**

A variation on object-level access is field-level access, in which a user can be prevented from seeing, editing, and/or deleting the value for a particular field on an object. Field-level access
allows us to hide sensitive information like the maximum salary for a position or a candidate's social security number without having to hide the whole object.

In the AppExchange platform, we set field-level access rules with the field-level security. We'll also learn more about that shortly.

Record-Level Security

To control data with a little more finesse, we can allow particular users to view an object, but then restrict the individual object records that they’re allowed to see. For example, record-level access allows an interviewer like Melissa Lee to see and edit her own reviews, without exposing the reviews of everyone else on her team.

In the AppExchange platform, we actually have four ways of setting record-level access rules:

• **Organization-wide defaults** allow us to specify the baseline level of access that a user has in your organization. For example, we can make it so that any user can see any record of a particular object to which their user profile gives them access, but so that they’ll need extra permissions to actually edit one.

• **Role hierarchies** allow us to make sure that a manager will always have access to the same records as his or her subordinates.

• **Sharing rules** allow us to make automatic exceptions to organization-wide defaults for particular groups of users.

• **Manual sharing** allows record owners to give read and edit permissions to folks who might not have access to the record any other way.
The combination of all of these sharing and security settings in the AppExchange platform means that we can easily specify user permissions for an organization of thousands of users without having to manually configure the permissions for each individual. Pretty neat! Now let’s get started learning more about each of these methods for controlling data, and actually implementing the security and sharing rules for our app.

**Controlling Access to Objects**

First let’s configure access to our Recruiting app custom objects. As we mentioned briefly above, we can control whether a user knows that a particular object exists in the app by modifying his or her profile. But exactly what is a profile, and what does it control?

**Introducing Profiles**

A *profile* is a collection of settings and permissions that determine what a user can do in the AppExchange platform, kind of like a group in a Windows network, where all of the members of the group have the same folder permissions and access to the same software. Profiles control:

- The objects the user can view, create, edit, and delete
- The object fields the user can view and edit (more on that later!)
Profiles are typically defined by a user's job function (for example, system administrator or sales representative), but you can have profiles for anything that makes sense for your organization. A profile can be assigned to many users, but a user can be assigned to only one profile at a time.

### Standard AppExchange Profiles

The AppExchange platform provides the following set of standard profiles in every organization:

- Read Only
- Standard User
- Marketing User
- Contract Manager
- Solution Manager
- System Administrator

Each of these standard profiles includes a default set of permissions for all of the standard objects in available on the platform. For example, users assigned to the Standard User profile can never create, edit, or delete a campaign.

When a custom object is created, all standard profiles automatically get full access to the object by default (except, of course, for the Read Only profile, which gets only read access). The only way to restrict access to a custom object for users assigned to a standard profile is by defining field- or record-level access rules, which we'll learn more about later.

You can find more detailed descriptions of all the standard profiles in the online help, but the important thing to know is that you can never actually edit the permissions on a standard profile. Instead, if you have access to the Enterprise, Unlimited, or Developer Editions of the AppExchange platform, you can make a copy of a standard profile and then customize that copy to better fit the needs of your organization. That's what we're going to end up doing for our Recruiting app (and as a result, Enterprise, Unlimited, and Developer Editions will be the only editions that the Recruiting app will support).

### Custom Profiles in Our Recruiting App

For our app, we've talked about four types of users: recruiters, hiring managers, interviewers, and standard employees. We might just jump the gun and say that this equals four different user profiles, but let's take a closer look.

Recruiters are pretty straightforward—they definitely represent a particular job function, and they need access to different types of data than other users. They need their own profile.

A hiring manager, however, is not exactly a single type of position. For most organizations, a hiring manager in the Sales department will almost certainly need access to a different type of data than a hiring manager in Engineering. However, for the purposes of our app, sales managers and software managers still need the same types of access to recruiting data—reviews, candidates, positions, and job applications. Let's keep this as a single...
profile for now, but if incorporating our app into an organization with other CRM functionality, we'll need to suggest as a best practice that the hiring manager permissions for recruiting-related data need to be replicated for any profile to which hiring managers belong.

Finally, let's look at interviewers and standard employees. Neither one of these user types reflects a particular job function, and when you think about it, just about anyone in an organization might be called upon to perform an interview. Let's define a single profile for a standard employee and find a way to grant interviewers access to the records that they need through some other mechanism. (Hint: we can use a combination of organization-wide defaults and sharing rules to make this work.)

**Try It Out: Creating the Recruiter Profile**

All right—we're finally ready to dig into the app and create our first profile! Let's start with the Recruiter profile.

1. Click **Setup ➤ Manage Users ➤ Profiles**.

   ![Figure 41: Standard Profiles](image)

   Here you should see the list of standard profiles that we talked about earlier. After we create our custom profiles, they'll also show up in this list.

   First, we can quickly tell which profiles we can play with by looking at the **Custom** column—if it's checked, that means it's a custom profile and we can edit anything about it. If that column is not checked, we can still click the **Edit** link; we just can't modify any of the permission settings. (What does that leave for us to edit on a standard profile? Well, we can choose which tabs should appear at the top of a user's page, and we can also select the apps that are available in the AppExchange app menu in the top-right corner of the page.)

2. Create a new profile named Recruiter based on the Standard User profile.

   There are actually two ways of doing this—we can either click **New**, select an existing profile to clone, name it, and click **Save**, or we can simply click **Clone** in the detail page of the profile that we want to copy, name it, and click **Save**. Ultimately, it's the same number of clicks, so choose the method you like best. Standard User is the profile that most closely resembles what we want our new Recruiter profile to look like, so it's a good starting point.

3. In the new Recruiter profile's detail page, click **Edit**.

   The Recruiter detail page should look and function exactly like the Standard User profile edit page except with one important difference: when you click **Edit**, you have the ability to modify any of the permission settings.
4. In the Custom App Settings area, make the Recruiting app visible to users assigned to the Recruiter profile, as shown in the figure below.

Tip: You can also give this profile access to any of the other available apps as well. Every profile needs to have at least one visible app.

When an app is visible, a user can select it from the AppExchange app menu at the top-right corner of the page. Be aware, however, that even if an app is visible, the app's tabs won't show up unless a profile has permissions to view the tabs and permission to view the associated object. (We'll set both of those permissions lower down in the Profile edit page.)

5. Select Default next to the Recruiting app.

Making this selection means that the Recruiting app will be displayed when a user logs in. You'll notice that when you select an app as the default, its Visible checkbox is automatically selected, because it doesn't make sense for an app to be the default if it's not visible to the user.

6. In the Tab Settings area, select Default On for the Position, Job Application, Candidate, and Review tabs.

Tip: You can choose whether you want other tabs to be displayed based on the additional apps that you made visible in the last step.

For the purposes of our Recruiting app, all of our custom recruiting tabs are on by default. For any other tabs that you select, you can choose which should be displayed on top of the user's page (Default On), hidden from the user's page but available when he or she clicks the All Tabs tab on the far right (Default Off), or completely hidden from the user (Tab Hidden).

Realize that even if you completely hide a tab, users can still see the records that would have appeared in that tab in search results and in related lists. (To prevent a user from accessing data, we have to set the proper restrictions...
in the Standard and Custom Object Permissions areas lower down in the Profile edit page—we'll get there shortly!

The **overwrite users' personal tab customizations** setting appears if you have an organization that's currently in use and you want to make sure your existing users are viewing the tabs that you've selected. You don’t need to select this for our app, because we’re defining a brand-new profile and no one has personalized his or her tab visibility settings yet. However, if you do want to select this option at some point in the future, just make sure you're not going to annoy your users by deleting all of their customizations!

![Figure 44: Profile Tab Settings Area](image)

Just below the Tab Settings area, the Administrative and General User Permissions areas of the profile allow you to grant special access to features and functionality that don’t map directly to particular AppExchange objects. None of these permissions affects our Recruiting app, but you can learn more about them in the online help.

7. In the Custom Object Permissions area, specify the object-level permissions for our Recruiter profile according to the table below.

**Table 18: Summary of Required Permissions: Recruiter**

<table>
<thead>
<tr>
<th></th>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recruiter</strong></td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
</tr>
</tbody>
</table>

**Tip:** Depending on the apps that you made visible above, you can also set additional object permissions on standard or other custom objects.

Since there are no instances when a recruiter should be allowed to delete a recruiting-related object, we should make sure that the object-level permissions for deletion are turned off on Positions, Candidates, Job Applications, and Reviews.

![Figure 45: Recruiter Profile Permissions](image)
By restricting the power to delete recruiting-related objects here, recruiters will never be able to delete these objects. However, the fact that we're granting recruiters permission to create, read, or edit our recruiting objects does not necessarily mean that recruiters will be allowed to create, read, or edit every recruiting object record. Why?

Here we see the result of two really important concepts in the world of the AppExchange platform:

• The permissions on a record are always evaluated according to a combination of object-, field-, and record-level permissions.
• When object- versus record-level permissions conflict, the most restrictive settings win.

What this means is that even though we are granting this profile create, read, and edit permissions on the recruiting objects, if the record-level permissions for an individual recruiting record prove to be more restrictive, those will be the rules that will define what a recruiter can access.

For example, our new profile gives a recruiter permission to create, edit, and view reviews. However, if we set organization-wide defaults for reviews to Private (a record-level permission), our recruiter will be allowed to edit and view only his own reviews, and not the reviews owned by other users. We'll learn more about record-level permissions later and go through more examples of how they work with the object-level ones, but for now, just understand that object-level permissions are only one piece of the puzzle.

8. Click **Save** to create your profile and return to the profile list view.

Congratulations! We're done with our first profile. As you can see, it really wasn't that hard, because we'd already analyzed our required permissions and knew what objects recruiters will need access to. In the next section, let’s quickly finish up our other two profiles and then move on to field-level security.

**Try It Out: Creating More Profiles**

Now that we've created our Recruiter profile, let's finish up with profiles for hiring managers and standard employees. As we mentioned earlier, these profiles are probably too generic to be used in a company that's using the AppExchange platform for other, non-recruiting functionality, but for our purposes, they'll work just fine for now.

To make each profile, go ahead and follow the steps that we outlined in the previous section. The important things to remember for these profiles are:

• The profiles should be named Hiring Manager and Standard Employee and should be based on the standard profile that best fits your needs. We used Standard User for our Recruiter profile, and that's probably a good one for Standard Employee as well. However, for your organization you might find that Hiring Manager more closely resembles the Contract or Solution Manager profile instead. It's up to you.
• The Recruiting app for both profiles must be set to **Visible**.
• The tabs for Positions, Candidates, Job Applications, and Reviews should be set to **Default On**.
• Standard and custom object permissions should reflect the required permissions that we worked on before.

To refresh our memories, let's take a look at our required permissions summary table:

**Table 19: Summary of Required Permissions: Hiring Manager, Interviewer, and Standard Employee**

<table>
<thead>
<tr>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hiring Manager</td>
<td>• Read</td>
<td>• Read* (No SSN)</td>
<td>• Read</td>
</tr>
</tbody>
</table>

Chapter 7: Securing and Sharing Data
This table is a little confusing right now, because Interviewer and Standard Employee are still separated into two different users. Let’s go ahead and combine them into a single Standard Employee user so that it’s a little easier to see what object-level permissions we need to grant. It’s easy to do, because Interviewers and Standard Employees have the same permissions on the Position object, and we already have asterisks on Candidates, Job Applications, and Reviews that ensures these users won’t look at anything to which they’re not assigned as an interviewer:

Table 20: Summary of Required Permissions: Combining Interviewers and Standard Employees

<table>
<thead>
<tr>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hiring Manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Read</td>
<td>• Read* (No SSN)</td>
<td>• Read*</td>
<td>• Read**</td>
</tr>
<tr>
<td>• Create</td>
<td>• Edit*</td>
<td>• Edit*</td>
<td>• Create</td>
</tr>
<tr>
<td>• Edit*</td>
<td></td>
<td></td>
<td>• Edit**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Employee</td>
<td>(Interviewer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Read (No min/max pay)</td>
<td>• Read* (No SSN)</td>
<td>• Read*</td>
<td>• Read**</td>
</tr>
<tr>
<td>• Create</td>
<td>• Edit*</td>
<td>• Edit*</td>
<td>• Create</td>
</tr>
<tr>
<td>• Edit*</td>
<td></td>
<td></td>
<td>• Edit**</td>
</tr>
</tbody>
</table>

* Only for those records that are associated with a position to which the hiring manager/interviewer has been assigned

** Only for those records that the interviewer owns

Great, but what about all of those asterisks and restrictions on visible fields? Do we need to take those into account when we set our object-level permissions?

Not at all. Those asterisks and field restrictions represent record- and field-level security settings that we’re going to have to specify elsewhere in our app. The only things we need to care about here are the permissions that these users will need to have access to at least some of the time—that’s the whole point of object-level permissions:
• For hiring managers, that's create, read, and edit on Positions and Reviews, read on Candidates, and read and edit on Job Applications
• For standard employees, that's read on Positions, Candidates, and Job Applications, and create, read, and edit on Reviews

Figure 46: Hiring Manager Profile Permissions

Figure 47: Standard Employee Profile Permissions

Fantastic! We've just finished defining profiles and object-level permissions for all the users in our Recruiting app. However, that's still just one piece of the security and sharing puzzle—we still need to make sure that sensitive data on these objects is protected from users who don’t need access, and then we need to drill down on the actual records that each user should be allowed to view and edit.

Controlling Access to Fields

Now that we've restricted access to objects as a whole, it's time to use a finer-toothed comb to manage the security of individual object fields. These are the settings that allow us to protect sensitive fields such as a candidate's social security number without having to hide the fact that the candidate object even exists.

Introducing Field-Level Security

In the AppExchange platform, we control access to individual fields with field-level security. Field-level security controls whether a user can see, edit, and delete the value for a particular field on an object.

Unlike page layouts, which only control the visibility of fields on detail and edit pages, field-level security controls the visibility of fields in any part of the app, including related lists, list views, reports, and search results. Indeed, in order to be absolutely sure that a user can’t access a particular field, it’s important to use the field-level security page for a given object to restrict access to the field. There are simply no other shortcuts that will provide the same level of protection for a particular field.

Field-Level Security in Our Recruiting App

Just to refresh our memories about what field-level security settings we need for our Recruiting app, let's take another look at our required permissions. We'll keep them organized by recruiter, hiring manager, and standard employee, because it turns out that (surprise!) field-level security settings are closely related to profiles:
Table 21: Revised Summary of Required Permissions

<table>
<thead>
<tr>
<th></th>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recruiter</strong></td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
</tr>
<tr>
<td><strong>Hiring Manager</strong></td>
<td>• Read</td>
<td>• Read*</td>
<td>• Read*</td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• Create</td>
<td>• Edit*</td>
<td>• Create</td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td>• Edit*</td>
<td></td>
<td>• Edit</td>
<td>• Edit</td>
</tr>
<tr>
<td><strong>Standard Employee</strong></td>
<td>• Read (No min/max pay)</td>
<td>• Read* (No SSN)</td>
<td>• Read*</td>
<td>• Read**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Edit**</td>
</tr>
</tbody>
</table>

* Only for those records that are associated with a position to which the hiring manager/standard employee has been assigned

** Only for those records that the standard employee owns

For field-level security settings, we need to zero in only on those rules that include field restrictions in parentheses, specifically:

- On Positions, hide minimum and maximum pay from standard employees
- On Candidates, hide social security numbers from hiring managers and standard employees

So let’s get down to it!

**Accessing Field-Level Security Settings**

So where do you access field-level security settings? It turns out that there are actually two ways to get to those settings—either through a profile’s detail page, or by choosing Setup ➤ Security Controls ➤ Field Accessibility.

The path that you choose depends on whether you’re focused solely on field-level security, or whether you want to fiddle with page layout settings at the same time. The first way is the simplest and requires the least number of clicks, but the second is sometimes more convenient when you’re building a new app, because you can perform two related tasks in the same page.

Because we have two different field rules that we need to implement for our Recruiting app, we’ll define one rule with the first method, and the other with the second.

**Try It Out: Restricting Access to a Position’s Minimum and Maximum Salary Fields**

Let’s get started actually implementing the first of our field-level security rules: *On Positions, hide minimum and maximum pay from standard employees*. We’ll define this rule by accessing field-level security settings through the Standard Employee profile’s detail page.
1. Click **Setup ➤ Manage Users ➤ Profiles**, and then select the Standard Employee profile.

![Figure 48: Standard Employee Profile Detail Page](image)

The first thing you'll notice about the Standard Employee profile's detail page is that it includes several more areas than the edit page that we originally used to define the profile. These additional areas include Page Layouts (which we learned about in Chapter 5: Enhancing the Simple App with Advanced Fields and Page Layouts on page 49), Field-Level Security, Record Type Settings, Login Hours, and Login IP Ranges. Although we won't go into detail here about how to use any areas other than Field-Level Security, they're part of what makes a profile so powerful in our application. You can learn more about them in the online help.

2. In the Field-Level Security area, click **View** next to the Position object.

3. Click **Edit**.
Here we see security settings for all of the fields on Position, including Min Pay and Max Pay, the two fields that we want to restrict. You’ll notice that some field-level security settings on some fields cannot be modified—this is because either they are system-generated fields or they act as lookup relationship fields (foreign keys) to other records.

Since the security settings checkboxes can be a little bit confusing, let’s do a quick exercise to map their values (Visible and Read-Only) to the three logical permission settings for a field: "Hidden," "Read Only," and "Editable".

Table 22: Field-Level Permission Mappings

<table>
<thead>
<tr>
<th>Permission</th>
<th>Visible</th>
<th>Read-Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Only</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Editable</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

After doing this exercise, it’s easy to see that most fields are editable, because their Visible checkbox is the only one selected. To restrict a field from ever being viewed by a user, all we have to do is deselect both checkboxes.

4. Next to the Max Pay field, deselect Visible.
5. Next to the Min Pay field, deselect Visible.
6. Click Save.

We’re done!

Try It Out: Restricting Access to a Candidate’s Social Security Number

For our second field-level security setting (On Candidates, hide social security numbers from hiring managers and standard employees), let’s implement it using the Field Accessibility page. You’ll see that this method is very similar to the last, but that the Field Accessibility page gives us more power and control over how we define our settings.
1. Click Setup ➤ Security Controls ➤ Field Accessibility, and then select the Candidate object.

Notice that with the Field Accessibility tool, we start off by choosing the relevant object, rather than a user profile. Then, we can choose to set field-accessibility either by selecting a single field and then seeing its security settings for every profile (View by Fields), or by selecting a profile and then seeing security settings for every field (View by Profile).

2. Click View by Fields, and then select SSN from the Field drop-down list.

Figure 50: Field Accessibility Page

The table that appears shows us the accessibility settings for social security number for each profile. If you move your mouse over the value for Field Access, hover text indicates whether the security setting is the result of a page layout setting or a field-level security setting.

Let's first set our field-level security for hiring managers.

3. Next to the Hiring Manager profile, click Editable.

Figure 51: Access Settings Page
By clicking the value in a profile's Field Access column, we finally get to see the real reason why the Field Accessibility page is so convenient: we can view and edit a field's field-level security and page layout settings in the same page.

The Field-Level Security area includes the same checkboxes that we discussed previously—to hide the social security number from hiring managers, we can simply deselect the Visible checkbox as we did with the Min Pay and Max Pay fields on Positions.

Before we do that, however, take a look at the Page Layout area. Here you can either change the visibility and editability of the field on the page layout, or you can choose a different page layout for the Hiring Manager profile altogether. Since we are focused on the security of the SSN field in this exercise we don't need to touch any of the Page Layout area settings. However, you should remember that this tool is here for the future, especially if you are ever trying to figure out why a field is or is not visible to a particular user.

4. In the Field-Level Security area, deselect Visible.
5. Click Save.

To finish up, all we need to do is repeat the process for the Standard Employee profile.

6. Next to the Standard Employee profile, click Editable.
7. In the Field-Level Security area, deselect Visible.
8. Click Save.

All done! We've just finished the second piece of our security and sharing puzzle by defining field-level security for the sensitive fields in our Recruiting app. Now, for the final (and most complicated) piece of the puzzle, we need to specify the individual records to which each user needs access. We need to protect our data without compromising any employee's ability to perform his or her job.

### Controlling Access to Records

By setting object- and field-level access rules for each of our three user profiles, we have effectively defined all of the objects and fields that any one of our Recruiting app users can access. In this section, we'll focus on setting permissions for the actual records themselves—should our users have open access to every record, or just a subset? If it's a subset, what rules should determine whether the user can access them? We'll use a variety of AppExchange platform security and sharing tools to address these questions and make sure we get it right.

### Introducing Organization-Wide Defaults

When dealing with record-level access settings, the first thing we need to do is to determine the organization-wide defaults (commonly called "org-wide defaults") for each object in our Recruiting app. Also called a sharing model, org-wide defaults specify the baseline level of access that the most restricted user should have. We'll use org-wide defaults to lock down our data to this most restrictive level, and then we'll use our other record-level security and sharing tools (role hierarchies, sharing rules, and manual sharing) to open up the data to other users who need to access it.

### Org-Wide Defaults in Our Recruiting App

To determine the org-wide defaults that we'll need in our Recruiting app, we need to answer the following questions for each object:
1. Who is the most restricted user of this object?
2. Is there ever going to be an instance of this object that this user shouldn't be allowed to see?
3. Is there ever going to be an instance of this object that this user shouldn't be allowed to edit?

Based on our answers to these questions, we can determine the sharing model that we need for that object as illustrated in the following diagram.

![Diagram showing the process of determining the sharing model for an object](image)

**Figure 52: Determining the Sharing Model for an Object**

For example, let’s consider the Position object in our recruiting app. To refresh our memories, here’s our table of required permissions:
Table 23: Revised Summary of Required Permissions

<table>
<thead>
<tr>
<th></th>
<th>Position</th>
<th>Candidate</th>
<th>Job Application</th>
<th>Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recruiter</td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
<td>• Edit</td>
</tr>
<tr>
<td>Hiring Manager</td>
<td>• Read* (No SSN)</td>
<td>• Read*</td>
<td>• Read*</td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• Create</td>
<td>• Edit*</td>
<td>• Edit</td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td>• Edit*</td>
<td></td>
<td></td>
<td>• Edit</td>
</tr>
<tr>
<td>Standard Employee</td>
<td>• Read (No min/max pay)</td>
<td>• Read* (No SSN)</td>
<td>• Read* (No SSN)</td>
<td>• Read**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Edit**</td>
</tr>
</tbody>
</table>

* Only for those records that are associated with a position to which the hiring manager/standard employee has been assigned

** Only for those records that the standard employee owns

Now let's go through and answer our list of questions for Position:

Table 24: Determining Org-Wide Defaults for the Standard Employee Profile

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Who is the most restricted user of this object?</td>
<td>A member of the Standard Employee profile. All that they're allowed to do is view a position.</td>
</tr>
<tr>
<td>2. Is there ever going to be an instance of this object that this user shouldn't be allowed to see?</td>
<td>No. Although the values for the minimum and maximum pay are hidden from standard employees, they're still allowed to view all Position records.</td>
</tr>
<tr>
<td>3. Is there ever going to be an instance of this object that this user shouldn't be allowed to edit?</td>
<td>Yes. Standard employees aren't allowed to edit any Position record.</td>
</tr>
</tbody>
</table>

According to our flowchart, answering "Yes" to question #3 means that the sharing model for Position should be set to Public Read-Only.

Going through the rest of our recruiting objects required permissions, we can easily figure out their sharing models, too. The Standard Employee profile is the most restricted user for each object, and there are going to be Candidate, Job Application, and Review records that particular employees won't be able to view. Consequently, the sharing model for Candidates, Job Applications, and Reviews should all be set to Private.
Try It Out: Setting Org-Wide Defaults

Now that we've figured out the org-wide defaults for each of our recruiting objects, let's go ahead and implement them in our Recruiting app.

1. Click Setup ➤ Security Controls ➤ Sharing Settings. If you see an introductory splash page, click Set Up Sharing at the bottom of the page to skip to the actual tool.

The Sharing Settings page is where we control both org-wide defaults and sharing rules. We'll talk more about this page when we talk about sharing rules a little further down. For now, let's just edit our org-wide default settings.

2. In the Organization Wide Defaults area, click Edit.

This page controls the org-wide defaults for every object in our organization. You'll notice that some standard objects (like leads and calendars) use a different set of org-wide default values than we have available for our custom recruiting objects. You can learn more about them in the online help. For now, let's just set our recruiting objects to the org-wide defaults that we decided on in the last section.

3. Next to Position, select Public Read Only.
4. Next to Candidate, Job Application, and Review, select Private.
5. Click Save.

Easy! Now that we've locked down our data with org-wide defaults, users are currently allowed to work on only Candidate, Job Application, and Review records that they own and allowed to view Position records that anyone owns. Because those settings are way too restrictive for any user to get any benefit out of our app, now we need to use role hierarchies, sharing rules, and manual sharing to open up Candidate, Job Application, and Review record access to those employees who'll need it.

Introducing Role Hierarchies

The first way that we can share access to records is by defining a role hierarchy. Similar to an org chart, a role hierarchy represents a level of data access that a user or group of users needs. Users assigned to roles near the top of the hierarchy (normally the CEO, executives, and other management) get to access the data of all the users who fall directly below them in the hierarchy. The role hierarchy ensures that a manager will always have access to the same data as his or her employees, regardless of the org-wide default settings. Role hierarchies also helpfully
define groups of users who tend to need access to the same types of records—we'll use these groups later when we talk about sharing rules.

To illustrate, let’s take a look at a portion of the role hierarchy for Universal Containers:

![Universal Containers Role Hierarchy](image)

Role hierarchies don’t necessarily need to match your org chart exactly. Instead, each role in the hierarchy should just represent a level of data access that a user or group of users needs. For example, suppose your organization employs a corporate lawyer who needs to access all of the records in the AppExchange platform. One easy way to accomplish this is by assigning the lawyer to the CEO role in your organization’s role hierarchy. Since the CEO role is placed at the top of the hierarchy, anyone assigned to that role automatically gets full access to any record in the organization. It doesn’t matter that technically the lawyer appears below the CEO in the regular org chart.

**Comparing Roles and Profiles**

Although it’s easy to confuse profiles with roles, they actually control two very different things.

As we learned earlier in this chapter, profiles control a user’s object- and field-level access permissions. Indeed, a user can’t be defined without being assigned to a particular profile, since the profiles specifies the apps and tabs that appear when he or she logs in, among a number of other useful things.

Roles, on the other hand, primarily control a user’s record-level access permissions through role hierarchy and sharing rules. Although a role assignment isn’t exactly required when we define a user, it would be foolish of us not to assign a role since it makes it so much easier to define our record-level permissions. Indeed, trying to define record-level permissions without assigning a role to a user would be a lot like trying to travel from New York to San Francisco by car when there’s an airplane available—there’s just a much more efficient way of doing it!

Because profiles control object- and field-level access whereas roles influence record-level access, a user is typically assigned to one of each. To help you remember which controls what, remember: **Roles control Records**.
Role Hierarchies in Our Recruiting App

Given the Universal Containers role hierarchy that’s pictured in Figure 54: Universal Containers Role Hierarchy above, let’s think about how implementing this hierarchy will open up certain kinds of record-level permissions to various users of our Recruiting app. Remember, since defining our org-wide defaults, our users are currently allowed to only view all Position records and to view and update other recruiting records that they own. That doesn’t make our app all that useful. However, once we implement our role hierarchy, we’ll automatically grant several kinds of record-level permissions to various users. For example:

- The CEO, Cynthia Capobianco, will be able to view and update every record that anyone else in the organization can view and update.
- The VP of Development, Andrew Goldberg, will be able to view and update any record that his managers or his managers’ employees can view or update.
- The VP of Human Resources, Megan Smith, will be able to view and update any record that Phil Katz, her recruiting manager or Mario Ruiz, Phil’s recruiter, can view and update.
- The Recruiting Manager, Phil Katz, will be able to view and update any record that is owned by Mario Ruiz, his recruiter.
- The Software Development manager, Ben Stuart, will be able to view and update any record that is owned by Melissa Lee, Tom Zales, or Craig Kingman, his software engineers.
- The director of QA, Clark Kentman, will be able to view and update any record that is owned by Flash Stevenson or Harry Potterham, his QA Engineers.
- The director of Product Management, Frank Linstrom, will be able to view and update any record that is owned by Amy Lojack or Andy Macrola, his product managers.

As we can see, the role hierarchy is very powerful in opening up data for people high up in the role hierarchy tree! However, let’s look at some of the gaps that we still have in our record-level permissions:

- Megan Smith (and her whole recruiting team) won’t be able to view any reviews that are owned by members of Andrew Goldberg’s Development team, because she doesn’t have a direct line down to any Development roles in the role hierarchy.
- Ben Stuart, the software development manager, also won’t be able to see any reviews that were written by members of the QA or Product Management groups, even if QA engineers or product managers interviewed candidates for a software engineering position in his group.
- Melissa Lee, a software engineer, won’t be able to see the records for candidates that she’s supposed to interview.

Clearly we’ll need to use other record-level sharing methods to open up data between peers in the same group, and also between groups that appear in different branches of the role hierarchy (we’ll get to those later in this chapter). However, the role hierarchy does give us a good start toward opening up record access, so let’s take a look now at how to define it.

Try It Out: Defining a Role Hierarchy

Implementing a role hierarchy in the AppExchange platform is easy once you have an idea of what the hierarchy should look like. It’s best to just start with your company’s org chart and then consolidate different job titles into single roles wherever possible. For example, if Ben Stuart’s software development group has a staff software engineer and a junior software engineer, these positions can be consolidated into a single Software Engineer role in the hierarchy.
Once that's all squared away, we can get started actually defining the role hierarchy itself. For our exercise, we'll go ahead and use the role hierarchy that we talked about in the previous sections.

1. Click **Setup ➤ Manage Users ➤ Roles**. If you see an introductory splash page called Understanding Roles, click **Set Up Roles** at the bottom of the page to skip to the actual tool.

![Use this dropdown list to change your role hierarchy display mode.](image)

**Figure 55: Empty Role Hierarchy Page in Tree View Mode**

The default view for this page is the tree view, as indicated in the drop-down list on the far right side of the Role Hierarchy title bar. When creating a role hierarchy, it’s probably easiest to stick with this or the list view, because they both make it easy to see how the roles all fit together in the hierarchy. The sorted list view is best if you know the name of a role that you want to find but aren’t sure where it fits in the hierarchy (or are too lazy to click open all the tree nodes or scroll down a long list). For our purposes, we’ll stick with the tree view for now.

When you first start defining a role hierarchy, the tree view displays a single placeholder node with the name of your organization. From this point, we need to add the name of the role that is highest up in the hierarchy—in our case, the CEO.

**Note:** If you’re building your Recruiting app with a free Developer Edition organization, you may have a role hierarchy predefined as a sample. That’s alright. You can still follow along and create some more roles.

2. Just under Universal Containers, click **Add Role**.
3. In the **Role Name** text box, enter CEO.
4. In the **This role reports to** text box, click the lookup icon and click **Select** next to Universal Containers.

By choosing the name of the organization in the **This role reports to** text box, we’re indicating that the CEO role is a top-level position in our role hierarchy and doesn’t report to anyone.

5. In the **Role Name as displayed on reports** text box, enter CEO. This text is used in reports to indicate the name of a role. Since you may not want a long role name, like VP of Product Development, taking up too much space in your report columns, it’s advisable to use a shortened, yet easily identifiable, abbreviation.
6. Leave any other options, such as **Opportunity Access**, set to their defaults. These access options don’t have anything to do with our Recruiting app, and only appear if you have the org-wide defaults for a standard object set to a level more restrictive than Public Read/Write.
7. Click **Save**.
Now that we've created our first role, we can assign the appropriate user to it.

8. In the CEO role detail page, click **Assign Users to Role**.
9. In the **Available Users** drop-down list, select All Unassigned.
10. Choose Cynthia Capobianco from the list, and click **Add** to move her to the **Selected Users for CEO** list.
11. Click **Save**.

If we return to the main Roles page by clicking **Setup ➤ Manage Users ➤ Roles**, we can now see our new CEO role in the hierarchy. Defining the rest of the roles is just an exercise that you can do on your own according to **Figure 54: Universal Containers Role Hierarchy** on page 103. (If you don’t define the role hierarchy, some of the tests that we talk about later won’t work as described.)

**Tip:** To speed up the process of adding a new role, click **Add Role** directly under the name of the role to which the new role should report. When you do this, the **This role reports to** text box is automatically filled in with the name of the appropriate role.

Not too hard, right? With org-wide defaults and a role hierarchy in place, we're actually pretty close to finishing up our record-level access permissions. All we have left to do is share recruiting-related records between groups that appear in separate branches of the role hierarchy, and between peers in a single group. Fortunately, we can accomplish both of those tasks with a combination of sharing rules and manual sharing. We just need to figure out what’s left that needs to be shared, and to whom.

**What’s Left to be Shared?**

So what is left to be shared? After reviewing our table of required permissions, it turns out it’s just a few more things (remember, since users always have access to the records that they own, we need to worry only about the read and update permissions for our record-level access settings):

- Recruiters need read and update access on every Position, Candidate, Job Application, and Review record that exists in the app.
- Hiring managers need:
  - Read and update access on Position and Job Application records on which they’re the hiring manager
  - Read access on Candidate records for whom they're the hiring manager
  - Read and update access on every Review record
Interviewers need read access on the Candidate and Job Application records for people they’re interviewing. That shouldn't be too hard! Let’s go do it.

**Introducing Sharing Rules**

First let’s see what we can do with sharing rules. Sharing rules let us make automatic exceptions to org-wide defaults for particular groups of users. We’ve already defined several specific groups with the roles that we created in the previous section, but we can also make up other groups as needed.

The thing to remember with sharing rules is that, like role hierarchies, we can use them only to open up record access to more users. Sharing rules and role hierarchies can never be stricter than our org-wide default settings.

**Sharing Rules in Our Recruiting App**

Sharing rules work best when they’re defined for a particular group of users that we can determine or predict in advance, rather than a set of users that is frequently changing. For example, in our Recruiting app, we need to share every position, candidate, job application, and review with every recruiter. Since recruiters all belong to either the Recruiting Manager or Recruiter roles in the role hierarchy, we can easily use a sharing rule to share those objects with the Recruiting Manager role and its subordinates.

Alternatively, consider another use case from our Recruiting app: interviewers need read access on the candidates and job applications for people they're interviewing. In this case, the set of interviewers is a lot harder to predict in advance—hiring managers might use different sets of interviewers depending on the position for which they're hiring, and the interviewers might come from different groups in the role hierarchy. As a result, this use case probably shouldn’t be handled with sharing rules—the team of interviewers for any given manager is just too hard to predict.

Let’s go through the set of required permissions we still need to implement and pick out the ones that would work best with sharing rules:

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Should we use a sharing rule?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recruiters need read and update access on every Position, Candidate, Job Application, and Review record that exists in the app.</td>
<td>Yes. As we discussed above, it’s easy to pick out the group of recruiters in our role hierarchy.</td>
</tr>
<tr>
<td>Hiring managers need read and update access on Position and Job Application records on which they're the hiring manager.</td>
<td>No. It's too hard to predict which positions will be assigned to which hiring manager. We'll need to handle this use case some other way.</td>
</tr>
<tr>
<td>Hiring managers need read access on Candidate records on which they're the hiring manager.</td>
<td>No. Again, it’s too hard to predict which positions will be assigned to which hiring manager.</td>
</tr>
<tr>
<td>Hiring managers need read and update access on every Review record.</td>
<td>Yes. Since we’re not restricting which reviews a hiring manager gets to read and update, we can easily pick out all of the hiring managers from our role hierarchy and define a sharing rule for them.</td>
</tr>
<tr>
<td>Interviewers need read access on the Candidate and Job Application records for people they're interviewing.</td>
<td>No. As we discussed above, it’s hard to predict who will be a member of an interview team for a particular position.</td>
</tr>
</tbody>
</table>
Great! Now that we know the required permissions we want to implement with sharing rules, let's go ahead and define them.

**Try It Out: Defining a Public Group for Reviewers**

Before we dive head first into creating our sharing rules, we need to make sure that we have the appropriate public groups set up. A *public group* is a collection of individual users, other groups, individual roles, and/or roles with their subordinates. Using a public group when defining a sharing rule makes the rule easier to create and, more important, easier to understand later, especially if it's one of many sharing rules that you're trying to maintain in a large organization. You'll need to create a public group if you ever want to define a sharing rule that encompasses more than one or two groups or roles, or any individual.

Looking at the required permissions that we want to implement, there's just one object that needs a public group for its sharing rule: Reviews. Since both recruiters and hiring managers need read and update access, let's go ahead and make a public group called Reviewers that encompasses their roles in the role hierarchy.

1. Click Setup ➤ Manage Users ➤ Public Groups.
2. Click New.

![Figure 57: New Public Group Page](image)

The New Public Group page allows you to choose other public groups, individual roles, individual roles including the roles' subordinates, or individual users.

3. In the Group Name text box, enter Reviewers.
4. In the Search drop-down list, choose Roles.
5. In the Available Members list, select Role: SW Development Manager, and click Add.
6. In the Search drop-down list, choose Role and Subordinates.
7. In the Available Members list, select Role and Subordinates: Recruiting Manager, and click Add.
8. Click Save.

Easy! Now we're ready to define our sharing rules.

**Try It Out: Defining a Sharing Rule for Review Records**

Since we just defined our Reviewers public group, let's go use it to define our sharing rule for Review records.

1. Click Setup ➤ Security Controls ➤ Sharing Settings.
Remember this page? We were last here when we defined our org-wide defaults.

2. In the Manage sharing settings for drop-down list, choose Review.

Choosing an object in this drop-down list allows us to focus in on the org-wide defaults and sharing rules for a single object at a time rather than looking at all of them in a long page—a really useful thing if you've got a large organization with several custom objects.

3. In the Sharing Settings area, click New.
4. In the Review: owned by member of drop-down list, select Public Groups.
5. Next to that drop-down list, choose Entire Organization.

Just as we talked about above, you can define a sharing rule only for a single public group, role, or role with all of its subordinates. By default, the AppExchange platform includes a default public group that encompasses every user in your organization.

6. In the Share with drop-down list, select Public Groups.
7. Next to that drop-down list, choose Reviewers.
8. In the Access Level drop-down list, select Read/Write.
9. Click Save.
10. Click OK in the dialog box that says this operation could take significant time.

And that's it! We've just created a rule that shares reviews written and owned by any member of the organization with all recruiters and hiring managers. Since reviewers and hiring managers all need the power to read and update reviews, we handled everyone with a single sharing rule and a public group.

To finish up here, go ahead and create two more sharing rules according to the following table:

Table 26: Additional Sharing Rules

<table>
<thead>
<tr>
<th>These records...</th>
<th>Owned by...</th>
<th>Should be shared with...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidates</td>
<td>The role and subordinates of the Recruiting Manager</td>
<td>The role and subordinates of the Recruiting Manager</td>
</tr>
<tr>
<td>Job Applications</td>
<td>The role and subordinates of the Recruiting Manager</td>
<td>The role and subordinates of the Recruiting Manager</td>
</tr>
</tbody>
</table>

Introducing Manual Sharing

Now let's talk about what we have left to do to define our sharing model. After implementing our sharing rules above, the following required permissions remain:

- Hiring managers need read and update access on Position and Job application records on which they're the hiring manager.
- Hiring managers need read access on Candidate records on which they're the hiring manager.
- Interviewers need read access on the Candidate and Job Application records for people they're interviewing.

We didn't implement those required permissions with sharing rules because it was too hard for us to come up with a consistent group of users who would need access to a particular set of records. Really, this is where the job of the recruiter comes into play. A recruiter like Mario Ruiz owns the Position, Candidate, and Job Application...
records for jobs that he’s trying to fill, and he also knows the hiring manager and interviewers who should be assigned to them.

Fortunately, we have one final type of record-access setting that allows Mario to share specific records with other specific users: manual sharing. With manual sharing, Mario can grant read or read/write access on records that he owns to any other user, role, or public group. Although it isn’t automated like org-wide defaults, role hierarchies, or sharing rules, manual sharing gives Mario the flexibility to share particular records with the ever-changing groups of interviewers and hiring managers with whom he has to deal every day.

**Try It Out: Defining a Manual Sharing Rule**

Let’s pretend that we’re a recruiter like Mario and we need to share a Candidate record that we own with another role, group, or user:

1. On the detail page for the candidate, click **Sharing**.

![Figure 58: Sharing Detail Page](image)

Since we own this Candidate record, we get to see details about who else can see the record and why. If we didn’t own this record, there would be a message about not having sufficient privileges.

**Tip:** If we wanted to view the names of each user who has access to the record rather than just the names of the roles and public groups, we could click **Expand List** in this page. Although the operation can take some time depending on the number of users in our organization, it’s helpful to determine whether we need to define a manual sharing rule for a particular user or if he or she already has access.

2. Click **Add**.
3. In the **Search** drop-down list, choose whether we want to manually share the record with a user, public group, role, or role and subordinates.
4. In the **Currently Not Shared** list, select the user, public group, or role that should have access to the record, and click **Add**.
5. In the **Access Level** drop-down list, specify whether the user, public group, or role should have read or read/write access to the record.
6. Click **Save**.

Not too hard! When we roll out our Recruiting app to users, we’ll have to train our recruiters to take these steps for the Position, Candidate, and Job Application records that their hiring managers and interviewers need to access. Once this training is complete, we will have implemented all of the required sharing and security settings that we discussed at the beginning of the chapter—well done!
Putting It All Together

Congratulations! We've just implemented all of our required security and sharing settings, first by defining object-level access with profiles, then by securing field-level access with field-level security, and finally by defining record-level access using org-wide defaults, role hierarchies, sharing rules, and manual sharing.

We learned about the difference between object-, field-, and record-level security, and how profiles and roles work together to first determine the objects and tabs that a user can possibly use, and then the specific records that the user can actually access and edit.

Let's now try it out for ourselves. To do so, we'll first have to define a number of users, and then we can play around with creating records and seeing who has access to what.

Try It Out: Creating Users for Our Recruiting App

To really put our Recruiting app through its paces, we'll first need to define the following users and assign a couple of them to some of the recruiting records that we imported earlier.

Table 27: Recruiting App Users

<table>
<thead>
<tr>
<th>User's First and Last Name</th>
<th>Title</th>
<th>Profile</th>
<th>Role</th>
<th>Owner of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil Katz</td>
<td>Recruiting Manager</td>
<td>Recruiter Profile</td>
<td>Recruiting Manager</td>
<td>• Product Manager position&lt;br&gt;• Candidate for Product Manager&lt;br&gt;• Job Application linking the Product Manager position with the candidate</td>
</tr>
<tr>
<td>Mario Ruiz</td>
<td>Senior Recruiter</td>
<td>Recruiter Profile</td>
<td>Recruiter</td>
<td>• Jr Software Engineer position&lt;br&gt;• Candidate for Jr Software Engineer&lt;br&gt;• Job Application linking the Jr Software Engineer position with the candidate</td>
</tr>
<tr>
<td>Ben Stuart</td>
<td>Software Development Manager</td>
<td>Hiring Manager Profile</td>
<td>SW Development Mgr</td>
<td></td>
</tr>
<tr>
<td>Melissa Lee</td>
<td>Staff Software Engineer</td>
<td>Standard Employee Profile</td>
<td>SW Engineer</td>
<td></td>
</tr>
</tbody>
</table>
Note: If you're implementing the Recruiting app in a Developer Edition organization, you'll have only one additional user to play with besides the System Administrator user. You can still try out all of the use cases that we describe below, but you'll have to update the user's profile and role for whatever use case you're working on.

Let's walk through the creation of Phil Katz. Then you can finish the other four users on your own:

1. Click Setup ➤ Manage Users ➤ Users.
2. Click New User.
3. Fill out the fields in the User Edit page according to above.

Because we're defining this user to test the security and sharing settings of our app, enter a real email address that you have access to in the Email field and then a "fake" email address in the Username field (for example, phil.katz@recruiting.com). We'll use the "fake" value in the Username field to log in to the app as Phil Katz, but we'll get Phil's automatically generated password at the real email account that you specified. Without that password, we'd never be able to log in! (For a real user, both Email and Username should generally have the same value.)

4. Click Save.

Now that we've created the Phil Katz user, let's give him ownership of the Product Manager position and its associated Job Application and Candidate records.

5. Click the Positions tab.
6. Click Product Manager.
7. Next to the **Owner** field, click **Change**.
8. Click the lookup icon [ ] and choose Phil Katz.
9. Click **Save**.
10. In the Job Applications related list, click the name of the listed Job Application and repeat Steps 3-5.
11. Click the name of the associated Candidate on the Job Application detail page and repeat Steps 3-5.

All done! Now create the other four users in **Recruiting App Users** on page 111 and assign them ownership of the indicated records.

**Try It Out: Verifying that Everything Works**

Now that we've got data assigned to actual users, let's go through our Recruiting app and see how the security and sharing permissions that we defined in this chapter play out:

1. First log in as Mario Ruiz—verify that he can see and edit both positions, all three candidates, all three job applications, and Amy Lojack's review. Verify that the **New** buttons are there for all recruiting objects.
2. Log in as Melissa Lee—verify that she can view positions but there's no **New** button. Verify that she can't see any candidates, reviews, or job applications.
3. Log in as Ben Stuart—verify that he can view positions and there's a **New** button. Verify that he can't see any candidates or job applications. Verify that he can view reviews and there's a **New** button. (What do reviews look like? Can he see the names of the candidates and job applications on them?)
4. Log in again as Mario Ruiz—have him manually share read/write access on the Jr. Software Engineer position with Ben. Have him manually share read access on the candidate with Melissa and Ben. Have him manually share read access on the job application with Melissa and read/write access on the job position with Ben.
5. Log in again as Melissa Lee—verify that she can now see the candidate and job application that Mario just shared with her but that she can't see the candidate's social security number. Have her create a review for that candidate.
6. Log in again as Ben Stuart—verify that he can edit the Jr Software Engineer position. Verify that he can read and update Melissa's review. Verify that he can update the job application to suggest that they hire the candidate.

How did we do? If all of these use cases worked correctly, you've just successfully set up security and sharing for our Recruiting app! Now let's go incorporate some business logic into our app with workflow rules.
Chapter 8
Using Custom Workflow

In this chapter ...

• Introducing Workflow
• Workflow in Our Recruiting App
• Try It Out: Creating Workflow Rules That Assign Tasks
• Try It Out: Creating a Workflow Rule That Updates Fields
• Try It Out: Creating a Workflow Rule That Sends Email Alerts
• Summing Up

Now that we've set up all our custom objects and made sure they're secure, it's time to start looking at ways we can make our Recruiting app more functional.

Up to this point, we've created little more than a glorified database—it holds all of the information we need and allows us to search for records based on various criteria, but it doesn't help our recruiters and hiring managers perform the functions of their jobs more effectively. There's no automation that lets a recruiter know when the status of a candidate has changed or when a new position has been entered into the system.

Indeed, when any changes are made, users have to remember to notify one another of the change or else rely on the fact that others will find the updates on their own. Neither solution is practical for the long term and invites the possibility that the Recruiting app won't be adopted consistently by all the employees at Universal Containers. How can we build processes into our app so that users won't need to rely on manual methods of communication to inform others of changes?
Introducing Workflow

Once again, the AppExchange platform helps us out with more built-in functionality. In this case, the tool that we'll use to solve our process automation problem is called workflow.

Workflow is an AppExchange platform business-logic engine that allows us to automatically send email alerts, assign tasks, or update field values based on rules that we define. Any time that changes to a record meet the conditions in a workflow rule, the platform automatically performs any actions associated with the rule.

For example, let's say Ben Stuart, a software development manager, has decided to extend an offer to Ethan Tran, a bright young candidate who's interested in the Jr. Software Engineer position. At Universal Containers, it's a recruiter's job to extend offers, but in this case, Mario Ruiz, the recruiter responsible for the Jr. Software Engineer position, doesn't know whether or not Ben has made a decision unless Ben emails or calls him directly.

Instead of relying on Ben to remember to tell Mario, we can set up a simple workflow that triggers the assignment of the appropriate task whenever the status on a Job Application record is set to Extend an Offer or Reject. As soon as Ben changes the status of the candidate's Job Application, workflow creates the appropriate task and sends Mario a notification email, as shown in the diagram below.

Pretty powerful, isn't it? In general, if we can come up with a standard rule that specifies when a particular event should happen, we can make it happen automatically with workflow. Workflow is the secret ingredient that's going to transform our Recruiting app from a glorified database into a fully functional tool that everyone finds useful.

Now that we've got a general idea of what workflow's all about, let's take a closer look at the individual components that make up workflow: rules, tasks, field updates, and alerts.
Introducing Workflow Rules

In general, a workflow rule is the main container for a set of workflow instructions. It includes the criteria for when the workflow should be activated, as well as the particular tasks, alerts, and field updates that should take place when the criteria for that rule are met.

Every workflow rule must be based on a single object you choose when you define the rule. This object influences the fields that are available for setting workflow activation criteria.

For example, if we define a workflow rule for the Job Application object, we'll be able to set workflow activation criteria based on the values of fields like Job Application Number and Status. We can also set workflow activation criteria based on standard fields, like Record Owner or Created Date, as well as fields based on the currently active user when a rule is evaluated, such as their Role or Time Zone.

We'll look at all the ways that we can set workflow activation criteria when we get to building our own workflow rules a little later. For now, just understand that the AppExchange platform makes it very easy to create detailed workflow rules that target specific situations.

Introducing Workflow Actions: Tasks, Field Updates, and Alerts

When a workflow rule is triggered, there are three types of actions that can occur:

Workflow Tasks

A workflow task assigns a task to a user according to a particular template. Just as in Microsoft Outlook, tasks include information about something that needs to be done by a certain time, such as making a telephone call or returning a library book. Assigned tasks appear in a user's My Tasks related list on their Home tab and generate reminder messages that pop up when a user logs in.

When we define a workflow task, we provide default values for the Assignee, Subject, Status, Priority, and Due Date fields for tasks that are generated by its associated workflow rule. We can also make sure that a notification email is sent to the assignee when a task is automatically generated.

Workflow Field Updates

A workflow field update changes the value of a particular field on the record that initially triggered the workflow rule.

Workflow Alerts

A workflow alert sends an email according to a specified email template. Unlike workflow tasks, which can only be assigned to users of the app, workflow alerts can be sent to any user or contact, as long as they have a valid email address.

A workflow rule can include any combination of these actions when the rule is triggered. For example, one rule might send out an alert and update two fields on a particular record. The action that one workflow rule takes can also trigger the execution of another workflow rule.

Workflow in Our Recruiting App

Now that we've oriented ourselves to the different components involved with workflow, let's take a look at our Recruiting app and see how we can use workflow to help us build out the requirements that we talked about in
Chapter 2: About the Sample Recruiting App on page 19. Then we'll spend the rest of this chapter implementing the rules that we come up with here.

We've already talked about one instance where workflow will give us a big advantage: automatically assigning a task to a recruiter when the status of a Job Application record changes to Reject or Extend an Offer. This will be a great start, but what else can we do?

If we look back at our last chapter on security and sharing, recall that we wanted to grant both recruiters and hiring managers permission to create new positions, but that ultimately we always wanted a recruiter to own those records because filling them is the recruiter's job responsibility. We hinted in the security and sharing chapter that we could accomplish this with workflow, and indeed we can! We'll simply need to use a workflow field update to change the record owner of a Position record if it wasn't created by a recruiter. We can assign them to Phil Katz, our recruiting manager, and he can then reassign them to the appropriate recruiter, if necessary. This is a perfect example of how we can use workflow field updates to add functionality to our app.

For a third way of using workflow, let's think about how position availability is advertised throughout Universal Containers. Like many organizations, Universal Containers prefers to fill positions with employee referrals, but employees often aren’t aware of which positions are currently open. We can use a workflow alert to automatically send email to every employee whenever a new position opens up. This way employees learn about positions as they become available and can immediately think of friends and family members who might be interested.

Sound good? While there are probably many more ways to use workflow to build a killer on-demand Recruiting app, let's stick with these three for now since they'll give us a good example of each of the three types of workflow actions that are available. To summarize what we'll be building:

1. A workflow task that assigns a task to a recruiter when the status of a Job Application changes to Reject or Extend an Offer
2. A workflow field update that reassigns ownership of a Position that's created by a hiring manager to a recruiter
3. An workflow alert that sends an email to every employee when a new Position is created

Now let's get started!

Try It Out: Creating Workflow Rules That Assign Tasks

For our first use of workflow, we want to create two rules: one that assigns a "Send Rejection Letter" task to the relevant recruiter whenever the Status field of a Job Application record is set to Reject, and one that assigns an "Extend Offer" task to the relevant recruiter whenever the Status field of a Job Application record is set to "Extend an Offer." Let's start with the "Send Rejection Letter" workflow rule.

Try It Out: Creating the "Send Rejection Letter" Workflow Rule

To make this rule entirely functional, we'll need to define both a workflow rule, which specifies the criteria for when the rule should be executed, and a workflow task that includes the "Send Rejection Letter" task template. Although we can define workflow components in any order, let's start with defining the workflow rule itself. It'll save us a couple of clicks a little later when we define the rule's associated workflow task.

2. Click New Rule.
The first thing we need to do is select the object that will be associated with our workflow rule. As we talked about earlier, every workflow rule must be associated with a single object to determine the fields that we can use to set criteria. Since we need to trigger this workflow rule when the Status field of a Job Application record is set to "Reject," we need to select Job Application here:

3. In the Select object drop-down list, choose Job Application, and click Next.

Now it's time to set our workflow rule details:

4. In the Rule Name text box, enter Send Rejection Letter.
5. In the Description text box, enter "Send a rejection letter when a hiring manager changes the status of a job application to Reject."

The Active checkbox specifies whether the workflow rule should actually start working as soon as it's saved. If you're working in an environment with other users, you might want to leave this deselected until you've communicated the change to the other people in your organization. Since we want to test and use the rule right away, we'll select it:

6. Select the Active checkbox.

Now specify when this rule should be evaluated. We can choose Only when a record is created (which ignores updates to existing records), When a record is created or edited and did not previously meet the trigger criteria (which repeatedly triggers the rule every time a record is saved only until the rule's criteria are met), or Every time a record is created or edited (which repeatedly triggers the rule every time the record is saved). Since we don't want to assign duplicate tasks to a recruiter every time the record is saved, we'll choose the second option.

7. Under "Evaluate this rule," select When a record is created or edited and did not previously meet the trigger criteria.

To finish up defining the rule, we need to set the criteria that will trigger execution of the rule's associated actions. Every workflow rule requires at least one row of filter criteria, but we can set as many filters as we want using additional rows.

8. In the first row of filters:
   - Set the Field column to Status.
   - Set the Operator column to equals.
   - Set the Value column to Rejected.

9. Click Save.
Well done! The AppExchange platform returns us to the detail page for the workflow rule that we just created. We're now ready to define the workflow task that should be generated whenever this workflow rule is triggered.

**Try It Out: Creating the "Send Rejection Letter" Workflow Task**

Like other detail pages that you might remember from earlier chapters, the detail page for a workflow rule includes definition details for the rule at the top of the page and then a series of related lists. In the workflow rule detail page, these related lists consist of the actions that should take place whenever the rule is triggered: workflow tasks, alerts, and field updates.

For our "Send Rejection Letter" workflow rule, we only need to create a single entry in the Workflow Tasks related list:

1. In the Workflow Tasks related list, click **New Task**.

Did you notice that the **Object** field is already filled in with Job Application? Here's where we saved ourselves a couple of clicks—if we'd started building our workflow task before our workflow rule, we would have needed to specify the object with which the task would be associated. That's because, like workflow rules, workflow tasks, field updates, and alerts all need to be associated with a single object.

In this case, we got to skip a page because the object with which a workflow action is associated must match the object of a workflow rule that uses it. For example, if we had a workflow rule associated with the Candidate object, any task, field update, or alert that's triggered by our Candidate workflow rule must also be associated with the Candidate object. Because we started building our "Send Rejection Letter" workflow task directly from the related list of the "Send Rejection Letter" workflow rule, the platform knew that the object associated with our task had to match the rule that we'd already built. That's why our new workflow task is already associated with the Job Application object.

The rest of the fields on this edit page make up the template for any "Send Rejection Letter" tasks that our workflow rule will generate.
Try It Out: Creating Workflow Rules That Assign Tasks

2. Next to the Assigned To field, click the lookup icon (ackle). Here we can choose the assignee of the task either by specifying a particular user, role, or the owner of the Job Application record that triggered the workflow rule in the first place. Since recruiters always own the Job Application records of the positions that they’re responsible for, and recruiters are responsible for sending rejection letters at Universal Containers, let’s select the record owner:

3. In the Type drop-down list, choose Owner.
4. Click Record Owner.

Caution: If you thought that choosing the Recruiter role for the Assigned To field might have been another valid option, be careful. If the assignee of a workflow task is a role that contains more than one assigned user, the person who triggered the rule will become the task assignee instead. For this reason, you never want to assign workflow tasks to roles unless you’re sure that only one user will ever be assigned to them at a time.

The rest of the workflow task fields are easy:

5. In the Subject text box, enter Send Rejection Letter.
6. In the Due Date drop-down lists, choose Rule Trigger Date plus 2 days.

This Due Date setting will give our recruiters two days to notify the candidate after they're first assigned the task.

7. In the Status drop-down list, choose Not Started.
8. In the Priority drop-down list, choose High.

The Notify Assignee checkbox allows us to send an email to the assignee as soon as the task is created by the workflow rule. This ensures that the assignee knows about the task without having to log in to the application on a regular basis, so let’s select it.

9. Select the Notify Assignee checkbox.
10. Click Save.
All done! We've just created our first workflow rule and task. You'll find that the remaining workflow actions all operate in a very similar way, so we'll speed through the rest of them, focusing just on the fields and options that are unique to each. First, let's finish up our second workflow rule that assigns a task by quickly creating the "Extend an Offer" workflow rule.

Try It Out: Creating the "Extend an Offer" Workflow Rule and Task

To wrap up our first use case, we need to create another workflow rule for when the status of a Job Application is set to Extend an Offer. This rule and task are almost identical to the "Send Rejection Letter" workflow rule and task, so we'll just list the values that you'll need in the following two tables.

Table 28: Values for Creating the "Extend an Offer" Workflow Rule

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Job Application</td>
</tr>
<tr>
<td>Name</td>
<td>Extend an Offer</td>
</tr>
<tr>
<td>Description</td>
<td>Make an offer when a hiring manager changes the status of a Job Application to Extend Offer</td>
</tr>
<tr>
<td>Active?</td>
<td>Yes</td>
</tr>
<tr>
<td>Evaluate this rule...</td>
<td>When a record is created or edited and did not previously meet the trigger criteria</td>
</tr>
<tr>
<td>Filter Criteria</td>
<td>Status equals Extend an Offer</td>
</tr>
</tbody>
</table>
Table 29: Values for Creating the "Extend an Offer" Workflow Task

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Job Application</td>
</tr>
<tr>
<td>Assigned To</td>
<td>Record Owner</td>
</tr>
<tr>
<td>Subject</td>
<td>Extend an Offer</td>
</tr>
<tr>
<td>Due Date</td>
<td>Rule Trigger Date plus 1 days</td>
</tr>
<tr>
<td>Status</td>
<td>Not Started</td>
</tr>
<tr>
<td>Priority</td>
<td>High</td>
</tr>
<tr>
<td>Notify Assignee?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

All done! Now let’s go try out one of our new workflow rules.

**Look at What We've Done**

Let's try out our new "Send Rejection Letter" workflow rule and see what happens:

1. Click the Job Applications tab, and select a Job Application record.
2. Click **Edit**, and change the **Status** field to **Rejected**.
3. Click **Save**.

The Send Rejection Letter task automatically appears in the Open Activities related list on the Job Application detail page, as shown in the figure below.

![Figure 63: Open Activities Related List on Job Application Detail Page](https://na1.salesforce.com/00Tx04123s5k1)

Pretty neat, don't you think? Not only that, but if you check in the recruiter's email inbox, he should have received the following automatically-generated email notification:

Dave Carroll has assigned a new task to you: Send Rejection Letter
Job Application: JA-00002 Due Date: 8/18/2006 Comments:
Send standard rejection letter to candidate.

For more details on this task, click on the link below:

https://na1.salesforce.com/00Tx04123s5k1
The link in the email message takes the recruiter directly to the task detail page, where he or she can locate the contact information of the candidate and update the task status after the task is completed. The task also shows up in the My Tasks area on the recruiter's Home tab and a reminder message will pop up in another day if the recruiter hasn’t yet changed the task’s Status to Completed. Suddenly, our Recruiting app has become a lot more interactive and powerful!

**Try It Out: Creating a Workflow Rule That Updates Fields**

For our next use case, we want to create a workflow rule that ensures the owner of a new Position record is always a recruiter. For this workflow, we can start out by defining a workflow rule named Assign Position to Recruiter:

1. Click **Setup ➤ Customize ➤ Workflow & Approvals ➤ Workflow Rules**.
2. Click **New Rule**.
3. In the **Select object** drop-down list, choose Position, and click **Next**.
4. In the **Rule Name** text box, enter Assign Position to Recruiter.
5. In the **Description** text box, enter "Reassign position records to a recruiter if they were created by another type of employee."
6. Select the **Active** checkbox.

While we know that recruiters should almost always own Position records, we don’t want to impede the organization if there’s a special case in which a non-recruiter should own the record instead. Let's choose to evaluate this rule only when a record is created so that if an exception needs to be made, the workflow won't supersede any changes that were made by a recruiter.

7. Under "Evaluate this rule," select **Only when a record is created**.

Finally, we need to make sure that this rule is initiated whenever a Position record is created by someone who isn’t a recruiter or a recruiting manager. We can specify this filter criteria in a single row by using a comma in the Value column as follows:

8. In the first row of filters:
   - Set the **Field** column to Current User: Role.
   - Set the **Operator** column to not equal to.
   - Set the **Value** column to Recruiter, Recruiting Manager.
9. Click **Save**.

Now let's finish up by creating the field update action for this workflow rule:

10. In the Workflow Field Updates related list, click **New Field Update**.
11. In the **Name** text box, enter New Position Owner.
12. In the **Description** text box, enter "Assign the Position to the recruiting manager."
13. In the **Field to Update** drop-down list, choose Owner.

Once you make a selection in this drop-down list, new options appear just below depending on the selection that you made.

14. In the **Owner** drop-down list, choose User.
15. Click the lookup icon (🔍), and click Phil Katz.
16. Select Notify Assignee.
17. Click Save.

Not too hard! To try out this rule, simply make sure that you're logged in as a hiring manager other than Phil Katz, and create a new position. As soon as you return to the detail page for the position, you'll see that Phil has automatically been assigned as the record owner.

Now let's build one more workflow rule so we can see how to create a workflow email alert.

**Try It Out: Creating a Workflow Rule That Sends Email Alerts**

For our final workflow use case, let's create a workflow rule and email alert that sends a notification whenever a new Position is created. We want all the employees at Universal Containers to know when there are new positions available so they have the best opportunity to bring in referrals.

For this workflow rule, there's a step that we'll need to handle first: we need to design a template for what the email alert should look like.

**Introducing Email Templates**

Just as the AppExchange platform includes built-in tools for setting security permissions, tracking events and tasks, and building business logic with workflow, it also provides a built-in tool for writing emails to users and contacts in your organization. Email templates allow you to create form emails that communicate a standard message, such as a welcome letter to new employees or an acknowledgement that a customer service request has been received.

To personalize the content of an email template, we can use merge fields to incorporate values from records that are stored in the system. For example, if we wanted to address an email recipient by their first name, we could write an email template as follows:

```plaintext
Dear {!Contact.FirstName},

...
```

In this example, `{!Contact.FirstName}` is a merge field that brings in the first name of the contact to whom the email is addressed, so an email to John Smiley would read:

```plaintext
Dear John,

...
```

For our workflow alert, we can build an email template to notify users of new Positions that have been added to the system. We can use merge fields to include information from the Position record, such as its title and the required skills. Let's go do that now, and then we can get back to finishing up our final workflow rule.

**Try It Out: Building an Email Template**

To build a new email template, we have to go to the Administration Setup area:

1. Click Setup ➤ Communication Templates ➤ Email Templates.

Here you should see a list of all the email templates that have already been defined for your organization, including several sample templates from salesforce.com.
2. Click **New Template**.

We can choose to create a text, HTML, or custom email template. HTML and custom email templates are the same except that HTML templates allow you to specify a letterhead to give your email the same look and feel as other emails from the same source. To keep things simple, we'll just stick with a plain text email for now.

3. Select **Text**, and click **Next**.

![Figure 64: Defining an Email Template](image)

The New Template page allows us to define the email template itself. The gray area near the top is where we'll generate the merge field codes for the fields in the email template below it, so let's skip past it for now and start with the **Folder** drop-down list.

4. In the **Folder** drop-down list, choose **Unfiled Public Email Templates**.

The Unfiled Public Email Templates folder is a standard, public folder available in every organization. By keeping the email template in a public folder, it'll be available to other users who have permission to view and edit email templates.

5. Select the **Active** checkbox.

This option will make our email template available when we create our workflow alert.

6. In the **Email Template Name** text box, enter **Recruiting App: New Position Alert**.

**Tip:** To help keep your email templates organized, it's a good idea to preface any template name with the name of the app that uses it. Or, even better, you can create a public email template folder with the name of the app, such as Recruiting App Templates, and file all the relevant email templates in it.

7. In the **Encoding** text box, accept the default of General US & Western Europe (ISO-8859-1, ISO-Latin-1).

8. In the **Description** text box, enter "Send update email to all Universal Containers employees."

Now we get to the heart of our email template—the email's subject and body text.
9. In the **Subject** text box, enter New Open Position Alert: 

We want to put the title of the new position in the subject of our email, so we'll need to use our first merge field here, just after the colon in our subject. To get the proper code, we'll have to go back to the gray merge field area near the top of the page.

10. In the **Select Field Type** drop-down list, choose Position Fields.

Although there are many objects to choose from in the **Select Field Type** drop-down list, because we're creating an email template for a workflow rule, we're limited to the fields for the object that will be associated with that workflow—in our case, Position. That's because the workflow rule that uses this email template won't know about any individual records other than the Position record that triggered the rule's execution. If we put in fields from another object, they'd be left blank in our email, because there wouldn't be a record from which to pull the values.

Now let's grab the field we want.

11. In the **Select Field** drop-down list, choose Position Title.

In the **Copy Merge Field Value** text box, a merge field code appears for Position Title. We can cut and paste it to the end of our subject line so the subject now looks like this: New Open Position Alert: {!Position__c.Name}. When an email is generated from this template, {!Position__c.Name} will be replaced with the relevant position title.

Easy, right? Now let's finish the remainder of our email.

12. In the text area just below the **Subject** text box, enter the following text:

```
There's a new position open at Universal Containers!

Title: {!Position__c.Name}
Functional Area: {!Position__c.Functional_Area__c}
Location: {!Position__c.Location__c}
Job Description
{!Position__c.Job_Description__c}
Responsibilities
{!Position__c.Responsibilities__c}
Skills Required
{!Position__c.Skills_Required__c}
Educational Requirements
{!Position__c.Educational_Requirements__c}

If you know of anyone great who might be able to fill this role, please contact the hiring manager, {!Position__c.Hiring_ManagerId__c}. Thanks!
```

13. Click **Save**.

That's it for our email template. Now that it's done, we're ready to create our New Position workflow rule and alert.
Try It Out: Creating the New Position Workflow Rule and Alert

Now that we've built our email template, we're ready to build the workflow rule and email alert that use it. By now this procedure should be very familiar to you:

2. Click New Rule.
3. In the Select object drop-down list, choose Position, and click Next.
4. In the Rule Name text box, enter New Position Created.
5. In the Description text box, enter "Send an email to everyone whenever a Position record is created."
6. Select the Active checkbox.

We only want this rule to execute once, whenever a new position is created.

7. Under "Evaluate this rule," select Only when a record is created.
8. In the first row of filters:
   • Set the Field column to Status.
   • Set the Operator column to equals.
   • Set the Value column to Open. If you recall, "Open" is the default value of the Status picklist, so all new positions get this status by default.
9. Click Save.

Now let's create the email alert for this workflow rule:

10. In the Workflow Alerts area, click New Alert.
11. In the Description text box, enter New Position Alert Email.
12. Next to the Email Template field, click the lookup icon (🔍), and select Recruiting App: New Position Alert.

We want to send this email to everyone at Universal Containers, but for this workflow rule there's no obvious way of doing that. We can work around this by relying on our role hierarchy and sending the email to everyone in the CEO role and its subordinates.

13. In the Recipient Type Search field, choose Role and Subordinates.
14. In the Available Recipients list, select Role and Subordinates: CEO, and click Add.
15. Click Save.

And that's all there is to it! To test out this workflow rule, all you need to do is create a new Position record with a Status value of Open. Within a few seconds, all users within your organization will receive an email letting them know that it's just been created. Go ahead—try it!

Summing Up

Check out our Recruiting app now! By leveraging the AppExchange platform's built-in workflow tool, we've transformed our app from a glorified database into a fully functional application that provides real value to its users.
We've almost met all of the requirements we talked about back at the beginning of the book, but we're not done yet. In the next chapter, we're going to move beyond the native capabilities of the AppExchange platform, like security and workflow, and introduce the AppExchange Web Services API. This API provides the key to incorporating functionality from all over the Web and will help us create a truly powerful on-demand application. In fact, once you've mastered the tools available in the platform, the API will be the way that you can let your creativity soar—the functionality of any app you build on the AppExchange platform will be limited only by the Web itself!
Chapter 9

Moving Beyond Native Apps

In this chapter ...

• Introducing Web Services
• Introducing the AppExchange API
• Implementing a Composite Application Piece: The Candidate Mapping Tool

Up to this point, we’ve built a compelling AppExchange app using various parts of the platform. We’ve created a data model to store our recruiting information, built user interfaces for this data, applied security and sharing models to protect and share the data, and put workflow logic in place to help manage the data.

Because of the power of the AppExchange platform, we were quickly able to put all of these pieces together in the user interface and didn’t need to do any traditional coding. However, if we revisit the requirements of our Recruiting app, we’ll see there’s still another item to implement, and it’s not quite as straightforward.

Our requirements state that we need to implement a mapping feature that shows where each candidate is located. Clearly no such functionality is native to the AppExchange platform, and we certainly don’t have time to build our own mapping engine. Is this type of functionality even possible?

Of course it's possible! Since the AppExchange platform runs on the Web, we can leverage the power of other websites to implement features that would never be available just through our platform alone. We can use our HTML and JavaScript skills to mash up Web mapping services from Yahoo! with our own recruiting data, and then place this functionality in our own app.

To implement this kind of composite application piece, we’ll first need to learn about the technology that makes it possible: Web services in general and the AppExchange Web API in particular. Once we review those technologies, we’ll implement the candidate mapping feature for our Recruiting app and see how easy it is to create a rich, user-centric app with few (if any!) boundaries.

Tip: To get the most out of this chapter, you should be familiar with HTML, JavaScript, Java, and server-side scripting. However, even if you’re not an experienced developer, you can still follow along to gain a deeper understanding of what can be done with the AppExchange platform.
Introducing Web Services

First we need to look at the technology that makes composite application pieces like our candidate mapping feature even possible: Web services. A Web service is the mechanism by which two applications that run on different platforms, that were written in different languages, and that are geographically remote from each other, can exchange data using the Internet. Web services makes data exchange between two such applications as straightforward as the way that two processes can exchange data on a single computer.

The way that data is exchanged between two Web services is similar to the way that data that is exchanged between a Web browser like Microsoft Internet Explorer and a Web server. Just as a Web browser uses a common network protocol (HTTP over TCIP/IP) to download HTML files hosted on a Web server, a Web service can also use this same network protocol to download data from another Web service. The key difference is the actual data that is sent and received—Web services use XML instead of HTML.

So why are we talking about Web services? Well, it turns out that the AppExchange platform includes a Web services API that allows us to share data with other Web services on the Internet. Couple our Web services API with a tool like Yahoo! Maps, and we're well on our way to implementing our candidate mapping feature! Before we jump in, though, let's see how the AppExchange Web services API works.

Introducing the AppExchange API

The AppExchange API defines a Web service that enables full, reliable access to all of the data in our organization, including the ability to read, create, update, and delete records. To get a better idea of what we need to do to use the AppExchange Web services API, though, we need to discuss its most common calls, how it acts as a Web service, and how to authenticate API calls so that our data remains secure.

Common API Calls

In addition to the basic read, write, update, and delete calls, the AppExchange API also allows us to request metadata related to our standard and custom objects, maintain passwords, perform searches, and much more. Although the API supports more than 20 different calls, there are just four key calls that make up the majority of all API traffic:

Table 30: Commonly Used AppExchange API Calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>Returns data according to a SQL-like syntax called SOQL (Salesforce Object Query Language)</td>
</tr>
<tr>
<td></td>
<td>SOQL specifies what kind data we're looking for and the criteria that should be used to select the data.</td>
</tr>
<tr>
<td></td>
<td>For example, to retrieve the Id and Name field values for each account whose name ends with Tool, we'd use: (Select Id, Name From Account Where Name Like '%Tools').</td>
</tr>
<tr>
<td>create</td>
<td>Defines a new record for a particular object.</td>
</tr>
<tr>
<td>update</td>
<td>Edits an existing record according to its record ID.</td>
</tr>
</tbody>
</table>
Although the exact use and syntax of AppExchange API calls is beyond the scope of this book, you can find detailed documentation and API coding examples on the AppExchange Developer Network at www.salesforce.com/developer.

**SOAP, XML, and the AppExchange API**

When we talked about Web services above, we learned that they communicate with one another by passing XML data over the Internet. Fortunately, that doesn't mean that we have to brush off our XML code references and expect to bury ourselves in DTD files when we create our composite application piece. It turns out that the AppExchange API uses SOAP (Simple Object Access Protocol), a protocol that defines a uniform way of passing XML-encoded data. This is great for developers like us, since virtually every major programming language has a built-in ability to create and consume SOAP messages. As a result, all of the details about handling Web services XML will be hidden by familiar-looking objects in whatever language we choose for coding. That means we'll never need to actually write or read any XML ourselves when we're working on our app.

**Authenticating Calls to the AppExchange API**

Finally, let's talk about authentication. Some Web services are public, meaning that any request to the Web service will be honored and the request carried out. This makes the lives of developers easier, since they don't have to jump through any hoops in order to access the data that the services provide, but it also means that anyone can access the data! With the private, valuable data of your organization stored in the AppExchange platform, that's just unacceptable. Consequently, the AppExchange API is a private Web service, requiring authentication to access the data, just as a user of your Recruiting app is required to log in to access any data.

The AppExchange API provides authentication through the use of a session ID, a short-lived token that provides access to all API calls. This token is analogous to a driver's license—when you're issued a driver's license, it expires after a period of time, but until it expires you're authorized to drive an automobile. Likewise, when you're issued a session ID, it expires after a period of time (a configurable period of anywhere from 30 minutes to eight hours), but until it expires you're authorized to make Web service calls using the AppExchange API.

There are two ways of acquiring a session ID:

- Using the `login()` call (best for composite application pieces that originate from outside of the AppExchange platform user interface)
- Using merge fields (best for composite application pieces that reside inside the AppExchange platform user interface)

**Authenticating with the login() Call**

The first way to make authenticated Web service calls to the AppExchange API is by using the `login()` call. Similar to the way the main user login page in the user interface works, the `login()` call takes a username and password and executes a login sequence on https://www.salesforce.com/... If the login is successful, the `login()` call returns a session ID and URL. The session ID represents the user's authentication token, and the URL points to the host that contains data for the user's organization.
Note: For performance and reliability, the AppExchange platform runs on multiple hosts (for example, na1.salesforce.com, na2.salesforce.com, and so on), but data for any single organization is always consolidated on a single instance. As long as you use the URL that is returned from the login() call, you should never need to know the actual instance that hosts an organization's data.

This method of authentication is appropriate for composite components that originate from outside of the AppExchange platform user interface. For example, if we were to write an external Web page that accepts candidate resumés for different open positions, we could use the AppExchange API's login() call to upload these candidates and resumés to our Recruiting app.

Figure 65: Authenticating with the login() Call

Authenticating with Merge Fields

The second way you can make authenticated Web service calls to the AppExchange API is by using merge fields. A merge field is text that gets replaced with a value when the AppExchange server delivers a Web page to a user's browser, similar to the way Microsoft Word generates labels or form letters using a database of addresses. We'll talk about merge fields more when we get to writing the code for our candidate mapping tool. For now, you just need to understand that the platform provides the user's session ID, and that we can obtain this session ID using a merge field. That means our composite application pieces can acquire the session ID of a user who's already logged in to the system without forcing them to log in again.

Note: The merge field method is also nice because it tells us about the user who's currently logged in. When our composite application piece uses this information to make AppExchange API calls, the API returns only data to which the user has access, according to his or her security and sharing settings.

Figure 66: Authenticating with Merge Fields

So once you've obtained a session ID and server URL, what do you do with them? The answer to this question is dependent on what programming language you're using to create your composite application piece. In general terms, you'll include the session ID in every API call, and you'll direct your composite application piece to make the API request to the server URL that you obtained. We'll see examples of this below, when we finally get to coding our candidate mapping feature.
Implementing a Composite Application Piece: The Candidate Mapping Tool

All right! Now that we've reviewed Web services and the AppExchange Web services API, let's get started implementing the composite application piece for our Recruiting app: the Candidate Mapping tool. Our requirements state that we need to generate a map for all the candidates that have applied for a particular position so that we can better understand potential relocation costs associated with a new hire. Since Universal Containers has offices all over the country, this map will also help us assign candidates to an office if a particular position is open in more than one location.

We know we can create this tool by mashing up the AppExchange API and Yahoo! Maps with some basic JavaScript and HTML. But where do we put this code, and how do we incorporate it into our app?

Introducing S-Controls

It turns out that the AppExchange platform includes a particular component that gives us both a place to store our composite application piece code and an easy way of adding it to any page layout or tab. This component is called an s-control.

An s-control is essentially an instruction that defines what the platform should render on a page. S-controls can contain any type of content that you can display or run in a browser, such as a Java applet, an ActiveX control, a Microsoft Excel file, or a custom HTML Web form. In addition to any file you upload, a custom s-control contains HTML code that controls how the content is displayed and referenced in a custom link. S-controls provide a flexible, open means of extending the AppExchange platform interface beyond what is currently available.

S-controls come in three flavors: Custom HTML, URL, and Snippet:

Custom HTML S-Controls
An s-control that contains the actual HTML that should be rendered on a page. When saved this way, the HTML is ultimately hosted on an AppExchange platform server.

URL S-Controls
An s-control that contains an external URL that hosts the HTML that should be rendered on a page. When saved this way, the HTML is ultimately hosted by an external website.

Snippet S-Controls
An s-control that contains HTML or JavaScript that is meant to be included in other s-controls. Similar to a helper method that is used by other methods in a piece of code, a snippet allows you to maintain a single copy of HTML or JavaScript code that can be used in multiple s-controls.

For our Candidate Mapping tool, let's create a custom HTML s-control that contains standard HTML and JavaScript. A combination of these two technologies actually gives us a lot of power, especially when you consider all of the work being done in the AJAX realm lately.

Try It Out: Defining the Candidate Mapping S-Control

Let's go ahead and create our candidate mapping s-control. Once it's defined, we'll hook it in to our Recruiting app's user interface and play around with it to make sure it works:
Note: There’s a lot to talk about in this exercise, so we’re going to break it up in a couple of sections to help us organize our thoughts.

1. Click Setup ➤ Build ➤ Custom S-Controls.
2. Click New S-Control.

Try It Out: Setting S-Control Attributes

Upon clicking New S-Control, we get to the Custom S-Control Edit page.

![S-Control Edit Page — Attribute Entry](image)

The top of this page consists of three text boxes that allow us to enter a label, name, and a description of our s-control. The Label field contains the name for the s-control that will be displayed in various lists and page layouts. The Name field specifies the name that other s-controls can use to reference this s-control (particularly useful when using snippets), and the Description field provides a place to briefly indicate what the s-control does. (Other users will be very thankful if you provide good descriptions for your s-controls, because they’re often hard to decipher at a single glance.)

3. In the Label field, enter Candidate Map.
4. In the S-Control Name field, accept the defaulted value of Candidate_Map.
5. Select the HTML type.
6. In the Description field, enter "This map shows the location of each candidate for a particular position."

Try It Out: Entering Our Code into the S-Control Content Area

Below the s-control attribute area is the content area for our s-control—this is where we’ll enter the HTML and JavaScript code that’s going to implement our mapping tool. There’s a lot going on in this edit page. We’ll be using only a few areas of the page for this exercise: we’ll enter some code in the content area, check the syntax, and save the new s-control. Consult the online help to give yourself a quick orientation to all the widgets that are available on the S-Control Edit page.

The code block that we’re using provides a simple implementation of our candidate mapping tool. It combines candidate address information with Yahoo! Maps to display those candidates who have applied for a particular position. This isn’t really a coding book, but in the next few sections we’ll briefly examine a few lines of the code that are related to the AppExchange API (the bolded sections) to see how we’ve leveraged some of the concepts we talked about earlier in the chapter. Now let’s go ahead and enter our code into the s-control:

7. Enter the code block that appears below into the s-control’s large text area box.
8. Click **Check Syntax** to verify that you haven't made any typos.
9. Click **Save**.

**Tip:** Alternately, download a copy of the code from [www.salesforce.com/developer/devguide](http://www.salesforce.com/developer/devguide) and save it on your computer. (A file containing the code was included in the zip file you may have downloaded with the CSV import files in Chapter 6: Expanding the Simple App Using Relationships on page 65.) Open the code in your favorite text editor. Select all the lines of code, and then copy and paste them into the content area.

```html
<html>
<head>
<script src="/soap/ajax/8.0/connection.js"></script>
<script type="text/javascript">
function initPage() {
    loadMap();
}

var reload = true;
var map;
var link;
var geoPoints = [];
var dataPointCount = 0;
var pointCounter = 0;

function geoCodeEnd(r) {
    if (r.success == 1) {
        if (r.GeoPoint.Lat != 0) {
            geoPoints[geoPoints.length] = r.GeoPoint;
            pointCounter++;
        }
    }
}

function loadMap() {
    // Create a map object
    map = new YMap(document.getElementById('mapContainer'));
    // Display the map centered on given address
    YEvent.Capture(map, EventsList.endMapDraw, doQuery);
    YEvent.Capture(map, EventsList.onEndGeoCode, geoCodeEnd);
    map.addZoomLong();
    doQuery();
}

function doQuery() {
    if (reload == false) return;
    var positionId = "{{!Position__c.Id}}";
    var qr = sforce.connection.query("Select Candidate__c From"
Job_Application__c Where Position__c = '' + positionId + '';

    if (qr.size > 0) {
        dataPointCount = qr.size;
        var ids = [];
        for (var i=0;i<qr.records.length;i++) {
            ids[i] = qr.records[i].Candidate__c;
        }
        sforce.connection.retrieve("Id, Last_Name__c, First_Name__c, Street__c,
        City__c, State_Province__c, Current_Employer__c", "Candidate__c", ids,
        plotMarkers);
    }
    reload = false;
}

function doNav(arg) {
    window.parent.location = "/" + arg;
}

function plotMarkers(qr) {
    var records = qr;

    var counter = 1;
    if (records.length > 0) {
        for (var x=0;x<records.length;x++) {
            var db = records[x];
            if (db.Street__c != null && db.City__c != null && db.State_Province__c
                != null ) {
                var addr = db.Street__c + ', ' + db.City__c + ', ' +
                db.State_Province__c;
                var marker = new YMarker(addr);
                marker.addLabel(counter);
                var name = db.First_Name__c + ' ' + db.Last_Name__c;
                counter++;
                name = '<a href="/" + db.Id + " target='_top'">" + </a>";
                marker.personName = name;
                YEvent.Capture(marker, EventsList.MouseClick,
                    function() {this.openSmartWindow("<div> This is where " +
                    this.personName + " lives!<BR>&nbsp;</div>") });
                map.addOverlay(marker);
            }
        }
    }
}</script>

</head>

<body onload="initPage();">
<div id="mapContainer"></div>

</body>
</html>
Including the AJAX Toolkit Library

In the code sample, several key parts relating to the AppExchange API have been highlighted in bold font. The first, repeated here, is the bit of HTML needed to include the AJAX Toolkit JavaScript library:

```html
<script src="soap/ajax/8.0/connection.js"></script>
```

This library is what enables our s-control to interact with our data using the AppExchange Web services API. Once this line is processed by the browser, every AppExchange API call that is described in the API documentation is now available to the JavaScript in the s-control. The syntax for using this toolkit is very similar to the syntax provided in the API documentation.

Writing a Function for the Body Tag’s Onload Event

Another important concept is the `initPage()` function. Notice that it is being called when the page's `onload` event occurs:

```javascript
function initPage() {
    loadMap();
}
```

Whenever we're writing an s-control, it's very important to write a function that handles this event in order to ensure that our s-control's HTML elements have been completely sent to the browser before we run any JavaScript. This is necessary because Web browsers render the HTML received from a Web server serially, but they also start processing the JavaScript that has been included in an HTML page as soon as they can. This means that JavaScript may begin to run before all of the HTML elements have been sent to the browser. By starting our JavaScript execution when the body tag's `onload` event fires, we can rest assured that any elements we want to access with JavaScript will already be represented in the browser.

Using Merge Fields to Set the Session ID

The next bit of highlighted code demonstrates a key concept: merge fields. Merge fields are the mechanism by which we can obtain specific and contextual information for our s-control code.

```javascript
var positionId = " {!Position_c.Id} " ;
```

In this case, the Position ID is made known to the executing JavaScript code through the use of the ` {!Position.Id}` merge field. The ID field is common to every standard or custom object record, and uniquely identifies the record in the platform. We can use this ID value in an API query to limit the results to only those records that contain that ID. In this line of code, we're grabbing the position record that matches this ID so that we can obtain its related candidates for mapping.

In addition to the ID, we could have requested just about any and every other field of the Position object using the merge fields technique.

We can now use this value to create a SOQL statement that specifies what data we'd like to pull out of our app. Using the AJAX toolkit we don't need to worry about authentication. When the toolkit is sent down to the browser, the important authentication data required, `session id` and `server url`, are automatically merged into the toolkit code.

As you can see, merge fields represent a very powerful mechanism for requesting data from the AppExchange platform. Many s-controls use merge fields exclusively and forego the need to use the API to request data altogether. The only drawbacks to merge fields are that they are always read only, and they cannot cause data to be created or updated on the platform.
Making an AppExchange API Call

The next highlighted segment is an actual AppExchange API call:

```javascript
var qr = sforce.connection.query("Select Candidate__c From Job_Application__c Where Position__c = " + positionId + ");
```

This query call is asking for all the Candidate record IDs in the Job Application records whose Position IDs match the one given to us in the merge field. Once the method executes, we'll know which Candidate records we need to map.

Accessing AppExchange Platform Data

The last two highlighted snippets demonstrate how to access the data that resulted from our query:

```javascript
for (var x=0;x<records.length;x++) {
    var db = records[x];
    if (db.Street__c != null && db.City__c != null && db.State_Province__c != null ) {
        var addr = db.Street__c + ", " + db.City__c + ", " + db.State_Province__c;
        var marker = new YMarker(addr);
        marker.addLabel(counter);
        var name = db.First_Name__c + " " + db.Last_Name__c;
        ...
    }
}
```

Our query resulted in a set of Candidate records that matched the criteria of the query. To obtain the address value on each of these records, we can loop over the set. In JavaScript, a record is defined as an instance of an SObject object, so we'll assign each object in the set of records to a variable called `db`.

SObject objects provide various methods for retrieving and setting values on the record, including both a dot notation and a key notation for accessing field values. The key notation expects a field name as an index and returns the value from the field for that record. An example of the key notation is `db["First_Name__c"]` as opposed to the dot notation form, `db.First_Name__c`. Both accessor forms are case sensitive. Notice that there isn’t any XML, SOAP, WSDL, or other details of the actual Web service in this function. Instead, we have a simple object from which we access values using simple methods.

This ability to easily retrieve the information from our application is one of the most powerful features of the platform. The custom objects and fields that we created in previous chapters are all instantly available through the API.

Introducing Hooks and Targets

By now you should have a better feel for what s-controls are and how they allow us to create composite application pieces in our app. And if you’ve been following along really closely, you might also be asking yourself a couple of questions. For instance, how and when will our users see and use the candidate mapping s-control that we just defined? And, how does the platform know what Position ID to send us?

It turns out that we can answer both of those questions by talking about hooks and targets. A hook is a location in the user interface from which we can launch or kick off an s-control or URL—the target. Below is a table of all the hooks that we have available to us in the AppExchange platform, along with the targets that we can launch from those hooks:
When we defined the custom objects for our Recruiting app, we also defined tabs and page layouts to display those objects. Since our candidate mapping tool shows the locations of candidates for a particular position, we'll hook our s-control to the Position object's page layout with a custom link. The platform will know which Position ID to send us, because we'll see our s-control only when we're looking at a particular Position record.

**Try It Out: Creating a Custom Link**

Now that our custom s-control is written and saved to the platform, we're ready to create the hook on which to hang this code. The first step is to create a custom link.

1. Click **Setup ➤ Build ➤ Custom Objects**.
2. Click **Position**.
3. In the Custom Buttons and Links related list, click **New**.

![Figure 68: Custom Link Edit Page](image)

The custom link edit page should look familiar, since it closely resembles the s-control editor that we saw previously.

4. In the **Label** text box, enter **Candidate Map**.
5. In the **Link Name** text box, accept the defaulted value of **Candidate_Map**.
6. In the **Description** text box, enter "Shows the candidates for a position using Yahoo! Maps."

Depending on the type of action that we want to take, we can choose where to display our link in a Position record's detail page. Actions that don't have any affect on the record (like our Candidate Mapping tool) can be displayed in the Custom Links area of the record's detail page. Actions that affect the record itself should be displayed as a Detail Page Button, and actions that affect multiple Positions in a list should be displayed as a List Button.
7. Select Custom Link as the type.

Next choose the action that we want to perform: we can either open a new page, pop up a new window, or simply execute some JavaScript. Because we want our candidate map to appear as part of our Recruiting app, let's choose the default of Display in new window.

8. In the Behavior drop-down list, select Display in new window.

Because we implemented our Candidate Mapping tool with an s-control, the content source, or target as we called it above, should be Custom S-Control.

9. In the Content Source drop-down list, select Custom S-Control.

Selecting Custom S-Control changes the layout of the custom link edit page so that instead of a large text area box we see only a Custom S-Control drop-down list.

10. In the Custom S-Control drop-down list, select Candidate Map.

Notice that our s-control is listed in this drop-down list according to its label value.

11. Click Save.

After clicking Save, we are reminded that no users will be able to see the link until we add it to a Position page layout. Makes sense, right? Creating a custom link is similar to adding a custom field to an object—it's defined in the AppExchange platform database, but no user will be able to see it until we explicitly put it somewhere.

Try It Out: Adding a Custom Link to a Page Layout

So let's do it! All we need to do to finish implementing our Candidate Mapping tool is add it to our position page layout.

1. Click Setup ➤ Build ➤ Custom Objects.
2. Click Position.
3. In the Page Layouts related list, click Edit next to Position Layout.

By now this should be a familiar page to you!

4. In the View drop-down list on the right side of the page, select Position Custom Links.

This action causes the table below the drop-down list to be filled with all of the custom links that have been created for the Position object. Since we've created only one so far, that's all we'll see in this list.

5. Click and drag the Candidate Map custom link over to the Custom Links area of the page layout.

Unlike regular fields, custom links can be placed only in the special Custom Links layout section.

6. Click Save.

We did it! Although the coding of the s-control was a little tricky, every other step was just as straightforward as what we've already done in previous chapters. Let's now go try it out!

Try It Out: Running Our S-Control

To try out our new candidate mapping tool, all we need to do is visit a Position record.
1. Click the Positions tab.
2. Select the Benefits Specialist record.
3. In the Custom Links section, click **Candidate Map**.

![Figure 69: Our Working Candidate Mapping Tool](image)

Voilà! Notice how our s-control is embedded within our Recruiting app as if it were natively built. Also notice how we didn't need to do a special login just for our s-control, and that it could automatically figure out which Candidate records to display based on the Position where we were just visiting. Pretty cool!

Once again, welcome to the power and flexibility of the AppExchange platform. With fewer than a hundred lines of code, we've introduced non-native functionality that looks as if it were made expressly for our Recruiting app. Couple the AppExchange Web services API with the thousands of Web services that are available on the Internet today, and the possibilities are limited only by our imaginations!
Chapter 10

Learning More

In this chapter ...

• Help and Training Options
• AppExchange Podcasts
• AppExchange Partner Program

This book has introduced the basic technologies associated with the AppExchange platform. We've created a fully functional Recruiting app, and we've introduced the Web services API to show you how it can be used to build composite apps. We welcome your comments. The AppExchange Developer Network counts on your feedback and ideas. Go to the ADN discussion boards at www.salesforce.com/developer/community/index.jsp and let us know what you'd like to see in future versions!

We hope you can feel the power of the AppExchange platform. But this book can give you only a glimpse of what is possible. We've barely touched upon subjects like the AppExchange platform's robust reporting and analytics capabilities, or how to build integrations between the platform and your other enterprise systems. There's a lot more to cover!

As you get started, make sure to take advantage of the wealth of resources that are available to help you build and deploy your apps. The AppExchange Developer Network website at www.salesforce.com/developer and the resources listed below will point you in the right direction. Have a great time creating apps with the AppExchange platform!
Help and Training Options

The platform offers lots of help and training options:

• **Find Answers to Your Questions**

  Click **Help & Training** at the top of any page in the application. Enter your keywords in the Search box and click **Go!**. The search returns online help topics, knowledgebase solutions, and recommended training classes that match the keywords you entered.

• **Take Training**

  Select the Training tab of the Help & Training window, choose your role and geographic location, and click **View Classes!** to find free, online training classes. All kinds of training is available to ensure your success and application expertise:

  • Classroom training

    Gain the knowledge and skills you need through instructor-led workshops and courses that provide comprehensive, intensive, hands-on sessions, including the Salesforce Administrator Workshop, AppExchange Laboratory (AppLab), AppExchange API Workshop, and AppExchange Advanced Data Migration. All classes are taught by expert instructors, include customized and comprehensive training materials, and are offered in many convenient locations.

  • Custom training

    Salesforce.com can also bring the classroom to your organization.

  • Online learning

    Take advantage of more than fifteen online training classes, available to you on-demand, 24/7, and just few clicks away!

To learn more or register for training, contact Successforce Education Services at **EducationServices@salesforce.com**, or visit **success.salesforce.com**.

• **Download Tip Sheets and Best Practice Guides**

  Select the Help tab of the Help & Training window and click **Tips** in the task bar to view and download tip sheets, implementation guides, and best practices for specific features.

AppExchange Podcasts

Thanks to the hard work of our colleagues on the Successforce website, we have an impressive number of podcasts available on our iTunes channel. You can access our iTunes area by searching for "salesforce" on the iTunes Music Store.

This public podcast channel keeps the AppExchange community connected by providing free access to a wide range of best practices, case studies, and product- and platform-focused digital audio content via the iTunes Music Store. Access to expert AppExchange and CRM content is a quick download away. Now you can hear salesforce.com podcasts—covering everything from luminary interviews and thought-leadership presentations to roundtable discussions and best practices—whenever, wherever you like. Choose from over 70 podcasts. Happy listening!
AppExchange Partner Program

With the emergence of The Business Web™, companies can offer their services and applications to businesses over the Internet as easily as retailers and auctioneers can connect with online consumers. As AppExchange partners, more than 150 companies are already participating in this new chapter of computing by making their offerings available on The Business Web. The AppExchange partner program makes it easy for both new and established businesses to join this growing community of on-demand providers.

Contact a Partner Success Manager and join the AppExchange partner program today!
In this chapter ...

- Introducing Apex
- Sample Schema Used in This Chapter
- A Sample Apex Script
- Basics
- Statements
- Methods
- Apex API Calls, Packages, Triggers, and Debugging

But wait, there’s more!

We’ve spent the better part of this book introducing you to the AppExchange platform as it currently exists today. We highlighted some of the most exciting components of the platform, such as custom objects, security, and workflow, and we went through a very brief example of how we can use s-controls to incorporate our own custom code with data that’s stored in the AppExchange database. Taken together, these tools give you the building blocks to produce full-featured on-demand applications that can satisfy any business need.

However, despite the current power and flexibility of the AppExchange, we know there’s still more we can do to make it the ultimate platform for business Web applications. In fact, even as we’ve been writing this book, our development team has been working to take the platform to a whole new level. Although these new technologies aren’t yet generally available to the developer community, we didn’t want to miss this opportunity to give you a preview of them.

**Empowering Developers**

The cornerstone technology of the new AppExchange is a new scripting language called Apex. Apex represents a monumental step for AppExchange developers, because it allows something entirely new and powerful—the ability to run business logic on our servers. This means that as an AppExchange developer, you’ll be able to use the same tools that salesforce.com’s own development team uses to build our own apps, including our flagship CRM products. You’ll be able to build a whole new breed of AppExchange applications, featuring sophisticated processes and business logic, with even faster performance and absolute data integrity. If your primary interest is customization, Apex will enable you to fine-tune, reface, or even significantly remodel any existing apps to better meet your business’s needs. And finally, Apex will allow the AppExchange developer community to freely create and share re-usable components like widgets, applets, buttons, process modules, and more.
About This Chapter

What follows here is a preliminary technical overview of Apex. You'll find that it’s quite different from the earlier parts of this book in that it is primarily descriptive, aimed at explaining what this new language is rather than how to use it. As the initial documentation of this new language, it’s more in the style of a traditional developer reference.

Nevertheless, if you're an experienced developer, reading the next twenty-five or so pages will provide you with a solid overview of how Apex is constructed and the kinds of functionality that you'll have at your fingertips once this language goes live. You'll see for yourself how Apex combines the power of a procedural language with sophisticated database query functionality. You'll probably recognize the syntactical similarities to Java, which make it very easy to get up and running with Apex if you're at all familiar with Java. And most importantly, we're confident that you'll quickly see the power that Apex brings you as a developer, and that you'll be as excited about the possibilities of Apex and the future of the AppExchange platform as we are!

Note: If you're interested in keeping up with the latest on Apex, we'll be posting news, information, and up-to-date resources at http://developer.appexchange.com/apex/.

Important: Any unreleased services or features referenced in this chapter are not currently available and may not be delivered on time or at all. Customers who purchase our services should make their purchase decisions based upon features that are currently available.
Introducing Apex

The AppExchange platform today provides a flexible data-level API that is analogous to the basic SQL commands: SELECT (via SOQL, the Sforce Object Query Language), INSERT, UPDATE, and DELETE—also referred to as Data Management Language (DML). Developers can issue these atomic API commands from client-side programs (typically written in Java, JavaScript using the AJAX Toolkit, or .NET) where the controlling logic is not present on AppExchange platform servers.

Aside from the performance costs of making multiple round-trips to AppExchange servers to accomplish common business transactions, this model cannot run complex multi-object manipulations in a single transaction. This diminishes the utility of the AppExchange as a robust platform for developers.

Apex is a procedural extension to the AppExchange API that allows flow control and transaction control to be executed on the server in conjunction with these basic database API calls, both SOQL and DML. The language's motivation is exactly the same as Oracle's PL/SQL—namely to thread together multiple SOQL and DML statements as a single unit of work on the server.

Apex is not a general purpose programming language because it is data-focused and does not have any ability to render the user interface. We anticipate it will be used by AppExchange developers in the same way that traditional Oracle application developers have used PL/SQL.

Apex Syntax and Semantics

Apex syntax and semantics mimic Java, including:

- Variable and expression syntax
- Block and conditional syntax
- Loop syntax
- Object and array notation
- Pass by reference

Where Apex uses embedded AppExchange-specific concepts, the syntax and semantics are easy to understand and encourage efficient use of the API.

The Apex language is strongly typed with direct (non-quoted) references to object and field names, both standard and custom. This is the primary motivation for a new language and runtime; we have full control over the syntax and the ability to reference dynamic schema elements. Writing our own parser and runtime model also allows us to handle AppExchange API-specific concepts such as query and queryMore cursors and other common idioms.

Apex Runtime

Apex scripts can include anonymous blocks to be executed on-the-fly as well as stored packages and triggers. They are parsed into an abstract syntax tree (ANTLR, ANother Tool for Language Recognition, is the basis of the parser) and the resulting compiled form is then executed at runtime like any other procedural logic on the AppExchange server.

The compiled classes are represented in the AppExchange Java application server like other utility classes, such as the ones that already exist to represent SOQL queries. Many of these Apex classes sit on top of existing AppExchange API data structures and classes. When a script is parsed and executed, it runs with a context user,
just as any other request or batch-process runs in the application. In this way the Apex runtime should be seen as just another piece of the normal AppExchange application server, but one that has additional rules and governors to prevent infinite loops.

**Sample Schema Used in This Chapter**

The examples used in this chapter rely on the standard Account and Contact objects. These objects are related by the foreign key field `Contact.AccountId` (child to parent). The reverse relationship from Account to Contact (from parent to child, also called the child relationship name) is called Contacts.

The following standard and custom fields are referenced on these objects:

<table>
<thead>
<tr>
<th>Table 32: Account Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Field Name</strong></td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>NumberOfEmployees</td>
</tr>
<tr>
<td>AnnualRevenue</td>
</tr>
<tr>
<td>OutstandingShares</td>
</tr>
<tr>
<td>TotalShares</td>
</tr>
<tr>
<td>Contacts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 33: Contact Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Field Name</strong></td>
</tr>
<tr>
<td>LastName</td>
</tr>
<tr>
<td>AccountId</td>
</tr>
<tr>
<td>OutstandingShares</td>
</tr>
<tr>
<td>FirstName</td>
</tr>
</tbody>
</table>

**A Sample Apex Script**

Before describing the language syntax in detail, let’s review the following sample Apex script to grasp the brevity and power of the language. This script will help you understand the detailed syntax described in Basics on page 154.

The sample Apex script:

1. Cleans up old data.
2. Inserts Accounts.
The Account Ids are written back to the script array, so we can reference those Ids when we create the Contacts in the last step.

3. Supplies the name standard field and the outstandingshares custom field.

4. Inserts two Contacts for each Account.

```java
Integer NUM = 10;

Account[] accs;

// Clean up old data
accs = [select id from account where name like 'test%'];
delete accs;
commit;

accs = new Account[NUM];

for (Integer i = 0; i < NUM; i++) {
    accs[i] = new Account(name='test ' + i, outstandingshares__c=i);
}
insert accs;

Contact[] cons = new Contact[0];
for (Account acc : accs) {
    cons.add(new Contact(lastName=acc.name + '1', accounted=acc.id));
    cons.add(new Contact(lastName=acc.name + '2', accounted=acc.id));
}
insert cons;

Now suppose we're writing an upgrade script that needs to copy the outstandingshares value from the Account object into the child Contact object. This can be accomplished simply in Apex with a foreign key SOQL relationship query and an update operation:

```java
cons = new Contact[0];
for (Contact con : [select id, account.outstandingshares__c
    from contact where lastname like 'test%']) {
    con.outstandingshares__c = con.account.outstandingshares__c;
    cons.add(con);
}
update cons;
```

Finally, we can use the script to move data from the child records to the parent using aggregate relationships. For example, suppose we wanted to sum this new field from the list of child Contacts back into the parent Accounts:

```java
accs = new Account[0];
for (Account acc : [select id, (select outstandingshares__c
    from contacts)
    from account where name like 'test%']) {
    Double tot = 0;
    for (Contact con : acc.contacts) tot += con.outstandingShares__c;
    acc.totalshares__c = tot;
    accs.add(acc);
}
update accs;
```

Now that we've seen some examples of Apex in action, let's discuss the language's basics.
Basics

Local Variables and Expressions

Local variables can be defined at any point in a block and assigned a scope from that point forward. Sub-blocks cannot redefine the same variable name (hiding the variable in the parent block), since Java does not allow this, but parallel blocks can reuse a variable name. For example:

```java
Integer i;
{
    // Integer i;   Not allowed
}
for (Integer j = 0; j < 10; j++);
for (Integer = 0; j < 10; j++);
```

All variables and expressions have a type that is:

- A primitive type
- The special type Null for the constant null
- A concrete SObject type (e.g. Account, Contact, MyCustom__c)
- An exception
- An array of one of the following:
  - A primitive
  - A concrete SObject type

The special value `null` has a Null type and can be assigned to any variable.

Methods that return no value have the void type.

Type checking is strict and enforced at compile time. For example, an SObject field of type Integer cannot be assigned a String value—this error is caught by the parser.

Note: Compile-time exceptions are returned as specific API fault codes, with the line number and column of the error.

Declaration

Variables are declared with Java-style syntax. For example:

```java
Integer i = 0;
String s;
Account a;
Account[] accs;
```

As with Java, multiple variables can be declared and initialized in a single statement using comma separation:

```java
Integer i = 0, j, k = 1;
```
Primitives

The primitive data types are:

Integer, Double, Date, Datetime, String, Id, Boolean

All variables allow null as a value and are initialized to null, including Boolean.

The initial value can be overridden when the variable is defined, by providing an expression following the equality operator (like in Java).

- The Boolean literals are true and false.
- Integer and Double literals are distinguished by the decimal point.
- Strings have the following features:
  - They must be surrounded by single quotes.
  - The same parsing rules apply as SOQL strings.
  - String field values can never be empty (only null).
  - String field values are always trimmed (the same semantics as Oracle and the AppExchange API). However, script-level String values can be empty or have leading or trailing spaces.
  - The same escape sequences are currently used as for SOQL strings.
- Date and Datetime literals are also specified in the same format as in SOQL.
- Ids are special strings that can only take valid AppExchange API 18-character Id values. Note that setting an Id to a 15-character value will actually set it to its 18-character representation. Any invalid Id will be rejected with a runtime exception.

The type AnyType cannot be used in a variable declaration, but AnyType fields in the AppExchange data model (such as EntityHistory.NewValue) are of this type – they must be converted at runtime using the valueOf() static methods.

SObject Abstract Type

An SObject type represents a row of data and can be declared using the API name of the SObject. For custom objects, use the usual __c syntax. For example:

```apex
Account a1;
Account a2 = new Account();
Account a3 = new Account(name='xxx');
CustomNoParent__c c;
```

**Note:** SObject is an abstract variable type. Variables cannot be defined with this type, instead they are declared with the specific SObject type name as above.

The Apex parser tracks the dependent custom objects used in a script (both at the script syntax level as well as embedded in SOQL statements) when the script is parsed and validated. The parser throws an exception when an invalid name is used. Note that read/write access and field-level security are honored based on the context user.
SObject fields can be accessed or changed with simple dot notation. For example:

```java
Account a = new Account();
Contact c = new Contact();
System.debug(a.name);
a.name = 'xxx';
System.debug(c.account.name);
```

All SObject variables are initialized to null, but can be assigned a valid object reference with the new operator, as illustrated in the previous examples. The syntax allows the specification of initial field values (validated against the schema) using name=value tuples.

All field references are validated against actual field names, including __c for custom fields. A parse-time exception is thrown when an invalid field name is used.

**Note:** Apex tracks the dependencies of scripts on custom fields.

In the example above, we're following a relationship from Contact to Account to get to the actual field value. The relationship value is null, so the overall expression evaluates to null. This expression does not yield a null pointer exception because you'll typically want to navigate multiple foreign keys and Apex doesn't check null values along the way, for efficiency. This syntax is useful when accessing fields of an SObject populated with a SOQL query that uses foreign key relationships.

SObject variables support the following operators:

```java
// Clear the map so that the SObject has no field values
s.clear();

// Clone the SObject so that two variables can point at
// the same data values, but not share the same internal
// memory
a.clone();
```

**Arrays**

Arrays are defined by following the basic type name with the [] characters. These characters must be the terminal portion of the type—arrays of arrays are not allowed. Arrays can be defined for any of the other types, but not Exception:

```java
Account[] accs1;
Account[] accs2 = new Account[]{};
Account[] accs3 = new Account[1]{new Account()};
Account[] accs4 = new Account[4];
Integer[] ints1;
Integer[] ints2 = new int[1, 2, 3];
Integer[] ints3 = new int[6];
```

All arrays are initialized to null but can be assigned a value with the array literal notation above, in either form.

Array variables support the following operators:

```java
// Clear the list to have 0 elements
a.clear();

// Add one element m to a. m must be of the
// appropriate type
a.add(<scalar mem>);
```
// Add all element of a2 to a. a2 must be of the
// appropriate type
a.addAll(<array a2>);

// Return the size of the array as an int
a.size();

// Remove the entry at the given index
a.remove(<integer index>)

// Remove the entry with the given id (SObject array only)
a.remove(<id>)

// Make a shallow copy of the list
a.clone();

// Make a deep copy of the list of SObjects - the objects
// themselves are also cloned
a.deepClone();

For more information, see Methods on page 169.

Arrays are zero-based as in Java. Array values can be accessed by position, with strong typing (the result is a
primitive or SObject value of the Array member type). In this example, we assign to an array position, overwriting
the value at that position:

```java
Account[] accs = new Account[]{};
Account a = new Account();
a.name = 'xxx';
accs.add(a);
System.debug(accs[0].name); // xxx
a.clear();
System.debug(accs[0].name); // null
```

This will have the following output:

- `'xxx'`
- `null`

**Arrays as Maps**

Since the Id of an SObject is so important we want to make it easy to find rows by their Id in the script context.
For this reason, arrays of SObjects, in addition to acting as lists, also act as maps from Id to SObject. Whenever
an SObject is added to an array (in a constructor, via add() or addAll(), or via positional setting), if that row
has an Id, then an internal pointer is kept with the Id-to-SObject mapping. Also, when insert is called on an
array, then, in addition to setting the new Ids on the rows themselves, the map is also updated. For example:

```java
Account acc = [select Id from account where name = 'xxx'];
Account[] accs = new Account[]{acc};
System.assert(accs[acc.id] != null); // true
```

This feature allows easy "script hash joins" against small lookup tables.

Note that if a new SObject row happens to be a member of multiple arrays, and insert is called on one of those
arrays (which sets the Id value on the row), then only the inserted array’s mapping will be updated. For example:

```java
Account acc = new Account(name='xxx');
Account[] accs1 = new Account[]{acc};
```
Account[] accs2 = new Account[]{acc};
insert accs1;
System.assert(!accs2[acc.id]);

Also note that it is illegal to add the same row (with a non-null Id) to the same array more than once. For example:

Account acc = new Account(name='xxx');
insert acc;
Account acc2 = [select id from account where id = :acc.id];
Account[] accs = new Account[]{acc};
// This would yield an error
// accs.add(acc2);

Finally, it is illegal for the same exact row to be present in an array more than once (even with null id), since that would lead to a violation of the mapping invariant when the array was inserted. For example:

Account a = new Account();
Account[] accs = new Account[]{a, a}; // Runtime exception

Case Sensitivity

Apex itself is case insensitive because SOQL is case insensitive, as are other scripting languages. Therefore, the variable and method names of Apex are also case insensitive:

Integer I;
// This would be an error
// Integer i;

In addition, any references to object and field names are case insensitive:

Account a1;
ACCOUNT a2;

And, of course, the SOQL statements themselves are fully case insensitive as well:

Account[] accs = [sELect ID From ACCouNT
where nAme = 'fred']

Also, the SOQL WHERE clause always uses the same filtering semantics as the online application (internationalization case-insensitive comparison of strings using collation). Script-level comparisons will obey these same semantics as SOQL, so the following expressions are all true:

'a' == 'A'
null < 'a'
!(null > 'a')

Script-level methods exist for binary string comparisons (for example, s1.equals(s2)) as well. For more information, see Methods on page 169).

Expressions

The most basic expression types are similar to Java:

- Literal
- New SObject or Array using the new operator. For example:
  - new Account( field_initializations)
  - new int[n]
  - new Account[]{elements}
- **LValues:**
  - Variable
  - Field reference of an SObject variable
  - A positional reference of an Array variable
- **Non-assignable field references:**
  - An Id reference of an SObject array (cannot be assigned)
  - An aggregate field of an SObject (for SOQL queries that use aggregate relationships). This type of expression yields a query result, much like an inline query. For more information about inline queries, see [Inline Queries](#) on page 160.
  - Inline queries. For more information, see [Inline Queries](#) on page 160.
  - Static methods. For an example, see [now](#) on page 169.

**Expression Composition and Operators**

SObject and Array expressions can additionally be followed by a chain of either method references or array position references, respectively, to form new expressions. This follows Java-style object method invocation syntax. For example:

- `new Account[]{new Account(name='xxx')}[0].name.length()`
- `[select id from account][0].name.toLowerCase()`

Additionally, Apex supports compound expressions based on the usual operators, with standard precedence. For example:

- The assignment and combinaded assignment operators: `=, +=, *=, -=, /=, |=, &=`, which return the result of the assignment (right associative)
- The ternary operator: `?:` (right associative)
- The Boolean operators: `&&, ||` (left associative)
  - `&&` binds tighter than `||`
  - Short-circuiting applies
- The comparison operators: `==, <, >, <=, >=, !=` (non associative)
  - Comparison operators are not tri-state: even though null can be an argument to various comparison operators below, the resulting Boolean expression is never null in Apex.
  - SOQL uses `=` for equality, but we want to mimic Java's `==`
  - `==` and `!=` compare object value equality, not reference equality:
    - In particular, they do not correspond to `==` and `!=` in Java. They currently use SOQL equality semantics. This means that for strings, these operators use collation according to the current user’s locale. This is because in Java `==` doesn’t even work for strings at all.
    - Testing for null and not null is done using the `==` and `!=` operators.
    - There is no IS NULL operator, as there is none in Java either.
Inequalities also behave the same as in SOQL:

- For numbers and dates, if either argument is Null then the expression is false.
- Null is always less than any String value.
- Boolean expressions cannot be used in inequalities, and neither can Ids.
- Only the 4 basic arithmetic operators are implemented (though unary negation is also allowed), and they come next in precedence order:
  - Integers and Doubles can be intermingled. If at least one value is a Double, the result is Double.
  - Dates can be the left side of an expression with Integers on the right hand side. The operation will return a new Date with the given number of days added or subtracted.
  - Datetimes can be the left side of an expression with Integers or Doubles on the right hand side. The operation will return a new Datetime with the given number of days added or subtracted, with the fractional portion corresponding to a fractional date.
  - String concatenation is supported using the + operator with any type of argument on the right-hand side.
- The Boolean negation operator !, unary negation – (and no-op +), unary pre-fix, and post-fix increment using ++ and ––
- Of course, parentheses can be used to change the normal operator precedence. For example:

\[ x += (y == 2) ? i++ \times 7 : (z = w = 4); \]

### Inline Queries

An inline query is a SOQL statement surrounded by square brackets ([query]) that allows on-the-fly SOQL evaluation in Apex. It evaluates (by running the query) to an array of SObjects, or to an Integer for `count()` queries. For example:

```plaintext
Account[] accs = [select id from Account
  where name = 'xxx'];
Integer i = [select count() from Contact
  where lastName = 'x'];
```

These types of inline query expressions can be used wherever Arrays (or Integers, in the case of `count()` queries) are used.

Inline queries can also be used to assign a single SObject value. When the LValue of an expression is a singleton SObject type, then Apex automatically assigns the one row from the query result to that LValue. A runtime exception results if zero rows are found or more than one row is found. For example

```plaintext
Integer i = [select count() from Account];
Integer i2 = 5 * [select count() from Account];
// This only works if one row is returned
Account acc = [select id from Account];
Account[] accs = [select id from Account];
```

**Note:** The last example (returning multiple rows) can only be used if the query results are returned in a single batch. If the query requires cursoring, because the results are very large, then a FOR loop must be used. The syntax above, if the results are too large, will cause a runtime exception.
The select statement can, in general, be any valid SOQL statement, including foreign key and aggregate joins. If foreign key joins are included, the resulting SObjects that are returned can be referenced using normal field notation. For example:

```plaintext
System.debug([select account.name from contact
    where firstname = 'xxx'][0].account.name);
```

Additionally, aggregate relationships in SObjects act as inline queries as well (in AppExchange WSDL the type of these fields is QueryResult, so they really are just like outer queries). For example:

```plaintext
for (Account a : [select id, name,
    (select lastname from contacts)
    from account
    where name = 'testAgg']) {
    Contact[] cons = a.contacts;
}
// This example works because we limit to only 1 contact
for (Account a : [select id, name,
    (select lastname from
    contacts limit 1)
    from account
    where name = 'testAgg']) {
    Contact c = a.contacts;
}
```

Finally and importantly, SOQL statements in Apex may reference script variables and expressions inline, preceded by a colon—these will be evaluated in script context before executing the SOQL statement. These expressions can be used as the filter literals in WHERE clauses as well as the numeric value in the LIMIT clause. For example:

```plaintext
Account A = new Account(name='xxx');
insert A;
Account B;
B = [select id from account
    where id = :A.id];
B = [select id from account
    where name = :('x' + 'xx')];
String s = 'XXX';
B = [select id from account
    where name = :s.toLowerCase()];
B = [select id from account
    where name = :'XXXX'.substring(0,3)];
B = [select id from account
    where name = :
        [select name from account
        where id = :A.id].name];
Contact C = new Contact(lastName='xxx',
    accountid=A.id);
insert new Contact[]{C, new Contact(lastName='yyy',
    accountId=A.id)};
B = [select id, (select id from contacts
    where id = :C.id)
    from account
    where id = :A.id];
// One contact returned
Contact D = B.contacts;
Integer i = 1;
B = [select id from account limit :i];
```
Statements

A block contains a series of statements. The outer script itself is considered a block.

Internal blocks are defined with curly braces as in Java. An internal block can be used wherever a statement is used. This allows the usual syntax for an if statement. For example:

```java
if (true)
    System.debug(1);
else
    System.debug(2);
```

Or,

```java
if (true) {
    System.debug(1);
    System.debug(2);
} else {
    System.debug(3);
    System.debug(4);
}
```

All non-block statements end with semicolon.

Assignment Statement

Apex supports the following assignment operators:

- =
- ++
- //
- +=
- -=
- *=
- /=
- &=
- |=

Assignment statements are specified as in Java:

```java
LValue = new_Value_expression | [query];
```

Where `LValue` can be:

- A simple variable
- An SObject reference, with the following limits:
  - Only fields in the main object.
  - Field-level security is honored based on the context user.
  - The Id field cannot be assigned or changed via Apex.
  - It is not legal to change field values through a relationship such as `Contact.account.name`. For example:
    - Only `Contact.accountId` can be assigned for a Contact.
• After saving this change, re-query to get other fields from the new Account.

• Examples of an SObject reference include:
  • acc.Id = <some id>; // Parser exception
  • acc.name = 'xxx';
  • acc.createdDate = <some date> // Parser exception, field not writable
  • con.accountId = acc.id;
  • con.account.name = 'xxx'; // Parser exception

• An array de-reference (optionally followed by an SObject de-reference). For example:
  • a[0] = 1;
  • accs[0].name = 'xxx';

It is important to note that assignment is done by reference. For example:

Account a = new Account();
Account b;
Account[] c = new Account[]{};
a.name = 'xxx';
b = a;
c.add(a);
System.debug(b.name);
System.debug(c[0].name);

The following is the output from the assignment by reference:
'xxx'
'xxx'

Similarly, two arrays can point at the same value:

Account[] a = new Account[]{new Account()};
Account[] b = a;
a[0].name = 'xxx';
System.debug(b[0].name);

The following output is from this assignment:
'xxx'

**Conditional Statement**

The conditional statement is like the Java conditional statement:

if (<Boolean expression>)
  <truestatement>
(else
  <falsestatement>)?

Conditional statements behave as in Java. There is no elsif statement as it is not needed.
Loops

There are five types of procedural loops, analogous to Java procedural loops:

```java
do statement while (expression);
while (expression) statement;

for (initial ; (condition); (inc)) statement;

for (type var : array) statement;
for (type var : [query]) statement;
```

The `do` and `while` loops function as in Java. The expression must be of type `boolean`.

The first `for` loop corresponds to the traditional Java loop of the form:

```java
for (int i=0, j=0; i < 100; i++);
```

**Note:** Multiple variables can be declared and initialized.

Apex first executes `initial`, which is typically an assignment statement, and then continues to perform the following tasks:

1. Tests the `condition`, and quitting when false.
2. Executes the `statement`.
3. Executes `inc`, which typically increments the variable set during `initial`.

The second `for` loop assigns the variable `var`, which must be of the array type of `array`, to each element in `array` successfully, and then runs `statement` for each value.

The third type of loop corresponds to a `SELECT` statement and will be described in `Query Assignment Statement and for Query Loops` on page 164.

All loops allow these loop control structures:

```java
break;
continue;
```

`break` exits the entire loop, and `continue` skips to the next iteration.

In order to prevent infinite loops or very inefficient iterative database calls, the script execution framework throws a runtime exception if too many script statements are executed without any database statements. For more information, see `Execution Model and Governors` on page 168.

Query Assignment Statement and for Query Loops

There are two ways to pull data from SOQL into the scripting environment:

- Assignment:
  ```java
  LValue = [query] // We've seen this already
  ```

- Looping:
  ```java
  for (SObject_type var : [query]) statement;
  for (SObject_array_type var : [query]) statement;
  ```
Notice that the SOQL statements can refer to script expressions in a WHERE clause using the \texttt{expression} syntax. For example:

```java
String s = 'xxx';
for (Account a : [select id, name from account
    where name like :(s+'%')]) {
    
}
```

The queries used in both formats can reference host script variables in the WHERE clause.

The first query format we have seen already. It can retrieve either the \texttt{count()} of a query or a small number of result rows (one batch) into a host array or single SObject.

The second and third formats retrieve all SObjects, using efficient chunking via the API calls \texttt{query} and \texttt{queryMore}.

The difference between the two formats is whether the script sees one row at a time, using an SObject variable, or batches of rows at one time using an SObject \texttt{array} variable. The single-row format is easily understood, but if DML statements are used inside the loop, this could lead to inefficient writing operations. The array format is more complicated, but it can be more efficient, because it facilitates the use of bulk DML calls. In this array format, the loop \texttt{statement} is executed \textbf{once per chunk and not once per row}.

\texttt{break} and \texttt{continue} can be used for both formats. When using the array form, \texttt{continue} skips to the next chunk.

Notice that, as with inline queries, aggregate relationship values act as query results and can be used wherever an inline SOQL statement is used.

Examples:

```java
// Find one account and count its contacts, demonstrating
// both an outer level FOR loop as well as one on an
// aggregate relationship
for (Account a : [select id, name,
    (select lastname from contacts)
    from account where name = 'testAgg']}) {
    Integer i = 0;
    for (Contact c : a.contacts)
        i++;
}
```

\textbf{DML Statements}

The DML statements are of the form:

- \texttt{insert SObject};
- \texttt{insert SObject_array};
- \texttt{update SObject};
- \texttt{update SObject_array};
- \texttt{upsert SObject};
- \texttt{upsert SObject_array};
- \texttt{optional_matching_field_name}; // Id-field is
  // used by default
- \texttt{delete SObject};
- \texttt{delete SObject_array};

All statements perform the same logic as the underlying API bulk calls, with all workflow and other processing done in Java. This is why Apex is important as a wrapper for the API.
This also means that API fault codes can be thrown, such as a missing required field or field integrity exception. Save errors that result from DML faults can be trapped and handled. For more information, see Try and Throw Statements (Exception Handling) on page 166.

The `insert` statement writes the Ids of the inserted rows back into the SObjects being saved. For arrays, Ids are written back to the script array, so you can reference those Ids. Inserting records that already have Ids produces an error. The contents of the SObjects are not otherwise altered by any of the statements.

While SELECT statements can use foreign key joins to retrieve data from parent SObjects, the `insert`, `upsert`, or `update` statements can only set the foreign key Ids for the relationships in the root object. Scalar fields in the parent records themselves cannot be changed by a DML call.

### Transaction Control Statements

The results of all DML operations are held until a transaction control statement is issued:

```plaintext
commit;
rollback;
```

If a top-level script invocation has pending rows when it completes (that is, uncommitted and not rolled back), then a runtime exception is generated.

### Procedure Statements

Void functions that have side effects are typically executed as stand-alone procedure statements in script. Functions that return values can also be run as a statement. The return value is ignored, as in Java.

### Try and Throw Statements (Exception Handling)

Script authors can catch some runtime exceptions and handle them using `try...catch` syntax:

```plaintext
try {
    <statement>
    <statement>
} catch (<exception type> <exception variable name>) {
    <handler statement>
    <handler statement>
    <other optional catches>
} finally { // Optional
}
```

Exception variables (typically named `e`) are of the script type `Exception` and support the following methods:

```plaintext
// Returns a general type of the exception as a String.
// Examples values are: ARRAY, MATH, DML, QUERY e.getTypeName()

// Returns the textual message of the exception e.getMessage()
```

The limit exceptions caused by the execution governor cannot be caught. To raise the exception again:

```plaintext
throw e;
```
Specific types of system exceptions can be caught using standardized exception names include: `ArrayException`, `DmlException`, `NullPointerException`, `MathException`. The general name `Exception` can also be used, but it must be the last catch block:

```
try {
    // try block
} catch (System.ArrayException e) {
    // catch block
} catch (Exception e) {
    // catch block
}
```

### User-Defined Exceptions

Script authors can define their own exceptions in Apex. These can be thrown and caught as expected. The following syntax is for an anonymous block defining an exception:

```
defineException MyException;
try {
    if (i > 5) {
        throw new MyException('This is bad');
    }
} catch (MyException e) {
}
```

Packages can define an exception as well, in which case the exception name can be referenced outside the package using dot notation.

Variables can be declared of type `Exception`, as in Java:

```
Exception e;
if (<condition>) {
    e = new MyException();
} else {
    e = new OtherException();
}
throw e;
```

### Handling DML Exceptions

Bulk DML exceptions can be handled by getting information about the failed rows from the `Exception` object. The following methods are available for this purpose:

```
// Returns the number of failed rows from the last DML operation
e.getNumDml()

// Returns the original row number of the ith DML exception
e.getDmlIndex(int)

// Returns the API fault code of the ith DML exception
e.getDmlStatusCode(int)

// Returns the textual message of the ith DML exception
e.getDmlMessage(int)
```
Example:

```java
try {
    insert accs;
} catch (System.DmlException e) {
    for (Integer i = 0; i < e.getNumDml(); i++) {
        // process exception here maybe using
        // accs[e.getIndex(i)]
    }
}
```

## Execution Model and Governors

Several governors are in place to prevent runaway scripts. The execution environment tracks statistics on the number and types of statements that have been executed and throws a runtime exception if one category exceeds the (contextual) limit. The statistics tracked are:

- Total number of SOQL statements issued
- Total number of SOQL rows retrieved. A query that produces a cursor counts as the total number, even if not all rows are retrieved into the script.
- Total number of DML statements issued
- Total number of DML rows processed
- Total number of transaction control statements
- The total number of uncommitted DML rows since the last transaction control statements
- The total number of script statements that have been executed since the last database call. In this way, we prevent an infinite script-loop.

The Execution object allows specifying the ceiling for all of these limits. These limits should be set based on the context of the script, the organization in which it is running, the context user, functional testing, and other related areas of impact. The point is that you can easily prevent runaway resource utilization since you have full control over the actual runtime model.

The execution environment also has other limits in place: maximum stack depth for a single script when it calls procedures, including: maximum trigger stack depth when a stored trigger calls code that initiates a new trigger execution; maximum string size, and maximum array sizes.

In all cases in the list above, a runtime exception is thrown if the limit is exceeded, and the exception cannot be handled (like an Error in Java).

Other information available from the runtime model:

- The variables and their values in scope at the end of the execution
- The dependency information on schema objects and fields and packages
- The debug log:
  - Logging level is specified in code and at script invocation. It takes one of four values: None, UserDebug Statements, Database Calls, or AllStatements. Higher levels of logging include all lower levels.
  - When AllStatement logging is turned on, the log will contain verbose information about every execution statement and expression, including actual values processed.
  - The `System.debug()` method writes to this log and is level dependent.
- The stack trace
Methods

Static methods are invoked with no object context and, like Java, they are always of the form:

```
Class.method(args)
```

All other methods occur on expressions of a particular type. Note that if the source object expression evaluates to null then the method invocation will always cause a null pointer exception in script execution.

The following tables summarize available methods by object type. The final section of this chapter discusses user-defined methods.

System Static Methods

Table 34: System Static Methods

<table>
<thead>
<tr>
<th>Package</th>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>now</td>
<td>Datetime</td>
<td></td>
<td>The current date and time</td>
</tr>
<tr>
<td></td>
<td>today</td>
<td>Date</td>
<td></td>
<td>The current day</td>
</tr>
<tr>
<td></td>
<td>assert</td>
<td>Void</td>
<td>Boolean condition, <em>any</em> message</td>
<td>If the first argument is false, then a runtime exception is thrown with the second argument as part of its message</td>
</tr>
<tr>
<td></td>
<td>debug</td>
<td>Void</td>
<td><em>any</em> message</td>
<td>Writes the argument, in string format, to the execution debug log</td>
</tr>
<tr>
<td>UserInfo</td>
<td>getUserId</td>
<td>String</td>
<td></td>
<td>The context user's Id</td>
</tr>
<tr>
<td></td>
<td>getFirstName</td>
<td>String</td>
<td></td>
<td>The context user's first name</td>
</tr>
<tr>
<td></td>
<td>getLastName</td>
<td>String</td>
<td></td>
<td>The context user's last name</td>
</tr>
<tr>
<td></td>
<td>getUserRoleId</td>
<td>String</td>
<td></td>
<td>The context user's role Id</td>
</tr>
<tr>
<td></td>
<td>getDefaultCurrency</td>
<td>String</td>
<td></td>
<td>The context use's default currency code, for a multi-currency organization</td>
</tr>
<tr>
<td></td>
<td>getLocale</td>
<td>String</td>
<td></td>
<td>The context use's locale</td>
</tr>
<tr>
<td></td>
<td>getLanguage</td>
<td>String</td>
<td></td>
<td>The context user's language</td>
</tr>
<tr>
<td></td>
<td>getOrganizationId</td>
<td>String</td>
<td></td>
<td>The context organization's Id</td>
</tr>
</tbody>
</table>
### Chapter 11: Things to Come: A Sneak Preview of Apex

<table>
<thead>
<tr>
<th>Package</th>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getOrganizationName</td>
<td>String</td>
<td></td>
<td>The context organization's company name</td>
</tr>
<tr>
<td>Currency</td>
<td>newInstance</td>
<td>Currency</td>
<td>Double, String</td>
<td>Creates a Currency literal. This is the only place that such a type is created in the language. It is solely for WHERE clauses on SOQL</td>
</tr>
<tr>
<td>Integer</td>
<td>valueOf</td>
<td>Integer</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>Integer</td>
<td>valueOf</td>
<td>Integer</td>
<td>AnyType</td>
<td>Cast as Integer</td>
</tr>
<tr>
<td>Double</td>
<td>valueOf</td>
<td>Double</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>Double</td>
<td>valueOf</td>
<td>Double</td>
<td>AnyType</td>
<td>Cast as Double</td>
</tr>
<tr>
<td>Boolean</td>
<td>valueOf</td>
<td>Boolean</td>
<td>AnyType</td>
<td>Cast as Boolean</td>
</tr>
<tr>
<td>String</td>
<td>valueOf</td>
<td>String</td>
<td>Integer</td>
<td>Same as Java</td>
</tr>
<tr>
<td></td>
<td>valueOf</td>
<td>String</td>
<td>Double</td>
<td>Same as Java</td>
</tr>
<tr>
<td></td>
<td>valueOf</td>
<td>String</td>
<td>AnyType</td>
<td>Output as String</td>
</tr>
<tr>
<td></td>
<td>valueOf</td>
<td>String</td>
<td>Date</td>
<td>Output the date in the standard format &quot;yyyy-MM-dd HH:mm:ss&quot;</td>
</tr>
<tr>
<td></td>
<td>valueOf</td>
<td>String</td>
<td>Datetime</td>
<td>Output the date in the standard format &quot;yyyy-MM-dd HH:mm:ss&quot; for the local time zone</td>
</tr>
<tr>
<td></td>
<td>valueOfGmt</td>
<td>String</td>
<td>Datetime</td>
<td>Output the date in our standard format &quot;yyyy-MM-dd HH:mm:ss&quot; for the GMT time zone</td>
</tr>
<tr>
<td>Datetime</td>
<td>valueOf</td>
<td>Datetime</td>
<td>String</td>
<td>Parse the string using the standard date format &quot;yyyy-MM-dd HH:mm:ss&quot; in local time zone</td>
</tr>
<tr>
<td></td>
<td>valueOfGmt</td>
<td>Datetime</td>
<td>String</td>
<td>Parse the string using the standard date</td>
</tr>
<tr>
<td>Package</td>
<td>Name</td>
<td>Return Type</td>
<td>Arguments</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>format &quot;yyyy-MM-dd HH:mm:ss&quot; in GMT time zone</td>
</tr>
<tr>
<td></td>
<td><code>valueOf</code></td>
<td>Datetime</td>
<td>AnyType</td>
<td>Cast as Datetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>newInstance</code></td>
<td>Datetime</td>
<td>Integer, Integer, Integer, Integer, Integer, Integer</td>
<td>Construct a Datetime from year, month (1=Jan), day, hour, minute, second in local time zone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>newInstanceGmt</code></td>
<td>Datetime</td>
<td>Integer, Integer, Integer, Integer, Integer, Integer</td>
<td>Construct a Datetime from year, month (1=Jan), day, hour, minute, second in the GMT time zone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>newInstance</code></td>
<td>Datetime</td>
<td>Integer, Integer, Integer</td>
<td>Construct a Datetime from year, month (1=Jan), day at midnight in local time zone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>newInstanceGmt</code></td>
<td>Datetime</td>
<td>Integer, Integer, Integer</td>
<td>Construct a Datetime from year, month (1=Jan), day, at midnight in the GMT time zone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date</td>
<td>String</td>
<td>Parse the string using our standard date format &quot;yyyy-MM-dd HH:mm:ss&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date</td>
<td>AnyType</td>
<td>Cast as Date</td>
</tr>
<tr>
<td></td>
<td><code>newInstance</code></td>
<td>Date</td>
<td>Integer, Integer, Integer</td>
<td>Construct a date from year, month (1=Jan), day</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>daysInMonth</code></td>
<td>Integer</td>
<td>Integer, Integer</td>
<td>Return the days in the month for the year, month (1=Jan) given</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>isLeapYear</code></td>
<td>Boolean</td>
<td>Integer</td>
<td>Return true if the year given is a leap year</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td><code>round</code></td>
<td>Integer</td>
<td>Double</td>
<td>Same as Java except Integer returned, error on overflow</td>
</tr>
<tr>
<td></td>
<td><code>floor</code></td>
<td>Double</td>
<td>Double</td>
<td>Same as Java</td>
</tr>
<tr>
<td></td>
<td><code>ceil</code></td>
<td>Double</td>
<td>Double</td>
<td>Same as Java</td>
</tr>
</tbody>
</table>
### Array Type Methods

**Table 35: Array Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Void</td>
<td></td>
<td>Sets the array to 0 length by dropping all elements</td>
</tr>
<tr>
<td>clone</td>
<td>Array (of same type)</td>
<td></td>
<td>Creates a shallow copy</td>
</tr>
<tr>
<td>contains</td>
<td>Boolean</td>
<td>Id</td>
<td>Whether the given Id is a member of the SObject array</td>
</tr>
<tr>
<td>deepClone</td>
<td>Array (of same SObject type)</td>
<td></td>
<td>Creates a deep copy</td>
</tr>
<tr>
<td>size</td>
<td>Integer</td>
<td></td>
<td>Returns the number of elements</td>
</tr>
<tr>
<td>add</td>
<td>Void</td>
<td>Element type of the array</td>
<td>Same as Java</td>
</tr>
<tr>
<td>addAll</td>
<td>Void</td>
<td>Array (of same type)</td>
<td>Same as Java</td>
</tr>
<tr>
<td>remove</td>
<td>Void</td>
<td>Integer or Id (for SObject arrays)</td>
<td>Remove the entry with the given index or Id</td>
</tr>
</tbody>
</table>
## SObject Type Methods

**Table 36: SObject Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Void</td>
<td></td>
<td>Clears all field values</td>
</tr>
<tr>
<td>clone</td>
<td>SObject (of same type)</td>
<td></td>
<td>Creates a copy of the SObject</td>
</tr>
</tbody>
</table>

## String Type Methods

**Table 37: String Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>Integer</td>
<td></td>
<td>Same as Java</td>
</tr>
<tr>
<td>toLowerCase</td>
<td>String</td>
<td></td>
<td>Same as Java</td>
</tr>
<tr>
<td>toUpperCase</td>
<td>String</td>
<td></td>
<td>Same as Java</td>
</tr>
<tr>
<td>indexOf</td>
<td>Integer</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>contains</td>
<td>Boolean</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>statsWith</td>
<td>Boolean</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>endsWith</td>
<td>Boolean</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>substring</td>
<td>String</td>
<td>Integer, Integer (optional)</td>
<td>Same as Java</td>
</tr>
<tr>
<td>compareTo</td>
<td>Integer</td>
<td>String</td>
<td>Same as Java</td>
</tr>
<tr>
<td>equals()</td>
<td>Boolean</td>
<td>String</td>
<td>Same as Java (binary comparison)</td>
</tr>
</tbody>
</table>

## Date Type Methods

**Table 38: Date Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Integer</td>
<td></td>
<td>Return year component of date</td>
</tr>
<tr>
<td>month</td>
<td>Integer</td>
<td></td>
<td>Return month component of date (1=Jan)</td>
</tr>
<tr>
<td>day</td>
<td>Integer</td>
<td></td>
<td>Return day of month component of date</td>
</tr>
</tbody>
</table>
### Datetime Type Methods

**Table 39: Datetime Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>year</code></td>
<td>Integer</td>
<td></td>
<td>Return year component (Local time zone)</td>
</tr>
<tr>
<td><code>month</code></td>
<td>Integer</td>
<td></td>
<td>Return month component (1=Jan) (Local time zone)</td>
</tr>
<tr>
<td><code>day</code></td>
<td>Integer</td>
<td></td>
<td>Return day of month component (Local time zone)</td>
</tr>
<tr>
<td>Name</td>
<td>Return Type</td>
<td>Arguments</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>-----------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>dayOfYear</td>
<td>Integer</td>
<td></td>
<td>Return day of year component (Local time zone)</td>
</tr>
<tr>
<td>hour</td>
<td>Integer</td>
<td></td>
<td>Return hour component (Local time zone)</td>
</tr>
<tr>
<td>minute</td>
<td>Integer</td>
<td></td>
<td>Return minute component (Local time zone)</td>
</tr>
<tr>
<td>second</td>
<td>Integer</td>
<td></td>
<td>Return second component (Local time zone)</td>
</tr>
<tr>
<td>yearGmt</td>
<td>Integer</td>
<td></td>
<td>Return year component (GMT time zone)</td>
</tr>
<tr>
<td>monthGmt</td>
<td>Integer</td>
<td></td>
<td>Return month component (1=Jan) (GMT time zone)</td>
</tr>
<tr>
<td>dayGmt</td>
<td>Integer</td>
<td></td>
<td>Return day of month component (GMT time zone)</td>
</tr>
<tr>
<td>dayOfYearGmt</td>
<td>Integer</td>
<td></td>
<td>Return day of year component (GMT time zone)</td>
</tr>
<tr>
<td>hourGmt</td>
<td>Integer</td>
<td></td>
<td>Return hour component (GMT time zone)</td>
</tr>
<tr>
<td>minuteGmt</td>
<td>Integer</td>
<td></td>
<td>Return minute component (GMT time zone)</td>
</tr>
<tr>
<td>secondGmt</td>
<td>Integer</td>
<td></td>
<td>Return second component (GMT time zone)</td>
</tr>
<tr>
<td>addYears</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of years to Datetime</td>
</tr>
<tr>
<td>addMonths</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of months to Datetime</td>
</tr>
<tr>
<td>addDays</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of days to Datetime</td>
</tr>
<tr>
<td>addHours</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of hours to Datetime</td>
</tr>
<tr>
<td>addMinutes</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of minutes to Datetime</td>
</tr>
<tr>
<td>addSeconds</td>
<td>Datetime</td>
<td>Integer</td>
<td>Add given number of seconds to Datetime</td>
</tr>
<tr>
<td>isSameDay</td>
<td>Boolean</td>
<td>Datetime</td>
<td>Return true if the two Datetimes correspond to the same day in the local time zone</td>
</tr>
</tbody>
</table>
### Exception Type Methods

**Table 40: Exception Type Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getType</td>
<td>String</td>
<td></td>
<td>Returns the type of exception, for example, DML</td>
</tr>
<tr>
<td>getMessage</td>
<td>String</td>
<td></td>
<td>Returns the user message</td>
</tr>
<tr>
<td>getNumDml</td>
<td>Integer</td>
<td></td>
<td>For DML exceptions returns the number of failed rows</td>
</tr>
<tr>
<td>getDmlIndex</td>
<td>Integer</td>
<td>Integer i</td>
<td>Returns the original row position of the i(^{th}) failed row</td>
</tr>
<tr>
<td>getDmlStatusCode</td>
<td>String</td>
<td>Integer i</td>
<td>Returns the API failure code for the i(^{th}) failed row</td>
</tr>
<tr>
<td>getDmlMessage</td>
<td>String</td>
<td>Integer i</td>
<td>Returns the user message for the i(^{th}) failed row</td>
</tr>
</tbody>
</table>

### User-Defined Methods

New methods can be defined in script, either in anonymous blocks or in stored packages. For example:

```java
Integer fact(Integer n) {
    System.assert(n > 0);
    if (n == 1) return 1;
    return n * fact(n - 1);
}
System.assertEquals(24, fact(4));
```

User-defined methods can be used anywhere that system methods are used—they can be recursive, they can have side effects (such as DML inserts that write back IDs), and so on.

Polymorphic methods are allowed. Apex first attempts to find a method with an exactly matching signature. If that is not found, then an approximate match is sought, using the few type coercion rules (for example, the "double allows integer" rule). If multiple methods are found, a parse-time exception is generated.

Arguments are passed by reference, so modifications are seen by the caller.

As in Java, methods can refer to themselves or to methods defined later in the same package or anonymous block. This occurs because Apex parsing happens in two phases to avoid the need for forward declarations.
Apex API Calls, Packages, Triggers, and Debugging

Apex and the AppExchange API

Apex defines these new AppExchange API calls:

```java
ExecuteAnonymousResult executeAnonymous(String block)
CompilePackageResult[] compilePackages(String packageDefinitions[])
CompileTriggerResult compileTrigger(String triggerDefinition)
RunTestsResult runTests(RunTestsRequest ri)
```

The two compile calls are used to create new stored Apex objects. You can use these calls via your development environment (IDE) to load text files that contain developer code. The calls return success or failure status as well as a description of parsing problems.

Status and package information is currently also visible in the AppExchange user interface in Setup. Packages are available under the Build setup section, and triggers appear on the custom object detail page and the standard object button page. Editing is also allowed in this location, via a simple web form.

The executeAnonymous call is used to compile and then execute script code without storing it in the metadata. This could be used, for instance, from a custom button to call back into a stored package. The return result includes:

- Status information for the compile and execute phases
- Error information
- Debug log content, written by System.debug()
- Detailed execution analysis when the application server is placed into appropriate logging mode
- The Apex stack trace of any uncaught script-level execution exceptions. Apex shows the package, method, or line number for each call stack element.

The runTests call is described in Testing on page 181.

Anonymous Apex Blocks

User-defined methods and user-defined exceptions can be defined on-the-fly in anonymous blocks. This facilitates sending self-contained scripts with user-defined methods that don’t need to be persisted in the metadata as a package. A user-defined method can refer to itself or later methods without the need for forward declarations. However, variables cannot be referenced before their actual declaration.

For example, the following is a legitimate self-contained anonymous block:

```java
Integer i = 0;
void myProc1() { myProc2(); }
void myProc2() { i++; }
myProc1();
```

Note that anonymous blocks are executed as the current user and will fail to compile based on permissions such as field-level security and read/write access to objects. This differs from the execution of package calls and triggers, as explained in Packages on page 178 and Triggers on page 180.
Packages

A package is a stored Apex script consisting of package-level variables (these act as thread-local variables, just like package-level variables in Oracle), package-level user-defined methods, package-level exceptions, and optional static initialization code.

Once compiled, packages can be invoked via method calls or variable references just like any other code (in an anonymous block, for instance).

By default, methods and variables have package scope only. To be used outside of the package, they must be defined with the public keyword. User-defined exceptions are always public, since they could be thrown from public routines anywhere, and therefore can be caught by external code.

Variables can be declared with the final keyword as well. The order of keywords must be public, final, and then the rest of declaration. This means that the variable can be assigned at most once, either in the declaration itself or via a static initializer.

Define constants in the manner illustrated by this example:

```apex
package pkg {
    define PkgException;
    public Integer pubInt;
    final Integer PRIVATE_INT_CONST;

    public void doIt() {
        // call me externally
    }

    Integer calculate() { return 2 + 7; }

    static {
        PRIVATE_INTCONST = calculate();
    }
}
```

Note that user-defined procedures cannot define local variables that hide a package-level variable of the same name. However, package-level variables can be referenced as `package_name.variable_name` anywhere, including in that package. The prefix does not have to be used inside the package.

Packages as WSDLs and Defining New API Calls

Apex allows the creation of new API calls—user-defined commands that take strongly typed arguments. The arguments and return types of these user-defined calls will be the actual Apex type (for example SObject types and arrays) and can be used from any client program. Each package defines its own endpoint. The WSDL can be downloaded from the package detail page in the AppExchange platform Setup user interface.

Package methods flagged with the keyword `webService` (methods must be public) are exposed in this manner. Invoking these methods always uses System context so that the current user’s credentials are not used. This means that any user who has access to these methods can perform the full power of that call, regardless of permissions, field-level security, read/write access, or sharing checks. Application developers must be careful when marking methods as `webService`. For example:

```apex
package myPackage {
    public webService String makeContact(
        String lastName, Account acc) {
```
Namespacing

AppExchange platform upgrading introduces the notion of packaging and namespacing into the system. Packaging with namespacing allows custom objects, custom fields, and other elements with the same developer name to exist in a single subscriber organization, disambiguated by the namespace prefix of their “Package AppExchange Application.” This prefix basically identifies the particular application to which the object belongs. An application developer chooses a globally unique namespace prefix and registers it as a managed package with the AppExchange registry.

Once packaged, custom objects, custom field names, and other elements are changed to have a unique prefix: `prefix__name__c`

These are fully-qualified names and can be used in SOQL statements or s-controls as well as Apex code to refer to these objects. However, these names are quite long and cumbersome. In addition, once an application is given an AppExchange namespace prefix, code that worked before the application became managed needs to be changed to add the prefix to all identifiers.

To help developers who may have large amounts of Apex code to modify in such an event, we provide a default namespace for schema names. When looking at identifiers, the parser considers the namespace in which the current object lives and will recognize other names in that namespace (application). Consequently, a stored package can refer to custom object and fields names directly, using `name__c` for those objects that are defined within its namespace (the same application). This is also true for s-controls when the s-control is packaged into an application. For top-level API calls such as `query` and `executeAnonymous`, Apex allows specification of the namespace for that API call via a SOAP header.

For Apex package method invocation, fully-qualified identifiers of the following form are allowed:

\[ namespace.package.method(args) \]

Use the special namespace `System` to identify system (built-in) static packages (for example, `System.System.debug()`). User-defined packages do not use the `System` namespace. Without the namespace prefix, system static packages and class names such as `Math` and `System` are hidden by user-defined packages with the same name.

For any expression `name1.name2...nameN`, Apex looks for the following in order of precedence:

1. Local variable `name1` with field reference `name2...nameN`
2. Package `name1` with static variable `name2` (with field reference `name3...nameN`)
3. If three or more terms, namespace `name1`, package `name2`, with static `name3` (with field reference `name4...nameN`)

For any method `name1.name2...nameN()`, Apex looks for the following in order of precedence:

1. Local variable `name1` (which may have field references), followed by method `nameN`
2. If there are two names: package `name1` with method invocation `name2`
3. If there are three names: namespace `name1`, package `name2`, method `name3`
4. If there are three or more names: package name1 with static variable name2 (which may have field references) followed by method nameN

## Triggers

Apex packages allow configuration of a save handler or trigger on a per-object-type and per-operation basis. At first, top-level customizable objects (as opposed to child objects such as ContactRole) allow triggers, and the definitions of these triggers will be stored in the cache server under PerEntitySchema for that entity type. The operations that invoke triggers are insert, update, or delete, with two events per operation: before and after. This mimics Oracle’s trigger definition mechanism. upsert will fire insert or update triggers as appropriate.

The before trigger can be used to reject changes before they occur. Script code will raise a validation exception, providing an optional field identifier in the case of an update operation and a user message that must be a reference to a translatable message, defined in the organization’s metadata.

After triggers can be used to affect changes in other entities, such as logging into an audit table or asynchronous event processing log. Both before and after triggers are fired within the main entity transaction.

Triggers cannot issue transaction control statements. Since triggers can cause other row changes to occur, which fire more triggers, Apex considers all such operations in a single unit and prevents infinite recursion.

Triggers execute in system context. This means that permission checks, field-level security, read/write permission, and sharing are ignored. This is similar to the way that workflow behaves: once defined in the organization, the triggers do not operate on behalf of the current user but instead as the administrator.

The same trigger body can handle multiple event types for an object—the syntax of the trigger definition determines these events. For example:

```
trigger accTrig on Account (before insert, before update, after insert, after update) {
    if (Trigger.isBefore()) {
        if (Trigger.new().name == 'bad') {
            throw new System.ValidationException('bad');
        }
    }
    if (Trigger.isInsert()) {
        System.assertEquals('xxx', Trigger.new().accountNumber);
        Trigger.new().accountNumber = 'yyy';
    } else if (Trigger.isUpdate()) {
        System.assertEquals('yyy', Trigger.old().accountNumber);
        System.assertEquals('zzz', Trigger.new().accountNumber);
    } else {
        if (Trigger.new().name == 'makeContact') {
            insert new Contact(
                lastName = Trigger.new().name,
                accountId = Trigger.new().id);
        }
    }
}
```

Implicit script variables are defined by a trigger to give the script author access to the runtime context. They are scoped to the Trigger system package:
### Variable Descriptions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInsert</td>
<td>Whether this trigger is being fired because of an insert operation</td>
</tr>
<tr>
<td>isUpdate</td>
<td>Whether this trigger is being fired because of an update operation</td>
</tr>
<tr>
<td>isDelete</td>
<td>Whether this trigger is being fired because of a delete operation</td>
</tr>
<tr>
<td>isBefore</td>
<td>Whether this trigger is being fired before the save has occurred</td>
</tr>
<tr>
<td>isAfter</td>
<td>Whether this trigger is being fired after the save has occurred</td>
</tr>
<tr>
<td>new</td>
<td>The new row image (for insert and update)</td>
</tr>
<tr>
<td>old</td>
<td>The old row image (for update and delete)</td>
</tr>
<tr>
<td>isApi</td>
<td>Whether the end user request is an API operation</td>
</tr>
<tr>
<td>isUi</td>
<td>Whether the end user request is an operation from the user interface</td>
</tr>
</tbody>
</table>

For an update trigger, both the old and new images are available. The new row image can be changed, but only in a before trigger (insert and update).

Note that triggers (and packages for that matter) are stored in the metadata with the version of the API used to compile them. The before triggers fire before we perform declarative rule validation. This ensures the declarative rules will always hold. Similarly, after the main save the after event fires before the commit takes place.

### Debugging

All API calls that can invoke Apex now support a debug facility that allows access to detailed information about the execution of Apex along with any explicit calls to `System.debug()`. A new SOAP input header `DebuggingHeader` allows the specification of a `LogType` which specifies logging granularity. The corresponding output header `DebuggingInfo` contains the textual log. When multiple triggers are invoked by a single command, each trigger logs its beginning and ending points.

Existing `create`, `update`, `delete`, and `upsert` commands all support this facility, as do the new `executeAnonymous`, `runTests`, and package-specific WSDL commands.

### Testing

In order to facilitate the same test-driven development methodologies for our application developers that we use internally in salesforce.com development, we encourage the creation of unit tests that are executable via the `runTestsAPI` call. Methods in packages are flagged as tests using the `testMethod` keyword. Methods must take zero arguments to be flagged as tests.

The `RunTestsRequest` structure allows the specification that all tests in all packages should be run, or only tests within a specific list of packages. The return structure explains how many total tests were run and returns error information for the failed tests.
In order that the tests be idempotent in terms of not modifying the state of the organization, the execution context for this API command does not allow executing any transaction control statements. For example:

```java
package myPackage {
    public testMethod void myTest() {
        ...
    }
}
```

### Handling and Reporting Uncaught Exceptions

If a script that you develop has a bug or otherwise allows a script-level exception to escape the script layer and appear as an error in the application, the following events will occur:

- A simple error message appears to the user (either in the user interface or API), identifying the application that has received the error and providing diagnostic information.
- Diagnostic information, including the Apex stack trace and the error message information given to the user, is sent to the registered email account of the third-party developer who created the script. No customer data is sent.
- If the exception is in a recursively called trigger, for example, the result of an insert in an outer trigger, the call fails with a `DmlException`, and DML exception methods can be used to access the failure reason.
- If the Apex engine itself causes an exception, for example, a genuine NPE in Java, the user will see an Internal Server Error.
Glossary

Apex

App
A collection of Salesforce components such as tabs, reports, dashboards, and custom s-controls that address a specific business need. Short for "application."

AppExchange
An on-demand application-sharing service and platform from salesforce.com that allows you to browse, install, and share apps and components built for the AppExchange platform. The AppExchange platform combines a powerful user interface, operating system, and customization and integration capabilities for CRM—plus a framework for development and deployment of new applications for your entire enterprise.

AppExchange API
An application programming interface that defines a Web service that provides direct access to all data stored in AppExchange from virtually any programming language and platform. See also Application programming interface (API) on page 183.

AppExchange app menu
A menu that enables users to switch between customizable applications (or "apps") with a single click. The AppExchange app menu displays at the top of every page in the Salesforce user interface.

AppExchange Developer Network (ADN)
The website at www.salesforce.com/developer that provides a full range of resources for AppExchange developers, including sample code, toolkits, an online developer community, and the test environments necessary for building apps.

AppExchange directory
A Web directory where hundreds of AppExchange apps are available to Salesforce customers to review, demo, comment upon, and/or install. AppExchange developers can submit their apps for listing on AppExchange if they wish to share them with the community.

Application programming interface (API)
The interface that a computer system, library, or application provides in order to allow other computer programs to request services from it and exchange data between them.

Auto number
A custom field type that automatically adds a unique sequential number to each record.

Client app
An app that runs outside the Salesforce user interface and uses only the AppExchange API—typically running on a desktop or mobile device. These apps treat the AppExchange platform as a data source, using the development model of whatever tool and platform for which they are designed. See also Composite app on page 184 and Native app on page 186.
Composite app
An app that combines native AppExchange platform functionality with one or more external Web services, such as Yahoo! Maps. Allows for more flexibility in the app and integration with other services, but may require running and managing external code. See also Client app on page 183 and Native app on page 186.

Controlling field
Any standard or custom picklist or checkbox field whose values control the available values in one or more corresponding dependent fields. See also Dependent field on page 184.

Custom field
Fields that can be added to customize an object for your organization's needs.

Custom HTML S-Control
An S-Control that contains the actual HTML that should be rendered on a page. When saved this way, the HTML is ultimately hosted on an AppExchange platform server. See also S-Control on page 188.

Custom link
A custom URL defined by an administrator to integrate your data with external websites and back-office systems.

Custom object
An object that you build to store information that's unique to your app. See also Object on page 186 and Standard object on page 188.

Customer Relationship Management (CRM)
A business strategy for developing and improving relations between companies and their customers.

Database
An organized collection of information. The underlying architecture of the AppExchange platform includes a database where your data is stored.

Database table
A list of information, presented with rows and columns, about the person, thing, or concept you want to track. See also Object on page 186.

Dependent field
Any custom picklist or multi-select picklist field that displays available values based on the value selected in its corresponding controlling field. See also Controlling field on page 184.

Email template
A built-in feature for writing emails to users and contacts in your organization that enables you to create form emails that communicate a standard message, such as a welcome letter to new employees or an acknowledgement that a customer service request has been received.

Entity relationship diagram (ERD)
A data modeling tool that helps you organize your data into entities (or objects, as they are called in the AppExchange platform) and define the relationships between them.

Field
A part of an object that holds a specific piece of information, such as a text or currency value.
Field dependency
A filter that allows you to change the contents of a picklist based on the value of another field.

Field-level security
Settings that determine whether fields are hidden, visible, read only, or editable for users based on their profiles.

Foreign key
A field whose value is the same as the primary key of another table. You can think of a foreign key as a copy of a primary key from another table. A relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another. See also Primary key on page 187.

Formula field
A type of custom field that automatically calculate its value based on the values of merge fields, expressions, or other values.

Function
Built-in formulas that you can customize with input parameters. For example, the DATE function creates a date field type from a given year, month, and day.

Group
A set of users that can contain individual users, other groups, or the users in a role. Groups can be used to help define sharing access to data.

Home tab
The starting page from which users can choose sidebar shortcuts and options, view current tasks and activities, or select each of the major tabs.

Junction object
A custom object that enables a many-to-many relationship between two other objects.

Layout
See Page layout on page 186.

Lookup relationship
A relationship between two objects that allows you to associate records with each other. On one side of the relationship, a lookup field allows users to click a lookup icon and select another record from a list. On the associated record, you can then display a related list to show all of the records that have been linked to it.

Manual sharing
Record-level access rule that allows record owners to give read and edit permissions to other users who might not have access to the record any other way. See also Record-level security on page 187.

Merge field
A field you can place in an email template, mail merge template, custom link, or formula to incorporate values from a record. For example, Dear {!Contact.FirstName}, uses a contact merge field to obtain the value of a contact record’s First Name field to address an email recipient by his or her first name.
Metadata-driven development
An app development model that allows apps to be defined as declarative “blueprints,” with no code required. AppExchange apps—their data models, objects, forms, workflows, and more—are defined by metadata.

Multitenancy
An application model where all users and apps share a single, common infrastructure and code base.

Native app
A type of AppExchange app that is built exclusively via metadata configuration and without coding. Runs entirely on the AppExchange platform without need for external services or infrastructure. See also Client app on page 183 and Composite app on page 184

Object
In AppExchange terms, an object is similar to a database table—a list of information, presented with rows and columns, about the person, thing, or concept you want to track. Each object automatically has built-in features like a user interface, a security and sharing model, workflow processes, and much more.

Object-level security
Settings that allow an administrator to hide whole tabs and objects from a user, so that they don’t even know that type of data exists. In the AppExchange platform, you set object-level access rules with object permissions on user profiles. See also Field-level security on page 185 and Record-level security on page 187.

On-demand computing
A new paradigm of computing in which you access apps over the network as a utility, rather than as a piece of software running on your desktop or in the server room.

One-to-many relationship
A relationship in which a single object is related to many other objects. For example, each Candidate may have one or more Job Applications.

Org-wide defaults
Settings that allow you to specify the baseline level of data access that a user has in your organization. For example, you can make it so that any user can see any record of a particular object that’s enabled in their user profile, but that they’ll need extra permissions to actually edit one.

Page layout
The organization of fields, custom links, related lists, and other components on a record detail or edit page. Use page layouts primarily for organizing pages for your users, rather than for security.

Picklist
A selection list of options available for specific fields, for example, the Country field for a Candidate object. Users can choose a single value from a list of options rather than make an entry directly in the field.

Picklist values
The selections displayed in drop-down lists for particular fields. Some values come predefined, and other values can be changed or defined by an administrator.
Primary key
A relational database concept. Each table in a relational database has a field in which the
data value uniquely identifies the record. This field is called the primary key. The relationship
is made between two tables by matching the values of the foreign key in one table with the
values of the primary key in another. See also Foreign key on page 185.

Profile
A setting that defines a user’s permission to perform different functions.

Record
A single instance of an object, for example, Software Engineer is a single instance of the
Position object.

Record-level security
A method of controlling data in which we can allow particular users to view an object, but
then restrict the individual object records that they’re allowed to see. See also Org-wide
sharing on page 185.

Related list
A section of a record or other detail page that lists items related to that record.

Relationship
A connection between two objects. Matching values in a specified field in both objects are
used to link related data; for example, if one object stores data about companies and another
object stores data about people, a relationship would allow you to find out which people
work at the company.

Role hierarchy
A record-level security setting that defines different levels of users such that users at higher
levels can view and edit information owned by or shared with users beneath them in the role
hierarchy, regardless of the organization-wide sharing model settings. See also Record-level
security on page 187.

Search layout
The organization of fields included in search results, in lookup dialogs, and in the recent
items lists on tab home pages.

Sharing model
A security model that defines the default organization-wide access levels that users have to
each other’s information.

Sharing rules
Rules that allow an administrator to specify that all information created by users within a
given group or role is automatically shared to the members of another group or role. They
also provide the ability to make automatic exceptions to org-wide defaults for particular
groups of users.

Snippet
A type of s-control that’s designed to be included in other s-controls. Similar to a helper
method that is used by other methods in a piece of code, a snippet allows you to maintain
a single copy of HTML or JavaScript that you can reuse in multiple s-controls. See also
S-Control on page 188.
SOAP (Simple Object Access Protocol)
A protocol that defines a uniform way of passing XML-encoded data.

SOQL (Salesforce Object Query Language)
A query language that allows you to construct simple but powerful query strings and to specify the criteria that should be used to select the data.

Standard object
A built-in object included with the AppExchange platform when you first start using it. You can also build your own custom objects to store information that’s unique to your app. See also Custom object on page 184 and Object on page 186.

S-Control
A component that you set up and define on the AppExchange platform to store your custom code. Use s-controls to create and display your custom data forms using components like custom links, web tabs, or custom buttons. For example, you can define a custom s-control containing JavaScript and address merge fields to display a map of a contact's address. See also Custom HTML S-Control on page 184, URL S-Control on page 188, and Snippet on page 187.

Tab
An interface item that allows you to navigate around an app. A tab serves as the starting point for viewing, editing, and entering information for a particular object. When you click a tab at the top of the page, the corresponding tab home page for that object appears.

URL (Uniform Resource Locator)
The global address of a website, document, or other resource on the Internet. For example, http://www.salesforce.com.

URL S-Control
An S-Control that contains an external URL that hosts the HTML that should be rendered on a page. When saved this way, the HTML is ultimately hosted by an external web site. See also S-Control on page 188.

Web service
A mechanism by which two applications can easily exchange data over the Internet, even if they run on different platforms, are written in different languages, or are geographically remote from each other.

Web tab
A custom tab that allows your users to use your Web applications or other websites from within the application.

Workflow alert
An email that is sent according to a specified email template when a workflow rule is triggered. Unlike workflow tasks, which can only be assigned to application users, workflow alerts can be sent to any user or contact, as long as they have a valid email address.

Workflow field update
The change to the value of a particular field on the record that occurs when a workflow rule is initially triggered.
Workflow rule
A "container" for a set of workflow instructions that includes the criteria for when the workflow should be activated, as well as the particular tasks, alerts, and field updates that should take place when the criteria for that rule are met.

Workflow task
The task that is assigned to an application user when a workflow rule is triggered.
Index

A
Activities, enabling for custom objects 38
AJAX Toolkit JavaScript library 139
Alerts, workflow
about 117
Anonymous blocks, Apex 177
Apex
anonymous blocks 177
ANTLR 151
AppExchange API calls 177
arrays 156
arrays as maps 157
blocks 162
case sensitivity 158
debugging 181
defining new API calls 178
definition 151
exceptions
DML 167
handling uncaught 182
runtime 168
user-defined exceptions 167
execution model and governors 168
expressions 158
inline queries 160
introduction 149
local variables and expressions 154
namespacing 179
operators 159
packages 178
packages, as WSDLs 178
primitive data types 155
runtime logic 151
sample schema 152
sample script 152
SObject abstract type 155
statements
assignment 162
conditional 163
DML 165
for query loops 164
loops 164
procedure 166
query assignment 164
transaction control 166
try and throw 166
syntax and semantics 151
testing 181
Apex (continued)
triggers 180
variable declaration 154
AppExchange API
accessing data 140
Apex calls 177
API 132
authentication 133
common calls 132
defining new calls with Apex 178
hooks and targets 140
initPage() function 139
login() call 133
making a query call 140
merge fields 134, 139
onload event 139
s-controls 135
session ID 133
SOAP protocol 133
Yahoo! Maps 135
AppExchange app menu 34
AppExchange Developer Network
blog 9
Developer Edition, signing up 9
documentation 9
message boards 9
toolkits 9
website 8
AppExchange directory 17
AppExchange platform
AJAX Toolkit JavaScript library 139
collaborative apps 13
composite component 131
data-centric apps 12
forms 12
introduction 11
key elements 12
metadata-driven development 15
multitenancy 14
navigation 12
Setup 32
structured information 12
supporting technologies 13
tabs 12
terminology 183
Web services 16
Arrays, Apex 156
Assignment statements, Apex 162
Attachments, enabling for custom objects 38
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
</tr>
<tr>
<td>AppExchange API 133</td>
</tr>
<tr>
<td>login () call 133</td>
</tr>
<tr>
<td>merge fields 134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks, Apex 162</td>
</tr>
<tr>
<td>anonymous 177</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidate custom object, creating 68</td>
</tr>
<tr>
<td>Candidate Mapping s-control</td>
</tr>
<tr>
<td>adding to page layout 142</td>
</tr>
<tr>
<td>composite component 135</td>
</tr>
<tr>
<td>defining 135</td>
</tr>
<tr>
<td>downloading code 136</td>
</tr>
<tr>
<td>running 142</td>
</tr>
<tr>
<td>Case sensitivity, Apex 158</td>
</tr>
<tr>
<td>Collaborative apps 13</td>
</tr>
<tr>
<td>Components</td>
</tr>
<tr>
<td>composite 21</td>
</tr>
<tr>
<td>native 21</td>
</tr>
<tr>
<td>Composite components</td>
</tr>
<tr>
<td>about 21</td>
</tr>
<tr>
<td>building 131</td>
</tr>
<tr>
<td>Candidate Mapping s-control 135</td>
</tr>
<tr>
<td>map 23</td>
</tr>
<tr>
<td>Conditional statements, Apex 163</td>
</tr>
<tr>
<td>Controlling fields 52</td>
</tr>
<tr>
<td>Create New drop-down list 41</td>
</tr>
<tr>
<td>Custom app wizard 33</td>
</tr>
<tr>
<td>Custom HTML s-control 135</td>
</tr>
<tr>
<td>Custom link</td>
</tr>
<tr>
<td>adding to page layout 142</td>
</tr>
<tr>
<td>creating 141</td>
</tr>
<tr>
<td>Custom object detail page, related lists 42</td>
</tr>
<tr>
<td>Custom object list page 42</td>
</tr>
<tr>
<td>Custom objects</td>
</tr>
<tr>
<td>definition 26</td>
</tr>
<tr>
<td>relationships, about 66</td>
</tr>
<tr>
<td>Custom tab wizard 38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Management Language (DML) 151</td>
</tr>
<tr>
<td>Data types, Apex 155</td>
</tr>
<tr>
<td>Data-centric apps 12</td>
</tr>
<tr>
<td>Database concepts 29</td>
</tr>
<tr>
<td>Debugging, Apex 181</td>
</tr>
<tr>
<td>Defining an app</td>
</tr>
<tr>
<td>Custom app wizard 33</td>
</tr>
<tr>
<td>Recruiting app 33</td>
</tr>
<tr>
<td>Dependent fields 52</td>
</tr>
<tr>
<td>Deployment status 38</td>
</tr>
<tr>
<td>Developer Edition, signing up 9</td>
</tr>
<tr>
<td>DML exceptions, Apex 167</td>
</tr>
<tr>
<td>DML statements, Apex 165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email templates 125</td>
</tr>
<tr>
<td>Entity relationship diagram, Recruiting app 78</td>
</tr>
<tr>
<td>Exceptions, Apex</td>
</tr>
<tr>
<td>DML 167</td>
</tr>
<tr>
<td>handling uncaught 182</td>
</tr>
<tr>
<td>runtime 168</td>
</tr>
<tr>
<td>user-defined 167</td>
</tr>
<tr>
<td>Execution model, Apex 168</td>
</tr>
<tr>
<td>Expressions, Apex 154, 158</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field dependency matrix 53</td>
</tr>
<tr>
<td>Field history, enabling for custom objects 38</td>
</tr>
<tr>
<td>Field updates, workflow</td>
</tr>
<tr>
<td>about 117</td>
</tr>
<tr>
<td>changing record owner 124</td>
</tr>
<tr>
<td>Fields</td>
</tr>
<tr>
<td>advanced 50</td>
</tr>
<tr>
<td>dependencies 52</td>
</tr>
<tr>
<td>formula 55</td>
</tr>
<tr>
<td>introduction 43</td>
</tr>
<tr>
<td>picklists, controlling 52</td>
</tr>
<tr>
<td>picklists, dependent 52</td>
</tr>
<tr>
<td>picklists, multi-select 50</td>
</tr>
<tr>
<td>picklists, standard 50</td>
</tr>
<tr>
<td>for query loops, Apex 164</td>
</tr>
<tr>
<td>Formula fields</td>
</tr>
<tr>
<td>about 55</td>
</tr>
<tr>
<td>calculating days open 56</td>
</tr>
<tr>
<td>calculating salary 55</td>
</tr>
<tr>
<td>custom field wizard 57</td>
</tr>
<tr>
<td>Custom formula field editor 57</td>
</tr>
<tr>
<td>sample formulas 55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Governors, Apex 168</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help and training options 146</td>
</tr>
</tbody>
</table>
Importing data
- download sample data 79
- Import wizard 79
Inline queries, Apex 160

Job application custom object, creating 73

Local variables, Apex 154
- login () call 133
- Lookup relationship field
  - about 66
  - creating 67
Loop statements, Apex 164

Merge fields
- authentication 134
- Position.Id 139

Metadata-driven development 15
Multitenant architecture 14

Namespacing, Apex 179
Native components
- about 21
- custom objects 21
- security and sharing rules 22
- workflow 23
Notes, enabling for custom objects 38

On-demand platform, introduction 8
Operators, Apex 159

Packages, Apex 178
- Page layouts
  - about 59
  - creating a new section 60
  - page layout edit page 59
- Partner program 147
- Picklist fields
  - controlling 52

Picklist fields (continued)
- dependent 52
- multi-select 50
- standard 50

Podcasts 146
Position custom object
- adding fields 44
- creating 36
- defining a tab 39

Primitive data types, Apex 155
Procedure statements, Apex 166
Programmable Web 8

Query assignment statements, Apex 164

Record, definition 27
Recruiting app
- composite component, map 23
- custom objects 21
- design 20
- entity relationship diagram 78
- introduction 19
- security and sharing rules 22
- tabs 22
- workflow 23

Related lists 42
Relationships
- custom object 66
- definition 28
- entity relationship diagram 78
- lookup custom fields 66
- many_to_one 68
- many-to-many 71
- one-to-one 68

Reports, enabling for custom objects 37
Review custom object, creating 76
Rules, workflow
- about 117
- updating fields 124

S-Control Edit page
- attribute entry 136
- code entry 136
S-controls
- Candidate Mapping 135
- create custom link 141
- Custom HTML 135
- downloading Candidate Mapping code 136
Index

S-controls (continued)
- introduction 135
- Snippet 135
- URL 135
Sample formulas 55
Session ID, AppExchange API 133
Sforce Object Query Language (SOQL) 151
Snippet s-control 135
SOAP protocol, Web services 133
SObject abstract type, Apex 155
Statements, Apex 162

T

Tabs
- color scheme 39
- creating Positions tab 39
- icon 39
- introduction 12
- launching custom tab wizard 38
- Recruiting app 22
Tasks, workflow
- about 117
- creating 120
Templates, email 125
Testing, Apex 181
Training options 146
Transaction control statements, Apex 166
Triggers, Apex 180
Try and throw statements, Apex 166

U

Universal Containers, about 20

URL s-control 135
User-defined exceptions, Apex 167

V

Variable declaration, Apex 154

W

Web services
- AJAX Toolkit JavaScript library 139
- API 132
- introduction 16
- SOAP protocol 133
- XML 133
Workflow alerts
- about 117
Workflow field updates
- about 117
Workflow rules
- about 117
- assigning tasks 118
- creating 119
- updating fields 124
Workflow tasks
- about 117
- creating 120
Workflow, definition 116
WSDLs
- Apex packages as 178

Y

Yahoo! Maps, composite component 135