The Pragmatic Programmers

Code in the Cloud

Programming Google AppEngine

Mark C. Chu-Carroll

Edited by Colleen Toporek

What readers are saying about *Code in the Cloud*

This is a great book if you want to learn about the cloud and how to use App Engine. It was nice to see examples in both Python and Java. Mark does an excellent job of explaining the technologies involved in a down-to-earth, less-hype–more-facts way that I found engaging. Very nice read indeed!

► **Fred Daoud**
Author, *Stripes: ...and Java Web Development Is Fun Again* and *Getting Started with Apache Click*

When you think about the distinction between essential and accidental complexity in web application development, chores like acquiring server hardware, installing an operating system, and worrying about how well those infrastructure choices are going to serve your application's needs down the road definitely fall into the latter group. *Code in the Cloud* explains how developers can use the services provided by the Google App Engine platform to write highly flexible and scalable web-based applications without worrying about a lot of the low-level deployment details that have plagued developers in the past.

► **Lyle Johnson**
Senior Analyst, Sentar Inc.

Compact, well-commented code, and clear explanations—what more could a new cloud developer want?

► **Dorothea Salo**
University of Wisconsin–Madison

Code in the Cloud

Programming Google App Engine

Mark C. Chu-Carroll

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://www.pragprog.com.

The team that produced this book includes:

Editor:	Colleen Toporek
Indexing:	Sara Lynn Eastler
Copy edit:	Kim Wimpsett
Production:	Janet Furlow
Customer support:	Ellie Callahan
International:	Juliet Benda

Printed in the United States of America.

ISBN-10: 1-934356-63-8

ISBN-13: 978-1-934356-63-0

Printed on acid-free paper.

P1.0 printing, April, 2011

Version: 2011-4-14

Contents

Part I

Getting Started with Google App Engine

Chapter 1

Introduction

Cloud computing is an innovative and exciting style of programming and using computers. It creates tremendous opportunities for software developers: cloud computing can provide an amazing new platform for building new kinds applications. In this chapter, we'll look at the basic concepts: what cloud computing is, when and why you should use it, and what kinds of cloud-based services are available to you as an application developer.

1.1 What's Cloud Computing?

Before we look at how to write cloud programs with Google App Engine, let's start at the very beginning and ask just what we mean by cloud computing? What is the *cloud*? How is it different from desktop computing or old-fashioned client-server computing? And most importantly, why should you, as a software developer, care about the cloud? When should you use it, and what should you use it for?

The Cloud Concept

In the modern world of the Internet and the World Wide Web, there are thousands upon thousands of computers sitting in data centers, scattered around the world. We use those computers constantly—for chatting with other people, sending email, playing games, and reading and writing blogs. When we're doing one of these everyday activities, we're accessing a program running on a server, using our browser as a client.

But where is the program actually running? Where is the data? Where are the servers? They're somewhere out there, somewhere in some

data center, somewhere in the world. You don't know where, and more importantly, you don't care; there's absolutely no reason for you to care. What matters to you is that you can get to the program and the data whenever you need to.

Let's look at a simple example. A few years ago, I started writing a blog. (The blog has since moved, but it's still a good example.) When I got started, I used Google's Blogger service to write it. Every day, I would open up my web browser, go to http://goodmath.blogspot.com/admin, and start writing. When I finished, I'd click on the Post button, and the blog post would appear to all of my readers. From my point of view, it just worked. All I needed was my web browser and the URL, and I could write my blog.

Behind the scenes, Blogger is a complex piece of software run by Google in one of its data centers. It hosts hundreds of thousands of blogs, and those blogs are read by millions of users every day. When you look at it this way, it's obvious that the software behind Blogger is running on lots of computers. How many? We don't know. In fact, it's probably not even a fixed number—when not many people are accessing it, it doesn't need to be running on as many machines; when more people start using it, it needs more machines. The number of machines running it varies. But from the user's point of view—whether that user is a blog author or a blog reader—none of that matters. Blogger is a service, and it works. When I want to write a post, I can go to Blogger and write it, and when people go to my blog's web page, they can read it.

That's the fundamental idea of the cloud: programs and data are on a computer somewhere out there, and you neither know nor care where that computer is.

Why call this collection of resources a cloud? A cloud is a huge collection of tiny droplets of water. Some of those droplets fall on my yard, providing the trees and the lawn with water; some run off into the reservoir from which my drinking water comes. And the clouds themselves grow from evaporated water, which comes from all over the place. All I want is enough water in my yard to keep the plants alive and enough in the reservoir so that I have something to drink. I don't care *which* cloud brings the rain; it's all the same to me. I don't care where on earth that water came from. It's all just water—the particular drops are pretty much exactly the same, and I can't tell the difference. As long as I get enough, I'm happy.

So think about the various data centers around the world where companies have swarms of computers—as clouds. Lots of the biggest players in network computing, including Google, Amazon, Microsoft, IBM, and Yahoo, all have thousands of machines connected to networks running all sorts of software. Each of those centers is a cloud, and each processor, each disk drive, is a droplet of water in that cloud. In the cloud world, when you write a program, you don't know what computer it's going to run on. You don't know where the disks that store the data are, and you don't need to care. You just need to know how many droplets you need.

Cloud to the Developer

Cloud computing is a fundamental change from how computers and software have worked in the past. Traditionally, if you wanted to run an application, you went out and bought a computer and software, set it up on your own premises, and ran your program. You needed to pick out which operating system you were going to run, handle the installation of your software, and maintain your computer—keeping track of software upgrades, security, backups, and so on.

With cloud computing, you don't do any of that. If you're a user of the cloud, you buy access to the application you want and then connect to it from anywhere. Installing the software, maintaining the hardware and software where the application runs, making sure that the data is kept safe and secure—none of that is your concern. In the cloud, you buy software as a service. If you need more storage than a typical user, you buy extra storage from the service provider. If that means buying and installing a new disk drive, that's up to the provider. You just buy storage-as-a-service from them: how they provide it is their problem. You tell them what you need—in both the physical sense ("I need 1TB of storage.") and in less tangible quality-of-service senses ("I need to guarantee that my storage is transactional, so that after I commit a change, data will never be lost."). You tell them your requirements, and some cloud provider will sell you a service that meets those requirements.

What this means is that when you're developing for the cloud, instead of buying a computer and running software on it, you break things down to basic building blocks, buy those pieces from service providers, and put them together however you want to build a system.

The building blocks are the resources you need to run a program or to perform a task. Resources include things like processing time, network

bandwidth, disk storage, and memory. As a user of the cloud, you don't need to be concerned about where these resources are located. You know what you need, and you buy that from whoever can provide it to you most conveniently.

For developers, cloud computing introduces an even bigger change. When you develop for the cloud, you're not building a piece of software to sell to your customers—you're building a service for your customers to use. Understanding that difference is crucial: you need to design your application around the idea that it's a service you're going to provide to users, not a standalone application that they're going to install on their computers. Your customers are going to choose a service based on the tasks they want to accomplish, so your application needs to be designed with the task in mind, and you must provide it in the most flexible way possible.

For example, if you want to build a to-do list application for a desktop computer, it's a fairly straightforward process. There are lots of variations in how you can arrange the UI, but the basic idea of what you're building is obvious. You would build one UI—after all, why would you need more than one? And you'd build it mainly for a single user. If you are developing this to-do list application for the cloud, though, you'd want multiple UIs: at the very least, you'd want one UI for people accessing your service using their desktop computer and one for people using a mobile browser on a cell phone. You'd probably want to provide an open interface that other people could use for building clients for other devices. And you'd need to design it for multiple users; if you put an application in the cloud, there's only one program, but it can be used by lots of people. So you need to design it around the assumption that even if users never work together using your application, it's still a multi-user system.

For developers, the most exciting aspect of cloud computing is its scalability. When you're developing in the cloud, you can write a simple program to be used by one or two people—and then, without ever changing a line of code, that program can scale up to support millions of users. The program is scale-neutral: you write it so it will work equally well for one dozen users or one million users. As you get more users, all you need to do is buy more resources—and your program will just work. You can start with a simple program running on one server somewhere in the cloud and scale up by adding resources until you've got millions of users.

Cloud versus Client-Server

In many ways, the basic style of development for cloud-based software is similar to programming for client-server computing. Both are based on the idea that you don't really run programs on your own computer. Your computer provides a window into an application, but it doesn't run the application itself. Instead of running the program on your computer, all you do on your own computer is run some kind of user interface. The real program is running somewhere else on a computer called a server. You use the server because, for whatever reason, the resources necessary to run the program aren't available on your local computer: it's cheaper, faster, or more convenient to run the program somewhere else where the necessary resources are easy to obtain.

The big difference between cloud and client-server development is in what you know: in traditional client-server systems, you might have a specific computer that is your server, and that's where your stuff is running. The computer may not be sitting on your desk in front of you, but you know where it is. For example, when I was in college, one of the first big computers I used was a VAX 11/780 nicknamed "Gold." Gold lived in the Rutgers university computing lab in Hill Center. I used Gold pretty much daily for at least a year before I actually got to see it. The data center had at least thirty other computers: several DEC 20s, a couple of Pyramids, an S/390, and a bunch of Suns. But of those machines, I specifically used Gold. Every program that I wrote, I wrote specifically to run on Gold, and that's the only place that I *could* run it.

In the cloud, you aren't confined to a specific server. You have computing resources—that is, someone is renting you a certain amount of computation on some collection of computers somewhere. You don't know where they are; you don't know what kind of computers they are. You could have two massive machines with 32 processors each and 64 gigabytes of memory; or they could be 64 dinky little single-processor machines with 2 gigabytes of memory. The computers where you run your program could have great big disks of their own, or they could be diskless machines accessing storage on dedicated storage servers. To you, as a user of the cloud, that doesn't matter. You've got the resources you pay for, and where they are makes no difference as long as you get what you need.

When to Develop for the Cloud

So now you know what the cloud is. It's a revolutionary way of thinking about computing; it's a universe of servers that you can build an application on; it's a world of services that you can build or that you can use to build other things. Now, the question is, when should you use it?

You can write almost any application you want in the cloud. In fact, many people strongly believe that everything should be in the cloud—that there's no longer any reason to develop applications for standalone personal computers. I don't go quite that far: many applications are well suited to the cloud, but that doesn't mean that it's the ideal platform for everything. You can build anything as a service in the cloud, but it might be a lot harder than developing it as a standalone application.

There are three kinds of applications that it makes sense to build in the cloud:

Collaborative applications.
: If the application you're building will be used by groups of people to work together, share data, communicate, or collaborate, then you really should build that application in the cloud. Collaboration is the cloud's natural niche.

Services.
: If you ask, "What does my application do?" and the most natural answer sounds like a service, then you're looking at a cloud application. The difference between an application and a service can be subtle—you can describe almost anything as a service. The key question here is what's the most natural description of it? If you want to describe the desktop iTunes application, you could say, "It lets people manage their music collections," which does sound service-like. But it leaves out the key property of what the iTunes desktop application does: it manages a collection of music files on the users' computers and lets them sync that music with their iPods using a serial cable. Described the latter way, it's clear that it's a desktop application, not a cloud application.

  On the other hand, if you take a look at something like eMusic, you'll come to a different conclusion. eMusic is a subscription-based website that lets users browse an enormous collection of music and buy a certain number of songs per month. eMusic is clearly a service: it lets people search through a library of hundreds of thousands of musical tracks, providing them with the

ability to listen to snippets, read reviews, comment on things that they've listened to, get suggestions for new things based on what they like, and ultimately select things to purchase. That's clearly a service, and it makes sense to keep it in the cloud.

Large computations.
Is your application intended to perform a massive computation, which you could never afford to do if you needed to buy your own computers to run it? If so, the cloud allows you to purchase time on a server farm of computers in an affordable way and run your application. This is great for people like genetics researchers, who need to run massive computations but don't have the money or other resources to set up a dedicated data center for their computations. Instead, they can purchase time on commercial data centers, which they share with many other users.

1.2 Cloud Computing Programming Systems

There are multiple ways of programming the cloud. Before we start actually writing programs, we'll take a quick look at a few examples to give you a sense of what sorts of options are available.

Amazon EC2

Amazon provides a variety of cloud-based services. Their main programming tool is called EC2, Elastic Computing Cloud.

EC2 is really a family of related services. Compared to App Engine, which provides a single, narrowly focused suite of APIs, EC2 is completely agnostic about programming APIs. It provides hundreds of different environments: you can run your application in EC2 using Linux, Solaris, or Windows Server; you can store data using DB2, Informix, MySQL, SQL Server, or Oracle; you can implement your code in Perl, Python, Ruby, Java, C++, or C#; you can run it using IBM's WebSphere or sMash, Apache JBoss, Oracle WebLogic, or Microsoft IIS. Depending on which combination you prefer and how much of each kind of resource (storage, CPU, network bandwidth) you plan to use, the costs vary from $0.10 per CPU hour and $0.10 per gigabyte of bandwidth to around $0.74 per CPU hour for high-end instances.

Amazon S3

Amazon provides another extremely interesting cloud service, which is very different from most other cloud offerings. S3, Simple Storage Service, is a pure storage system. It doesn't provide the ability to run programs; it doesn't provide any filesystem; it doesn't provide any indexing. It's pure block storage: you can allocate a chunk of storage that has a unique identifier, and then you can read and write bytes from that chunk using its identifier.

A variety of systems have been created that use S3 for storage: web-based filesystems, native OS filesystems, database systems, and table storage systems. It's a wonderful example of the cloud's resource-based paradigm: the computation involved in storage is completely separated from the actual data storage itself. When you need storage, you buy a bunch of bytes of storage space from S3. When you need computation, you buy EC2 resources.

S3 is a really fascinating system. It's very focused: it does exactly one thing and does it in an incredibly narrow way. But in an important sense, that's exactly what the cloud is about. S3 is a perfectly focused service; it stores bytes for you.

S3 charges are based on two criteria: how much data you store and how much network bandwidth you use storing and retrieving your data. Amazon currently charges $0.15 per gigabyte per month and about $0.10 per gigabyte uploaded and $0.17 per gigabyte downloaded.

On a related note, Google provides a very similar cloud service, called Google Developer Storage, which replicates the basic features of S3 in the Google cloud.

IBM Computing on Demand

IBM provides a cloud service platform based on IBM's suite of web service development that uses WebSphere, DB2, and Lotus collaboration tools. The environment is the same as the IBM-based environment on EC2, but it runs in IBM's data centers instead of Amazon's.

Microsoft Azure

Microsoft has developed and deployed a cloud platform called Azure. Azure is a Windows-based platform that uses a combination of standard web services technologies (such as SOAP, REST, Servlets, and ASPs) and Microsoft's proprietary APIs, like Silverlight. As a result,

you get the ability to create extremely powerful applications that look very much like standard desktop applications. But the downside is it's closely tied to the Windows platform, so the application clients run primarily on Windows. While there are Silverlight implementations for other platforms, the applications tend to only be reliable on Windows platforms and only fully functional in Internet Explorer.

So that's the cloud. Now that we know what it is, we're going to start learning about how to build applications in the cloud. Google has put together a really terrific platform, called App Engine, for you to build and run your own cloud applications.

In the rest of the book, we're going to look in detail at the key pieces of building cloud-based web applications. We'll start off working in Python. Python's great for learning the basics: it lets you see what's going on, and it makes it easy to quickly try different approaches and see what happens.

We'll go through the full stack of techniques that you need for building a Google App Engine application in Python, starting with the basic building blocks: HTTP, services, and handlers. Then we'll look at how you work with persistent data in the cloud using the App Engine datastore service. And then, we'll look at how to build user interfaces for your applications using HTTP, CSS, and AJAX.

From there, we'll leave Python for a while and move into Java. In my opinion, Java can be a lot more convenient for building complex applications. Not that Python can't or shouldn't be used for advanced App Engine development, but my preference is to use Java. And App Engine provides access to an absolutely brilliant framework called GWT, which abstracts away most of the boilerplate plumbing of a web-based cloud application, allowing you to focus on the interesting parts. We'll spend some time learning about how to build beautiful user interfaces using GWT and how to do AJAX-style communication using GWT's remote procedure call service.

Finally, we'll spend some time looking at the most complicated aspects of real web development. We'll look at the details of how you can do sophisticated things using the App Engine datastore service, how to implement server-side processing and computation using things like cron, and how to integrate security and authentication into your App Engine application.

In the next chapter, we'll start this journey through App Engine by looking at how to set up an App Engine account. Then we'll look at how to set up the software on your computer for building, testing, and deploying App Engine applications written in Python.

1.3 Acknowledgments

Writing a book is a long, difficult process, and there's no way that anyone can do it alone. Getting this book done took a lot of effort from a lot of people.

I'd like to thank

- the technical reviewers, Nick Johnson, Scott Davis, Fred Daoud, Lyle Johnson, Krishna Sankar, and Dorothea Salo, for their input and feedback;
- my editor, Colleen Toporek, who put up with my endless delays, writer's block, and god-awful spelling and who kept the book on track;
- the App Engine team at Google for building such an amazing system for me to write about; and, of course,
- my wife and my thoroughly evil children for dealing with me while I spent hours at the keyboard working.

Chapter 2

Getting Started

In this chapter, we're going to take our first look at Google App Engine and get started using it. You'll learn how to do the following:

1. Set up a Google App Engine account.
2. Download and set up the Google App Engine SDK.
3. Create a simple Google App Engine application.
4. Test an application locally.
5. Deploy and monitor a Google App Engine application in the cloud.

This isn't going to be the most exciting chapter in the book, but it's stuff that you need to get out of the way in order to be able to get to the interesting stuff. And there will be an interesting tidbit or two.

2.1 Setting Up a Google App Engine Account

The first thing you need to do in order to write cloud applications with Google App Engine is open an App Engine account. When you're developing for the cloud, you're renting computing and storage resources for your application. The App Engine account provides you with a basic set of free resources and a mechanism for buying more of various types of resources when you need them.

Creating an account with Google App Engine is free. A basic, no-charge App Engine account gives you the ability to run up to ten applications, along with these features:

- 6.5 hours of CPU time per day
- 10 gigabytes per day each of outgoing and incoming bandwidth
- 1 gigabyte of data storage
- Privileges to send 2,000 email messages per day

Counting CPU Time

You get 6.5 hours of free CPU time per day. But as you work, if you buy CPU time, you might end up using much more than that, even more than 24 hours of CPU time in a day. In Google App Engine, your application isn't running on one server; it's running in a Google data center. Each incoming request is routed to some machine in the cluster. There can be multiple users accessing your system at the same time and therefore using CPU time on multiple physical computers. What you're billed for is the total amount of CPU time used by your application on all of the computers that wind up running any part of it. So you can end up using more than 24 hours of CPU time per day.

If you need more, you can buy additional resources in each category.

To get a Google App Engine account, you first need to have a standard Google account. If you already use Gmail or iGoogle, you've got one. If not, just go to Google.com, select Sign In from the top right corner of the screen, and click on the Create an Account Now link.

Once your Google account is ready, you can get started with Google App Engine by pointing your browser at http://appengine.google.com. You'll see a standard Google login screen; go ahead and log in with your Google username and password. The first time you do this, you'll need to authenticate yourself using SMS messaging with your cell phone. In order to prevent spammers from setting up App Engine accounts, Google set up a mechanism that requires a unique telephone number. Don't fool around here: you can only use a given phone number to set up one App Engine account—once that number is used, you can't create another account using that number again.

After you fill out the form, you'll get a new page in your browser that asks you to enter an authentication code. Within ten minutes, you'll receive an SMS message with an authentication code on your cellphone. Enter that code, and you're ready to go.

2.2 Setting Up Your Development Environment

Now that you have a Google App Engine account, the first thing you'll want to do is create an application. Already the process of developing for the cloud is a bit different from normal application development. To write a new program to run on your own computer, you'd just open up an editor and start typing. For a cloud app, you need to register your application on a cloud server in order to create a space for it to run and to provide you with the tools that you'll need to work on it.

Before you download the Google App Engine tools, make sure that you have Python installed on your machine. Python is in a state of flux right now, transitioning into a significantly rewritten version of the language. As a result, there are several incompatible versions of Python that are in common use. For App Engine, you'll need to use Python 2.5. So make sure that you've got the right version of Python installed. Installing Python on the different operating systems that you can use for developing App Engine services is beyond the scope of this section, but if you go to the main Python homepage at http://python.org, you can find up-to-date installation instructions.

You'll also need a text editor or IDE to use for writing code. There are plenty of excellent examples of free tools; just pick one that you're comfortable with, and make sure you have it installed.

When you have the tools you need to write Python programs, you can download the Google App Engine Python SDK by logging into the App Engine account you created in the previous section and clicking Create an Application. This brings you to a form to give your application a name and a description. The form will look roughly like the one in Figure 2.1, on the following page. (App Engine is updated frequently, so the exact form may appear slightly different.)

To create your application, you need to provide some information to the Google App Engine service:

An application identifier.

This is a unique name for your application, distinct from every other application being run by any other App Engine user. It will be used to form the URL for your application. This is the one thing about your application that you *cannot* change, so choose carefully! You can type in a name and check to make sure that no one else has already used it by clicking the Check Availability button. I recommend choosing a personal prefix for your application name;

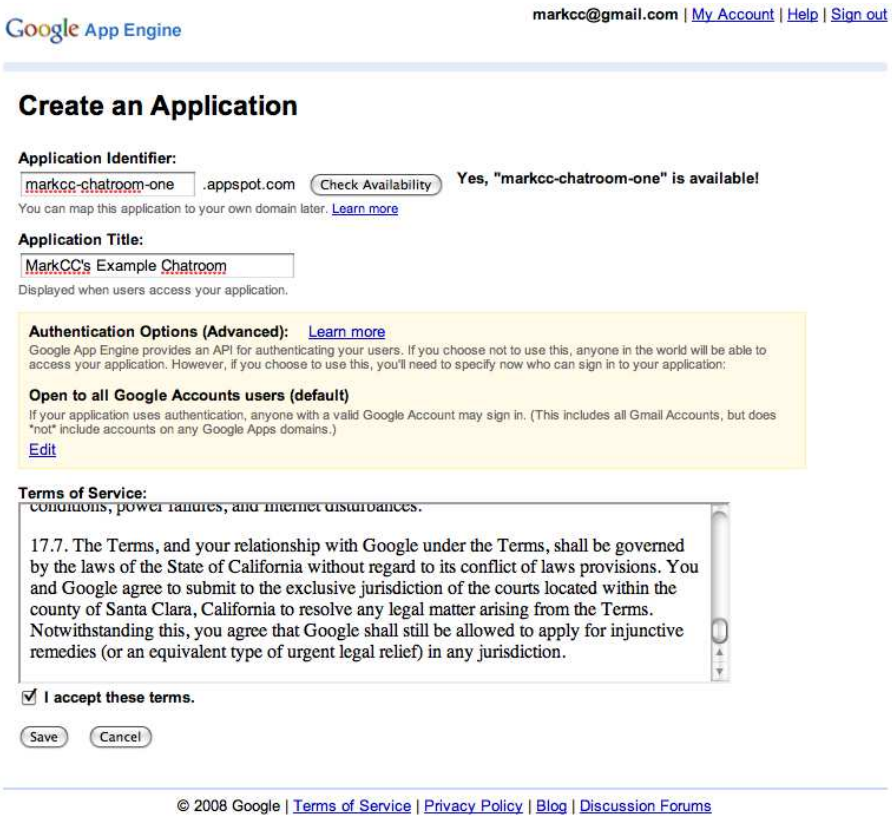


Figure 2.1: The Create an Application form

doing so makes it more likely that you'll avoid name collisions with anyone else, and it gives your family of applications a common identity within the universe of App Engine programs. In all of the applications that I built for this book, I used the prefix markcc. For the sample application that we're going to walk through, I chose the name markcc-chatroom-one, so the URL for my application is going to be http://markcc-chatroom-one.appspot.com.

An application title.

This is the name for your application that all the users of your application will see and that will appear on your login page. For the example, I used MarkCC's Example Chatroom. You can change the application title from the control panel any time you want.

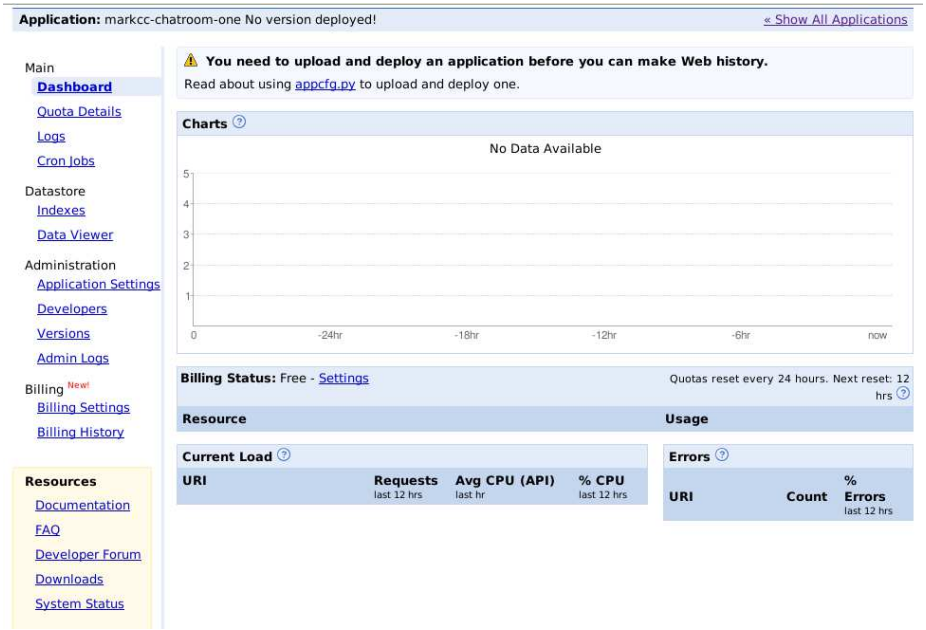


Figure 2.2: The Google App Engine control panel

Security and authentication settings.

You can set initial security and authentication settings for your application. Don't worry about this for now; we'll come back to that in Chapter 17, *Security in App Engine Services*, on page 260.

Terms of service.

Before you can create an application on Google App Engine, you have to accept Google's terms of service. Take the time to read this through so you understand the commitments you're making and the guarantees Google is giving to you as an Google App Engine developer. Then click the check box to indicate that you accept the terms.

When you're done filling out the form, click Save, and the framework of your brand-new application will be created by Google App Engine. After you've saved your initial application description, you'll get a control panel that you'll use for building and monitoring your application (Figure 2.2).

Once you're sitting in front of the application control panel, you're just about ready to start programming. Notice, though, that programming for the cloud is different from other kinds of development. You can't edit files on the Google App Engine server. You need to write them locally and then use an administrative script to transfer them into the App Engine environment, where they can run.

The next step is to get the tools. On your control panel, toward the bottom left, you'll see a box labeled Resources. This box contains links to software, forums, and documentation that you'll want as you learn and use Google App Engine. For right now, click on the Download link and download the appropriate version of the App Engine SDK for Python. Once it's downloaded, go ahead and install it. The installation process varies slightly, depending on which operating system you're using: for Windows or Macintosh, the download contains an automated installation program—just run it, and it will do the work. If you're using Linux, the download is a zip file, so unzip it in an appropriate location.

If you're using Windows or MacOS, you're all ready to start. If you're using Linux, take the directory where you unpacked the SDK zip file and add it to your path.

There are two main programs in the SDK that you'll use:

- dev_appserver.py runs a simulated Google App Engine environment that you can use to test your applications on your local computer.
- appcfg.py uploads and configures your applications using Google App Engine in the cloud.

2.3 Starting to Program in Python with Google App Engine

Now we're ready to do some programming!

Python App Engine is, at its core, very simple. The main engine is a lightweight, secure CGI executor. CGI is one of the oldest interfaces for executing programs in response to HTTP requests. The bare bones of Google App Engine are pure CGI. The big advantage of this is that if you've ever done any CGI scripting in Python, you can just about take those scripts and use them in Google App Engine. Any framework and any Python library that was written for CGI scripting can be used in Google App Engine—just include the framework/library files when you upload your application code.

Why Start with Python?

I'll say more about this later, but I'm not a huge Python fan. We're starting with it first for a few reasons.

First, Python is a very pleasant language that makes it possible to do a lot with a small amount of code. We can start writing Google App Engine programs with just a few lines of code. There's very little in the way of required infrastructure. When you're learning how to develop for the cloud, Python is great way to start.

Second, my tastes shouldn't dictate how *you* should build your application. Python is a very powerful, flexible language, and it's got excellent support in Google App Engine. If you're a Python aficionado, then after reading this book, you should be able to build your App Engine applications in Python.

Third, we're going to learn tools like GWT, which generate a lot of code for us, taking care of the underlying mechanisms of the client/server interaction in our cloud applications. For developing complicated applications, that saves us incredible amounts of effort. But it's important to understand what's happening behind the scenes.

Python gives us a good way of exploring the primitive infrastructure of a cloud application. We'll be able to look at each piece of technology, build it up, and learn about how it works. When we get to GWT, it will be easy to understand what's really going on.

If you decide you like doing your cloud programming in Python, you'll learn enough to be able to do it. But even if you never write a cloud app using Python, taking the time to explore the basic technologies of cloud applications using Python will help you understand and debug your real applications in whatever language you decide to use.

The easiest way to work with Google App Engine is to use its own framework, called webapp. webapp is a very elegant and powerful framework that is really easy and pleasant to work with.

In addition, Google App Engine provides you with access to Google services like login, data storage, security, authentication, and payments inside your application. In this book, we'll focus on using the webapp framework—but once you know how to work with and execute your programs using App Engine, you'll be able to write App Engine applications that use these services using other frameworks as well.

In most of this book, we'll be working on a chat room application. But before we get to that, we'll do the cloud equivalent of "Hello, World." For the cloud, it's a simple program that runs on a cloud server and generates a welcome page to be displayed in the user's browser.

Since cloud applications typically use web browsers as their user interface, our application must generate HTML rather than just plain text. Whenever we generate output, we first need to include a MIME header, which is a single line that specifies the format of the content that follows. For HTML, the content type is text/html.

The Google App Engine SDK expects you to have all of your application's files stored in a directory hierarchy. For our first program, we'll create a directory named chatone. In that directory, we'll write our trivial welcome program in a file named chat.py:

Download chatone/chat.py

```
import datetime

print 'Content-Type: text/html'
print ''
print '<html>'
print '<head>'
print '<title>Welcome to MarkCC\'s chat service</title>'
print '</head>'
print '<body>'
print '<h1>Welcome to MarkCC\'s chat service</h1>'
print ''
print 'Current time is: %s' % (datetime.datetime.now())
print '</body>'
print '</html>'
```

To be able to run the program, we need to tell Google App Engine what language it's written in, what resources it needs, where the code is, and how to map requests that are sent to the server on to the code.

In Google App Engine, we do that by writing a file named app.yaml:

Download chatone/app.yaml

```
application: markcc-chatroom-one
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: chat.py
```

We always start app.yaml files with a header, whose fields are as follows:

application:
: The name of the application we're building. This must exactly match the name you supplied when you created the application control panel.

version:
: A string specifying the version of our application. This is really for our information so that we can do things like query the server to find out what version it's running or identify the code version that caused some bug. This can be any string identifier that you want.

runtime:
: The language in which we're going to write the program: either Java or Python.

After the header, we need to write a list of handler clauses. We use these to describe to the Google App Engine server what it should do when it receives an incoming HTTP request. The server is going to route HTTP requests to scripts that we write. The handler clauses in the app.yaml file are how we tell it which requests to route to which Python scripts. In each handler clause, we specify a **url** pattern using a regular expression, followed by a clause that specifies the action to take when a request matching the pattern is received. For our example, there's only one handler. *Any* request that reaches our application will be answered by running our welcome script, so the **url** pattern is /.*, which will match any request. Whenever a request is received, we want to run our welcome script. So the action is this: script: chat.py, meaning "execute the script named chat.py."

To test our application, we'll first run it locally using dev_appserver.py:

```
$ ls
chatone
```

```
$ ls chatone
app.yaml  chat.py
$ dev_appserver.py chatone
INFO     2009-06-18 23:13:31,872 appengine_rpc.py:157] Server: appengine.google.com
INFO     2009-06-18 23:13:31,880 appcfg.py:320] Checking for updates to the SDK.
INFO     2009-06-18 23:13:31,994 appcfg.py:334] The SDK is up to date.
WARNING  2009-06-18 23:13:31,994 datastore_file_stub.py:404] Could not read \
             datastore data from /tmp/dev_appserver.datastore
WARNING  2009-06-18 23:13:31,994 datastore_file_stub.py:404] Could not read \
             datastore data from /tmp/dev_appserver.datastore.history
INFO     2009-06-18 23:13:32,058 dev_appserver_main.py:463] Running application\
             markcc-chat-one on port 8080: http://localhost:8080
```

With the application running locally, we can test it using our web browser. Look at the last line of output from running dev_appserver.py—it provides the URL for this session—in this case, http://localhost:8080. If we point the browser at that URL, we get something like this:

Welcome to MarkCC's chat service

Current time is: 2009-06-18 22:30:47.345731

Since we know that it works, we can now deploy it to the cloud server by running the appcfg.py command. appcfg.py is the main developer's interface to Google App Engine, so it supports a number of different applications. To send code to the server, use its update command:

```
$ appcfg.py update chatone
Scanning files on local disk.
Initiating update.
Cloning 1 application file.
Uploading 1 files.
Deploying new version.
Checking if new version is ready to serve.
Will check again in 1 seconds.
Checking if new version is ready to serve.
Closing update: new version is ready to start serving.
```

And the code is now deployed on the server. You can access it by going to http://your-app-name.appspot.com.

Even in this first example, we can start to see the basic flavor of programming for Google App Engine. We used nothing from the webapp framework, but the basic concept is still there: the app.yaml file specifies how incoming requests are routed to the scripts that make up our program, and the way our program communicates with the user is by generating HTML content that will be rendered in the user's browser.

The problem with the trivial approach is that it does everything manually. It generates the MIME content header and the HTML page structure itself. Doing this work manually is very verbose and extremely error-prone. It gets worse when you start making the application interactive and you need to parse input from the incoming requests. The webapp framework provides infrastructure that takes care of the basic HTTP request/response cycle, parsing the incoming requests, generating the necessary headers, and managing the communication with the web server to send the response. In addition, webapp provides access to a set of template processors that allow you to create skeletons of your responses so that you don't need to output the entire HTML structure yourself. For now, our HTML is straightforward enough that we don't need to use the template capabilities, but we'll look at them in detail in Chapter 6, *Organizing Code: Separating UI and Logic*, on page 70.

Here's a version of our welcome page application using a basic webapp skeleton:

Download chatone/chatonewa.py

```
❶ from google.appengine.ext import webapp
  from google.appengine.ext.webapp.util import run_wsgi_app
  import datetime

❷ class WelcomePage(webapp.RequestHandler):
❸     def get(self):
❹         self.response.headers["Content-Type"] = "text/html"
❺         self.response.out.write(
            """<html>
                 <head>
                   <title>Welcome to MarkCC's chat service</title>
                 </head>
                 <body>
                   <h1>Welcome to MarkCC's chat service</h1>
                   <p> The current time is: %s</p>
                 </body>
               </html>
            """ % (datetime.datetime.now()))


❻ chatapp = webapp.WSGIApplication([('/', WelcomePage)])

❼ def main():
      run_wsgi_app(chatapp)

  if __name__ == "__main__":
      main()
```

Let's take a quick walk-through to see what the webapp pieces mean:

❶ First, we need to import the pieces of the webapp framework that we're going to use. For this, we're using two basic webapp building blocks: the webapp module itself and a webapp function called run_wsgi_app.

❷ Next, we create a webapp RequestHandler. webapp understands how HTTP works and provides utility classes for working with all of the basic elements of the HTTP protocol. The basic operation in a cloud application is responding to requests from a user. The RequestHandler class is built for doing that—it's the webapp class that you'll use the most.

❸ The only kind of HTTP request that we're going to handle in the welcome application is **GET**. In webapp, you handle that by providing an implementation of the get() method in a subclass of RequestHandler.

❹ Instead of generating things such as the MIME header manually, webapp provides you with a response structure, which includes a Python map. To set the value of any HTTP header, you assign a value into that map. The only header that we're going to use is Content-Type, so we put that into the map.

❺ Instead of generating output directly to standard out, webapp provides you with a buffered output channel that it will manage. You write the content of your response to that output channel. This is the most common error made by new App Engine developers: they use plain old print statements to print out HTML, and then they wonder why they're not seeing the result. You need to make sure that you *always* use the webapp output channel.

❻ To use our webapp RequestHandler, we need to create an application object. An application object looks a lot like an app.yaml file: the app.yaml file describes how to map incoming requests onto particular application scripts, and the application object describes how to take the requests that were mapped to this script and map them to specific RequestHandler classes. We map requests for the root URL—that is, requests to http://markcc-chatroom-one/—to our welcome request handler.

❼ The rest of the file makes use of a common Google idiom. To actually run the application, the script needs to invoke run_wsgi_app. But instead of executing that statement directly, we implement a main function and use an indirect way of invoking it. This idiom

makes it easy to reuse this script. If we just executed run_wsgi_app directly, then if we were to import this script into any other Python code, that line would be executed. The main function plus the conditional invocation ensures that we'll only execute it if we specifically ran this script—it won't be executed by modules that import this.

We can deploy the new version of this exactly the same way as the last one. I changed the filename of the Python script, so I needed to also update the app.yaml file to reference the new Python script file. Once that's done, deployment is done by just running appcfg.py update chatone. That will upload the changes. Immediately after running the update, go to the application URL, which will run the new version of the code. It will still appear in Google App Engine as version 1, because I didn't change the version identifier in app.yaml.

This quick and dirty explanation just gives you a sense of what developing for Google App Engine is like. In later chapters, we'll look at all of the aspects of developing interesting web applications in App Engine in greater detail.

2.4 Monitoring Your Application

As the developer of an application, you'll want to monitor it—to be able to detect how many users you have, the quantity of resources being consumed, the number of hits, which scripts are most active, what's working, what isn't, and so on. You can do that by using the Google App Engine control panel.

The monitoring features of the Google App Engine control panel is going to be really important for you. Your cloud application is running on a computer that you don't control. You can't just run a profiler or a debugging to try to see why things aren't working the way you expect. So when you find that something is taking much longer than you think it should (and that's something that *will* eventually happen if you write any serious cloud applications), the monitor is your information source.

As an example, in my day job, I build cloud-based data analysis applications for Google. I've had cases where an analysis that usually runs for one hour suddenly started taking eight hours. That's obviously a really bad thing. I was able to figure out what was going on by going into the control panel, looking at the information it could give me, and

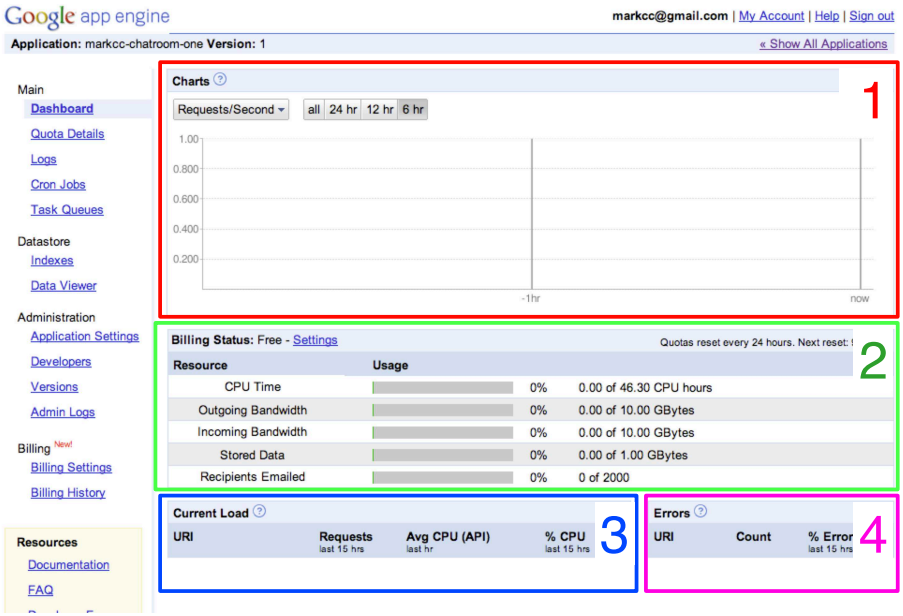


Figure 2.3: The sections of the control panel

discovering that certain shards of the data being processed by the system were extremely large—far larger than normal. That led me to the problem: someone else had changed the code that feeds data into my analyzer, and it was generating data in a form that affected the performance of my code. It would have been incredibly difficult to figure that out without a tool to let me see what was going on in the cluster of server machines.

You can monitor all aspects of your application— from information logged by your code while it was running, to information recorded by the Google App Engine servers, to detailed information about the resources that your application has consumed—in the control panel.

The main view in the application control panel is called the dashboard. The dashboard provides a broad summary of everything you want to know about the status and resource usage of your application, with links to the places where you can get more detailed information. It's divided into four sections, shown in Figure 2.3:

1. A Graph view of the performance of your application. This can be used to display the way that your application has been using resources over time.
2. A Billing Status view, which shows you how much of each resource category you've used and how much you have available under your billing plan;
3. A Load view, which shows the different URI patterns declared in your app.yaml file and how much CPU time has been spent responding to requests for each.
4. An Error view, which shows basic summary information about any errors that occurred in your application.

Finally, the control panel provides a set of useful links, listed under Resources. The resources include developer forums, where you can discuss issues with other Google App Engine developers and Google's App Engine team; the official, up-to-the-minute App Engine documentation; and answers to frequently asked questions. I strongly advise you to make use of those links, particularly the developer forums. Most problems that you encounter developing App Engine programs will be similar to things that other App Engine developers have encountered, and the forums will have the answers that you need.

In addition to the basic resource information, there's a set of administration links on the left side of the control panel main view, which allows you to pick various aspects of the system to use to look at in greater detail.

One example of the useful data provided by the detailed links is information from your application's logs. Every request that is received by the Google App Engine server generates a data record describing the request. In addition, errors are logged here. When your application throws exceptions, they aren't shown to users. The only place that you'll see them is here in the logs. In addition, you can also add logging statements to your program. For now, we'll just look at request logging.

Open the application's control panel. The left side of the screen contains a collection of links to various views and tools for monitoring, managing, and administering your application. In the topmost section, labeled Main, the third link from the top is Logs. Clicking on the Logs link changes the view to show you information from the logs. At this point, if you didn't make any mistakes, the view should be pretty much

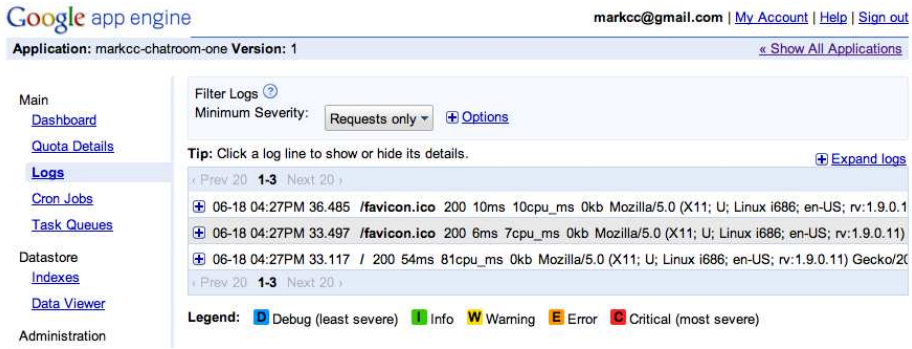


Figure 2.4: The request log view

empty—the default view of the logs is a compact view of any errors that might have occurred in your application.

To see the set of requests that have been processed by the application, use the drop-box menu at the top of the log list, labeled Minimum Severity. Open the menu, and select Requests Only. The result, as shown in Figure 2.4, is a list of the log entries for each of the requests received by the application.

Now you're all set up. You have your Google App Engine account, and the App Engine tools are installed. You're ready to build an App Engine application. In the next chapter, we'll start building a real chat application that runs in the cloud using the App Engine tools that we just finished setting up.

References and Resources

Google App Engine Developers Guide. . .
. . . http://code.google.com/appengine/docs/
The official Google App Engine documentation for both the Python and Java APIs.

Common Gateway Interface (CGI) http://www.w3.org/CGI/
The official standard and documentation for CGI.

Part II

Programming Google App Engine with Python

Chapter 3

A First Real Cloud Application

In this chapter we're going to build our first nontrivial cloud application: a basic Python chat room. Along the way, we'll look at these factors:

- The HTTP protocol used by cloud applications and how cloud applications communicate
- How to take a normal, non–cloud program written in Python and wrap it in an HTTP framework so that it works in the cloud
- How managing data and variables is different in a cloud application

3.1 The Basic Chat Application

As a running example, we're going to build a chat service in Python. Chat is a good example because it's familiar—we've all used chat services. But even though it's an old, familiar sort of application, it's got many of the typical attributes of a cloud service.

What makes a cloud app different and interesting is that cloud applications are intrinsically multiuser. You can't build a cloud application without thinking about how you're going to handle multiple users and how you're going to handle data for multiple users.

A chat application is simple but pretty typical. To build a chat, we need to think about interactions between multiple users, we need to store and retrieve persistent data, and we'll have multiple streams of data for different discussions. On top of that, it's easy to build a simple version and then gradually add features that demonstrate more and more of the capabilities of App Engine.

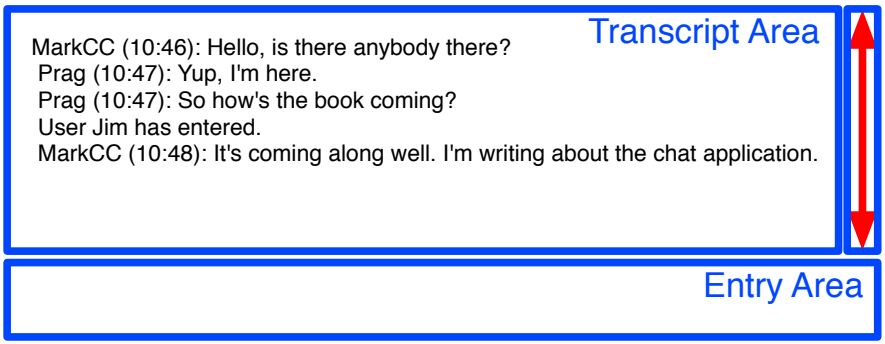


Figure 3.1: A mockup of the chat interface

For now, we'll ignore the user interface and focus our attention on the back end. (We'll deal with the user interface in Chapter 7, *Making the UI Pretty: Templates and CSS*, on page 87.) For now, we're just going to write a basic back end for a traditional local chat application, with the calls to hook a user interface onto it. We're not going to implement the entire thing in this chapter. Instead, our goal for most of this book will be to get to the point where we can do everything that's in this code.

A basic chat application is not very complicated. Imagine a typical chat. The user interface of the chat is pretty straightforward—it should have two boxes, one where you can see the transcript and one where you enter new text. The transcript should have the sequence of messages that have been sent in the chat, each marked by the name of the sender and the time that it was sent. The basic chat interface, then, should look something like the mockup in Figure 3.1.

Now that we have some idea of what it should look like, we can start thinking about how to build it. Before we get into how to build this as a cloud application, let's start by thinking about how to build the back end of a chat application as a standard server program. So we'll work on an application skeleton that does everything we need for chat in standard Python without using any App Engine code at all.

What do we need? Looking at the basic transcript, we can see that the chat system has a virtual space, which users can enter and leave. After they've entered, they can send messages. Any message that gets sent will be visible to everyone who has entered. That gives us the three

basic objects that we want to work with: the space (which we'll call a room), users, and messages.

We want multiple conversations to be available in our space, so people can decide what they want to talk about and with whom. We'll call each space that a conversation can happen in a *room*. There are three interesting things that can happen in a room: someone can enter, someone can leave, and someone can send a message. Also, to keep things simple, instead of automatically updating everyone whenever a message gets sent, we'll just provide a way for users to periodically ask for a transcript of what's been going on in the room. A simple implementation of a room is shown below. It's *not* written the way that you'd write it for the cloud: cloud apps behave very differently, so they need to be implemented differently. In the rest of this book, we'll build up to the point where we have an App Engine cloud app that does everything that this one does but in the cloud way. You'll see how different they are.

Download basechat.py

```
class ChatRoom(object):
    """A chatroom"""

    rooms = {}

    def __init__(self, name):
        self.name = name
        self.users = []
        self.messages = []
        ChatRoom.rooms[name] = self

    def addSubscriber(self, subscriber):
        self.users.append(subscriber)
        subscriber.sendMessage(self.name, "User %s has entered." %
                               subscriber.username)

    def removeSubscriber(self, subscriber):
        if subscriber in self.users:
            subscriber.sendMessage(self.name,
                                   "User %s is leaving." %
                                   subscriber.username)
            self.users.remove(subscriber)

    def addMessage(self, msg):
        self.messages.append(msg)

    def printMessages(self, out):
        print >>out, "Chat Transcript for: %s" % self.name
        for i in self.messages:
            print >>out, i
```

The next thing the chat room needs is users. A user has a name and is signed in to some set of rooms. A user can enter a room, leave a room, or send a message. If they haven't entered a particular room, they're not allowed to send a message to that room.

Download basechat.py

```
class ChatUser(object):
    """A user participating in chats"""
    def __init__(self, username):
        self.username = username
        self.rooms = {}

    def subscribe(self, roomname):
        if roomname in ChatRoom.rooms:
            room = ChatRoom.rooms[roomname]
            self.rooms[roomname] = room
            room.addSubscriber(self)
        else:
            raise ChatError("No such room %s" % roomname)

    def sendMessage(self, roomname, text):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            cm = ChatMessage(self, text)
            room.addMessage(cm)
        else:
            raise ChatError("User %s not subscribed to chat %s" %
                            (self.username, roomname))

    def displayChat(self, roomname, out):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            room.printMessages(out)
        else:
            raise ChatError("User %s not subscribed to chat %s" %
                            (self.username, roomname))
```

The chat message is the simplest part: it's just a message from a user. The message needs a reference to the user who sent it and to the text that the user wants to send, and it needs to include the time the message was sent.

Download basechat.py

```
class ChatMessage(object):
    """A single message sent by a user to a chatroom."""
    def __init__(self, user, text):
        self.sender = user
        self.msg = text
        self.time = datetime.datetime.now()
```

```
    def __str__(self):
        return "From: %s at %s: %s" % (self.sender.username,
                                       self.time,
                                       self.msg)
```

To test the chat message object, we'll just throw together a quick main program—that is, we'll write the code that actually makes our application do something when it's run. In this case, instead of running the chat interactively, we'll just make it show a demo of what things might look like when the program actually runs. We'll make it create some users, have them subscribe to some chats, and have them send some messages.

Download basechat.py

```
def main():
    room = ChatRoom("Main")
    markcc = ChatUser("MarkCC")
    markcc.subscribe("Main")
    prag = ChatUser("Prag")
    prag.subscribe("Main")

    markcc.sendMessage("Main", "Hello! Is there anybody out there?")
    prag.sendMessage("Main", "Yes, I'm here.")
    markcc.displayChat("Main", sys.stdout)


if __name__ == "__main__":
    main()
```

When we run that, we get the following:

```
Chat Transcript for: Main
From: MarkCC at 2009-06-27 15:10:51.181035: User MarkCC has entered.
From: Prag at 2009-06-27 15:10:51.181194: User Prag has entered.
From: MarkCC at 2009-06-27 15:10:51.181218: Hello! Is there anybody out there?
From: Prag at 2009-06-27 15:10:51.181237: Yes, I'm here.
```

It's far from pretty. Those timestamps are distractingly verbose, and there's no formatting to make the text easier to read, but the basic chat functionality (rooms, subscriptions, users, and messages) are all there.

3.2 The Basics of HTTP

The way that we just designed and implemented the skeleton of a chat room is a reasonable approach for a traditional application. But when you're designing an application to run in the cloud, there's an additional step. In a traditional application, you have to think about how to design the data processing back end of your application, and you have

to design a user interface. When you're programming a cloud application, you still need to do those things, but you also need to design a *protocol* for your application.

In cloud applications, the back end runs on a server or collection of servers somewhere in a data center. The user interface runs in a user's web browser. What the protocol does is describe just how the back end and the front end communicate in order to produce a working application that looks like it's running inside the user's browser.

Most cloud applications, and pretty much all App Engine applications, are built using a basic protocol called HTTP (Hypertext Transfer Protocol). For your cloud applications, you need to design a protocol that can be layered on top of HTTP. Layering just means that the protocol is built so that each interaction in your application protocol is described in terms of HTTP interactions. That's one of the key things that makes programming for the cloud so different: cloud application are built around HTTP interactions between a client and a server: layering your application properly onto HTTP is *the* key to building an attractive cloud application that will provide a good user experience. HTTP can seem a bit clunky when you're not used to it, but a bit later in the book you'll see how you can build just about any kind of application using HTTP interactions.

You may already be familiar with HTTP, but it's worth taking the time to review because knowing the basics of HTTP is essential in order to understand how an App Engine application works. Before we examine how to put together an application protocol for our chat room, we need to review the basics of HTTP.

HTTP is a simple request/response client/server protocol. Put another way, it's a protocol that allows two parties to communicate. One of them is called the *client*, and one is called the *server*. The client and the server behave differently; in HTTP, the client really drives the communication, by sending requests to the server; the server processes the requests from the client, and sends responses. That's basically all HTTP does: it describes a way for a client to send a request to a server and get a response.

To make things even simpler, every request in HTTP is centered on a *resource*. A resource is anything on the network that has a name assigned to it. The other name for a resource is a Universal Resource Locater, or URL. A URL is a sort of filename, except that it can be used

to name many different things: a file, a program, a person, a process, or pretty much anything else you can imagine. Every request in HTTP is either asking to retrieve data from a resource or send data to a resource.

Each HTTP request from the client invokes a *method* on the server. (Don't let the terminology confuse you: even though they call these "methods," there's really nothing object-oriented about HTTP.) There are four basic methods in HTTP (plus about a dozen extensions, which we won't look at because they aren't needed for web applications):

GET Asks the server to retrieve some information from the resource and send it to the client.

HEAD Asks the server to tell it information about a resource. It's basically like a GET request, except that the response only contains metadata. You can use head requests to do things like ask, "How big is this resource?" without having to download the entire thing. In general, most web applications don't use the HEAD method, but it's useful once in a while.

PUT Stores data in a resource. The client sends information to the server to store in the target resource, and the server response just says whether or not the data was successfully stored.

POST Sends data to a program on the server. POST requests are sort of strange. There's not a lot of difference between a PUT and a POST. The distinction really dates back to the early days of the World Wide Web, when people ran web servers on private, individual machines. In early web server implementations, all GET and PUT requests were interpreted as requests to fetch or store files. So to do something that ran a program on a web server, you needed a different HTTP protocol request, which specifically asked you to run a program. In modern systems, we use PUT and POST pretty indistinguishably.

Every HTML request that you generate using a web browser and every HTML request that your App Engine services will process has three parts:

- The request line, consisting of the HTTP method for the request, followed by the URL of the resource, followed by a protocol version specification. In most requests that you make using your browser, the method is GET, and the version specification is usually HTTP/1.1.
- A series of *header* lines, containing metadata about the request (such as the Content-Type specification we used in the welcome application in Section 2.3, *Starting to Program in Python with App*

Engine, on page 25). Most of the requests from a browser will have header lines telling the server what browser you're using (the User-Agent header) and some user identifier (the From: header). Headers can also contain cookie references, language identifiers, network addresses, and so on. In fact, anything can be put in a header: servers just ignore headers that they don't recognize.

- A body, consisting of some arbitrary stream of data.

A blank line separates the end of the headers from the beginning of the body. In general, the message body for GET and HEAD requests is empty. For example, here's a simple GET request:

```
GET /rooms/chatter HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101
Host: markcc-chatroom-one.appspot.com
```

When you send an HTML request to a server, the server responds with a similarly structured message. The difference is that in the place of the request line, the server starts the response with a *status line*. The status line starts with a status code and a status message that tell you whether the request was processed successfully, and if not, what went wrong. A typical status line that your cloud service needs to handle looks like HTTP/1.1 200 OK, where HTTP/1.1 tells you the protocol version being used by the server, 200 is the status code of the response, and OK is the status text.

The status code always consists of three digits. The first digit is the general response kind, where:

1 means "informational response."

2 means success.

3 means redirect, which tells the client to look in a different location to get the resource. (Essentially, a redirect is a message to the client meaning, "If you want that resource, look for it at this other URL.")

4 indicates a client error (for example, 404 means that the client requested a resource that doesn't exist on the server).

5 indicates a server error (for example, the request caused the server to execute a script, and the script crashed).

For example, here's a successful response to the GET request shown above:

```
HTTP/1.1 200 OK
Date: Sat, 26 Jun 2009 21:41:13 GMT
Content-Type: text/html Content-Length: 123

<html>
  <body>
    <p> MarkCC: Hello, is there anybody out there?</p>
    <p> Prag: Yes, I'm here.</p>
  </body>
</html>
```

Let's try walking through a complete request/response cycle. Suppose our application uses POST to submit chat messages. A request could look like the following:

```
POST /submit HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101
Host: markcc-chatroom-one.appspot.com
From: markcc@phouka.local

<ChatMessage>
  <User>MarkCC</User>
  <Date>June 26, 2009 16:33:12 EDT</Date>
  <Body>Hello, is there anybody out there?</Body>
</ChatMessage>
```

If that was successful, the response could be something like this:

```
HTTP/1.1 303 See other
Date: Sat, 26 Jun 2009 21:41:13 GMT
Location: http://markcc-chatroom-one.appspot.com/
```

3.3 Mapping Chat into HTTP

To make our Python chat application work as an App Engine web app, we need to map the basic operations of the application on to HTTP requests and responses.

In this version, we won't deal with subscriptions: there's one chat room, and if you connect to the chat application, you're in the room. For now, we don't need to worry about users entering and leaving.

Imagine you're using a chat room. What are the things that you want to be able to do?

First of all, you want to see any new messages in the room. Translating that into HTTP, the room is a resource, and you want to see its contents. That naturally fits as a GET: you want your browser to retrieve the contents of the chat room and then show it to you.

HTTP Status Codes

The HTTP standard has an extensive list of status codes for server result messages. These are the ones you'll encounter the most:

200 OK The request was completed successfully. The body of the message will contain the result of the successful request.

301 Moved permanently The requested has been permanently moved, and this and all future requests for the resource should be sent to the new location.

303 See other For this request, the result can be found by doing a GET on a different URL. The URL can be found in the Location header of the message. This is generally used for the result of PUT requests, where after the PUT was successfully processed, the server tells the client where to look to see the result of the operation.

401 Unauthorized The request was valid, but the user has not provided any authentication data to show that he or she should be permitted to see the resource. The user could use some other request to get an authentication code and then retry the request.

403 Forbidden The request was valid, but the user is not allowed to access the specified resource. This is similar to 401 but indicates that either the user has been authenticated and still can't access the resource or that even if the user was authenticated, that user is not permitted to access it.

404 Not found No resource was found at the specified location.

500 Internal Server Error Any error in the server during its processing of the request will end up generating a 500 error. In particular, for App Engine services that you implement, the client browser will receive a 500 when your request handlers crash and/or throw an exception.

501 Not implemented The request wants to perform some operation that the server doesn't support. You'll see this most often if you do something like misspell the URL in a POST request.

You also want to be able to send messages, so you need your browser to be able to talk to the chat room as an active entity and tell the chat room application that you've got something to say. Again, the chat room is the resource, but this time you want to talk to it. That's either a PUT or a POST. We decide whether to use PUT or POST by asking, essentially, do you want to replace the contents of the resource, or do you want to talk to the resource? Posting a message to a chat room is clearly the latter. We don't want to replace the contents of the chat room; we want to talk to it and tell it that there's a new message to be added to the conversation. So sending a new message is definitely a POST.

That gives us the framework that we need for our application. We're going to have one resource, which is a chat room. Users can GET that resource to see the current state of the chat. We need another resource, which is the active process that they'll POST to when they send a new message to add to the chat.

Now we need to think a bit about UI issues. How is the user going to be able to POST data to our application? We need to provide a way to do that. The easiest is to create a form in the page that's sent when a user asks to view the chat room. So the chat page will have a title at the top, and then it will have a transcript of what's in the room, and then, at the bottom, we'll have an entry form that takes the user's name and the message that the user wants to post.

To implement this in App Engine, we need to build one RequestHandler that implements GET for the chat room content and then build another RequestHandler that receives the POSTs and adds things to the chat.

The chat room main page is almost the same as the code we used in Chapter 2, *Getting Started*, on page 20. The main difference is that we need to add some dynamic content to it; that is, we need to generate the text for the messages that have been posted. So, we can't just use quoted HTML; we need to generate some. For our first attempt, we'll create a global variable that contains a list of messages. When we render the page, we'll iterate over that list of messages and add them to the page.

Download chattwo/chattwo.py

```
class ChatMessage(object):
    def __init__(self, user, msg):
        self.user = user
        self.message = msg
        self.time = datetime.datetime.now()
```

```
    def __str__(self):
        return "%s (%s): %s" % (self.user, self.time, self.message)

Messages = []

class ChatRoomPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
           <html>
             <head>
               <title>MarkCC's AppEngine Chat Room</title>
             </head>
             <body>
               <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
               <p>(Current time is %s)</p>
           """ % (datetime.datetime.now()))
        # Output the set of chat messages
        global Messages
        for msg in Messages:
            self.response.out.write("<p>%s</p>" % msg)
        self.response.out.write("""
           <form action="" method="post">
           <div><b>Name:</b>
           <textarea name="name" rows="1" cols="20"></textarea></div>
           <p><b>Message</b></p>
           <div><textarea name="message" rows="5" cols="60"></textarea></div>
           <div><input type="submit" value="Send ChatMessage"></input></div>
           </form>
         </body>
       </html>
       """)
```

Handling a POST is a completely new step, but the webapp framework makes it really easy to do. In a handler for a GET request, we implement a subclass of RequestHandler that has a get method. For a POST request, we follow the same pattern: we implement a method of a RequestHandler, in this case, the post method. The RequestHandler superclass ensures that when post is called, the fields of the object contain everything we could want to look at from the request. To get at the data from the fields of the form that produced the POST, we just call the get method on the request, using the label specified in the form. In our code, we'll get the username and message from the POST request, and using them, we'll create a message object and add it to the global message list.

Download chattwo/chattwo.py

```
def post(self):
    chatter = self.request.get("name")
    msg = self.request.get("message")
    global Messages
    Messages.append(ChatMessage(chatter, msg))
    # Now that we've added the message to the chat, we'll redirect
    # to the root page, which will make the user's browser refresh to
    # show the chat including their new message.
    self.redirect('/')
```

Now, we need to put it together as an application. There are two parts: first, we need to write the Python code that creates the application object and maps requests to our request handlers; and second, we need to write the app.yaml file. Then we'll be able to test our application.

The app.yaml file is pretty much exactly the same as before. I changed the name of the Python file for the new example, so we need to change the script entry in the app.yaml file to the new name.

Download chattwo/app.yaml

```
application: markcc-chatroom-one
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: chattwo.py
```

Here is the Python webapp glue code:

Download chattwo/chattwo.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomPage)])


def main():
    run_wsgi_app(chatapp)

if __name__ == "__main__":
    main()
```

Now when we run it (using dev_appserver.py, as in the last chapter), we have a simple, working chat room. Go ahead, give it a try. It works beautifully in dev_appserver.py! So now we can upload it to the App Engine servers. Just like before, we use appcfg.py update to upload it to App Engine.

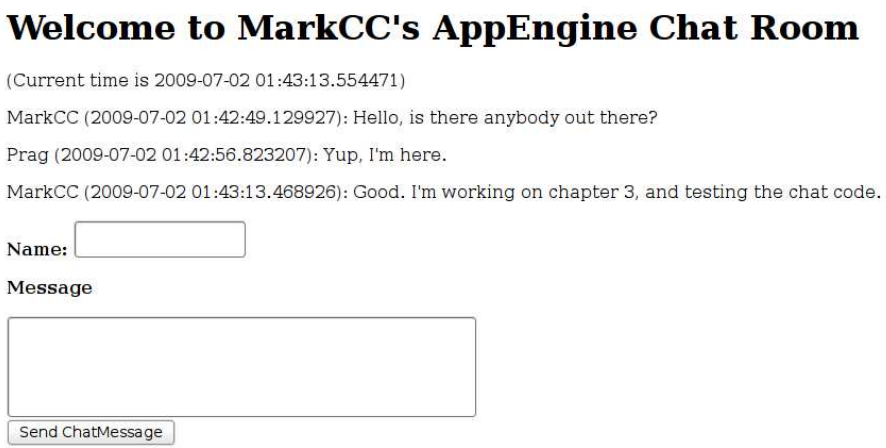


Figure 3.2: The Chatroom app in action

You can see the result in Figure 3.2. It looks exactly like it did running on the local development server. I sent a few messages using two different usernames and got a beautiful chat transcript.

But then I needed to take a break to give my son a bath and put him in bed. When I got back, I sent another message. You can see the result in Figure 3.3, on the next page.

When I sent the new message, all of the older messages were gone! The transcript doesn't have anything except the new message I just added. We didn't write any code to delete old messages, in fact, our code has absolutely no concept of getting rid of messages; we just continually add messages to the chat room. So what happened to the transcript that we had before? Where did the old messages go?

They didn't go anywhere. We got bitten by one of the basic and important differences between cloud code and regular code. When you write code for your own server, you know that every request is going to be handled by your server. If you start a Python interpreter and send requests to it for processing, you know that the Python interpreter will stay around for as long as you want it to.

When you send a request to a cloud server, it gets routed to *some* server in *some* cloud data center. There's no guarantee that any two requests will be routed to the same server or even to servers on the

Figure 3.3: The Chatroom app after taking a break

same continent! Even if they do wind up running on the same server, there's no guarantee that the cloud server will keep a Python interpreter running your code all of the time. In cloud-based programming frameworks like webapp, request handlers are *stateless*, which means that you cannot count on any variables containing a value that was set while processing a different request. You need to program as if each and every request was running in a completely new Python interpreter.

The reason that the application worked at all was basically pure luck. When we ran it locally, the dev_appserver just used a single Python interpreter, so the application ran fine there. When we uploaded it to the App Engine server, it was running out in the cloud. When the first request was received by App Engine, it started a Python interpreter and used it to run the request. When I submitted a message, that sent a second request off to App Engine. The main App Engine server saw that there was a Python interpreter on one of the data-center processors that had just handled a request for that application and that the interpreter wasn't busy, so it routed the request to that interpreter.

But then I took a break and was away from my computer for fifteen minutes; at some point, the App Engine service noticed that the Python interpreter running my chat application had been idle for a while, so it shut it down. When I submitted my next message, there was no live Python interpreter running the chat room code, so it started a new one.

To get around this issue, we need to be completely explicit about how we manage data that we want to share between different requests when building a cloud application for App Engine. We can't rely on module or class variables to manage the state of our application: we must explicitly store all of our application's data whenever we change it and explicitly retrieve data whenever we want to access it.

The point to take home is you can't take basic data management for granted in cloud apps: you need to be explicit about it. Fortunately, webapp provides a very nice, persistence service, called datastore. I'll describe datastore in the next chapter.

References and Resources

RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. . . . . http://www.w3.org/Protocols/rfc2616/rfc2616.html
The HTTP 1.1 protocol standard from the W3C.

Wikipedia article on HTTP . . . . . http://en.wikipedia.org/wiki/Http
A concise, thorough, informal description of HTTP.

Django . . . . . http://www.djangoproject.com/
Django is a widely used web service development platform. It's one of the alternative frameworks that can be used in Google App Engine. Many App Engine facilities are modeled on or borrowed from the Django framework.

Django Nonrel . . . . . http://www.allbuttonspressed.com/projects/django-nonrel
Django Nonrel is a variant of Django that's focused on making it easier to use Django with platforms that aren't based on relational databases, like App Engine.

Chapter 4

Managing Data in the Cloud

In this chapter, we're going to modify our chat application to use persistent storage with the Google App Engine datastore. Along the way, we'll look at the general problem of storing and managing data in a cloud application.

4.1 Why Didn't Chat Work?

When we left off in the last chapter, our chat application had a problem. It seemed to be working, but if we stopped using it for a few minutes and then went back, it lost the history of the chat.

The reason is that our chat application currently relies on global variables for storing the chat history. But when we're running an application in the cloud, that method doesn't work: the data in global variables doesn't necessarily survive between different requests.

The first question to ask is why? What is it about cloud applications that forces us to handle data differently?

When you run an application on your own computer, your operating system creates a process. The process allocates memory for storage, where it keeps all of its data. The process continues to run on your computer with the storage it allocated from the operating system until you tell it to quit. If you use the application for a while, go off and do something else, and then come back to it, it's still the same process, which has access to the same memory.

The cloud is a whole different world. The rules that you're used to—even things as simple as the ways that variables work—are very different. This program isn't running on your computer; it's running on who-knows-how-many computers, which are living who-knows-where

in some data center connected to the network. We wrote a cloud application and uploaded it using Google App Engine's service. From the moment we did that, the program was running. But what it means for it to run isn't what you'd expect from your desktop experience. A program that's "running" in the cloud might not actually be running on any computers. In fact, as I write this, I've got the final version of the Python chat room application uploaded to App Engine—and it's actually running on absolutely no machines at all! (And how do I know that? Because I uploaded it several weeks ago, and I haven't loaded the page into my browser in at least a week. App Engine's servers definitely don't have it loaded on any computers right now. If I load the chat app's page in a browser, App Engine will load it to handle the request.)

As we saw in the earlier chapters, cloud applications are built around request processing. If there are no requests waiting to be processed, there is no need for the program to be running on any computer. If there are a few requests coming in, running it on one computer might be enough. If your application got mentioned on Slashdot, you might need a thousand machines to handle all of the incoming requests! That's one of the things that makes the cloud such an interesting place to program: it gives you a self-scaling platform.

Of course, that comes at a price. If the program could be running on many computers or it could be running on no computers, then there's no way that you can count on what is actually in memory when your code starts to handle a request.

You can't count on your application's request handlers having data in memory when it starts processing a request. While you're processing a request, every single piece of data that you might want to be able to look at in the future must be explicitly saved into a shared storage area called *persistent storage*. Every piece of data that you want to use in processing a request must be retrieved from persistent storage.

How do we work with persistent storage? That's one of the aspects of cloud programming that is fundamentally different from old-fashioned local applications. When you're just starting to write cloud applications, the process can seem cumbersome and frustrating. But it's not a bad thing; the fact that your application is made up of stateless pieces has some wonderful effects. It's part of what makes your programs *scale*. If your application is handling one request every five seconds, running it as a Python application on a single computer would work fine; you could use global variables for data and get all of your results without having to deal with fancy persistent storage. But if your application

The Cloud: Functional Programming?

The cloud as an environment encourages something like a functional style of programming. In functional programming, you aren't allowed to store mutable state; in other words, you aren't allowed to use assignments to alter the value of variables. You can't store anything to be shared between different calls to the same function. You need to pass everything that the function needs *explicitly* as a parameter. That style is very natural for cloud programming.

If you're not used to it, the functional programming style can seem very unnatural, very difficult. But it really isn't—in fact, it's downright attractive once you get used to it! The more complicated your application becomes, the more attractive the functional style becomes.

At Google, we generally program in three languages: C++, Java, and Python. None are functional languages; they're all state-heavy, imperative, object-oriented languages. But the more I've read and written code in this code base, the more I've found that functional code is the best way of building large things. If the code is basically functional, I've found that it's much easier to understand and test and less likely to produce painful bugs. It's gotten to the point where I cringe a little when I see code that isn't functional. Almost everything that I write ends up being at least mostly functional—now I use non-functional code only when the language and the compiler aren't up to the task of keeping the code efficient.

started getting more users, what would happen? One request a second, no problem. 10 per second? No problem. 100 per second? That might start to get difficult. 1,000 per second? 10,000 per second? 100,000 per second? At some point, your application will break: it won't be able to handle the number of requests it receives. But in the cloud, as the number of requests increases, the number of machines running your application can also increase, so no matter what your request rate, you always have the capacity to handle them. A good persistent storage mechanism means you don't need to worry about how many machines are running your program. Whether it's one, ten, or one thousand machines makes no difference. In my project at work, my code runs on a network of thousands of machines every night. In that kind of environment, sharing data using global variables is obviously

General Data Management in the Cloud

Every cloud programming system provides some mechanism for storing persistent data. The exact mechanics vary, but the basic mechanism is almost always database-like. Some systems give you access to a small, fast database system like MySQL. Others, like App Engine, provide more flexible, database-like storage. We'll only look at the App Engine datastore, but there are plenty of others, some of which can be used by App Engine programs.

ridiculous: how can an assignment to a global variable in my Python program be shared among a thousand machines? But because the system uses persistent storage, it's never a problem. When one part of the system gets too slow and starts to exceed its deadlines, I just change one configuration file specifying the maximum number of machines that it can use, and that's all I need to do. It starts running on more machines, which allows it to finish faster.

4.2 Making Chat Persistent

Google App Engine has a custom data persistence system called the *App Engine datastore*. The App Engine datastore is very database-like, only it's a lot easier to use for things like Python objects. Unlike relational databases, datastore does not require a strict schema; it's very flexible and dynamic about how it lets you store and manage persistent data. For retrieving things in Python, App Engine provides a custom query language called GQL. GQL looks a lot like the SQL language used to query conventional relational databases, but it's customized for working with datastore objects instead of relational tables. You don't need to use GQL to work with the datastore, but it's really well suited to the task and easy to use.

Creating and Storing Persistent Objects

The datastore has a lot of options to let you do things in the way that makes the most sense for your application. The basic datastore operations are easy to use. As you use the datastore more, you can start to use more complex features as you need them. For now, we'll stick with the basics.

GQL and the Datastore

If you've worked with a relational database before, you know that the query language, SQL, is a deep part of the database. SQL is the only way that you interact with the database. All of the data in the database is stored in relational tables. Everything that you do with those database tables is done using SQL. Because of this, SQL and the database are, effectively, nearly one and the same thing to you as a relational database programmer.

In the Google App Engine datastore, that's not the way that things work at all. App Engine provides a basic storage engine that allows you to store objects and data. That storage system doesn't have a query language at all. The datastore is nothing but a high-volume, massively parallel storage system.

Internally, the datastore uses indices to help make queries work quickly. Normally, the indices are generated automatically based on the tests that you run in the App Engine development environment. It's also possible for you to build your own indices, as we'll see in Chapter 14, *Advanced Datastore: Queries and Indices*, on page 216. GQL is a query language that's been built for working with the datastore and these indices. Basically, GQL is nothing but a wrapper over datastore indices. It's a convenient, lightweight query language that makes it really easy to search datastore indices. It's not really a part of the datastore. It's just a useful library that was written to help you use the datastore.

Understanding GQL's purpose helps make sense out of a whole lot of things about how GQL works. GQL isn't an integral part of datastore; it's just a utility that helps make it easy to use datastore indices.

The App Engine datastore is pretty different from how you'd normally program in Python. Normally when you create a class in Python, you don't need to declare the fields of the class, you just assign values, and the fields are automatically created. To use the datastore, you have to give up some of that flexibility. With the datastore, you have to create *models* of your objects, which tell the datastore what fields the object will have and what types of values they will have. (Actually, you can use the Python assign-as-you-want style by using something called an expando model, but you really shouldn't. For a cloud application, you really should think out your data well enough to define a proper model for it.)

Enough background. The easiest way to grasp the App Engine datastore is by jumping right in and looking at some code. As I said, in the datastore, you need to define a model to tell the datastore about your objects. In Python, the model is a class object that is a subclass of db.Model, and the fields are defined by creating class-members of the model class. It's a sort of awkwardly non-Pythonic way of doing things. Below, I've taken the ChatMessage from our chat application and turned it into a datastore model:

Download persist-chat/pchat.py

```
class ChatMessage(db.Model):
    user = db.StringProperty(required=True)
    timestamp = db.DateTimeProperty(auto_now_add=True)
    message = db.TextProperty(required=True)

    def __str__(self):
        return "%s (%s): %s" % (self.user, self.timestamp, self.message)
```

In the App Engine datastore, a model defines a collection of named *properties*. You define a type of storable object by creating a subclass of db.model, and you define the properties of the object by assigning property types to class variables in the class itself. The datastore supports a good collection of datatypes: strings, numbers, dates, lists, references and more. It even lets you define your own new types of storable objects. We'll talk more about the complex things you can do in Chapter 13, *Advanced Datastore: Property Types*, on page 191.

Our chat message has three fields: a string containing the name of the user that sent the message, another string containing the message, and a timestamp that specifies when the message was sent. Each of those fields is specified as a property.

user The username is just a string property. Every message must have a username, so we specify that it can't be null by providing the keyword argument required=true. The value of a string property in the datastore is just a Python string, which cannot be longer than 500 characters.

time The time property is an instance of db.DateTimeProperty, which specifies a property whose value is an instance of Python's datetime. For this property, we get to use an interesting capability of the way that the datastore represents properties using Python classes. Every message should have a timestamp. But we don't really want to have to specify it when we create a message; we want the timestamp to be *now*—that is, the time when the message was received by the application. So what we do is use a special keyword parameter auto_now_add for the property that says, "If this property isn't explicitly initialized when an instance of the model type is created, then automatically initialize it to the current time." Because the property is represented by an instance of a Python class, the class can define custom initializer parameters to provide type-specific functionality like auto_now_add without requiring any special primitives. As you'll see when we look at advanced datastore topics in Chapter 13, *Advanced Datastore: Property Types*, on page 191, you can define your own new property types and provide your own type specific extensions.

message Finally, we get to the content of the message. Like the user field, message is a required string property. But in the datastore, a string can't be more than 500 characters. Probably most chat messages will be shorter than that, but not all of them. So instead of using db.StringProperty, we use db.TextProperty. db.TextProperty is a string that can be as long as you want, but because it's an arbitrary length, you can't use it for sorting or searching.

Since we've created a model with the information needed to describe its instances, we don't have to provide our own initializer method now. db.Model will autogenerate an initializer with keyword parameters and types based on the property names and types we specified as fields of the class.

We've got a storable class. How do we actually store values? It couldn't possibly be any easier: every object that is an instance of a subclass of db.Model provides a zero-parameter method called put. If you call put on an object, it's stored in the datastore for your application. Here's

a modification of our POST handler, which creates an instance of our ChatMessage class, and then, at ❶, it stores the new chat message:

Download persist-chat/pchat.py

```
class ChatRoomPoster(webapp.RequestHandler):
    def post(self):
        chatter = self.request.get("name")
        msgtext = self.request.get("message")
        msg = ChatMessage(user=chatter, message=msgtext)
❶       msg.put()
        # Now that we've added the message to the chat, we'll redirect
        # to the root page,
        self.redirect('/')
```

That's it. Calling put() on a model instance stores the instance in the datastore and makes it available for retrieval using queries.

Retrieving Persistent Objects

The last thing we need to know is how to retrieve what we've stored. Below is the part of our GET handler that retrieves all of the messages from the datastore; the rest of the method—everything outside of the part that retrieves the messages and prints them—is completely unchanged.

Download persist-chat/pchat.py

```
# Output the set of chat messages
messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY time")
for msg in messages:
    self.response.out.write("<p>%s</p>" % msg)
```

You can retrieve things using a query language called GQL. As you can see from the code, GQL looks a lot like SQL. The big difference is that GQL isn't querying over tables, it's querying over model types. The query from our chat room selects all instances of ChatMessage, not over all rows of a table.

Depending on what you want to query, sometimes it's clearer to use a different style of GQL. You can omit the SELECT * FROM type part of the query by calling the gql method of the model class. For example, the GQL query from our code above could also be written ChatMessage.gql("ORDER BY timestamp").

Using GQL Queries to Improve Chat

One problem that our chat application has is its verbosity. Right now, each time you refresh your display of the chat, you get the entire chat.

The App Engine Datastore versus Relational Databases

At this point, the difference between datastore models and relational database tables might sound small. After all, every instance of ChatMessage is exactly the same: they've got a set of typed fields, which look a lot like the columns in a relational database. At a first glance, it looks pretty much like a relational database that uses stylized Python classes to create its tables instead of SQL CREATE TABLE statements.

That is, in fact, very much *not* the case. The datastore has a much richer range of data types and data structures than a relational database. In the datastore, we can have properties of a model that have list types, where the elements of the list can be any storable value and where you can use the elements of the list as a part of a GQL query. You can have reference properties, which are used to describe non-containment links between objects. You can have hierarchical, tree-structured datatypes and queries that traverse the tree. (That's not to say that the datastore is better than a relational database; it's just different. For example, relational databases have much better performance on joins than the datastore. But the datastore lets you use familiar data structures that make sense in your application in a clear, scalable way.)

After a conversation has been going on for a while, that gets to be very long, and the part that you're interested in is the most recent part of the chat, which is all the way at the bottom of the page.

Chat room users don't want to have to constantly scroll through the messages they've seen before. Most of the time, they know what was said before and only want to see the latest messages. For example, they might want to only see the last twenty messages in the chat room, or they might want to only see messages posted within the last five minutes.

Using GQL, it's downright trivial to fix the verbosity issue by adding clauses to our GQL query. To see the twenty most recent messages, we can add a LIMIT clause, and to see the messages from the last five minutes, we can add a WHERE clause.

Of course, we don't want to restrict our users so that they can *only* see one of those concise views—when they first enter a new chat, they

may want to see the entire history. So we'll add new handlers to our application for the two new cases. We'll leave the full chat where it was and add two new URLs for time-limited and count-limited short views.

Adding the Count-Limited View

First, let's add the counted view. GQL queries have a LIMIT clause, which specifies a maximum number of results for the query. For example, when you indicate LIMIT 20, you get the first twenty values that match the query in the specified sort order. Since we want to get the twenty most recent query results, we need to make sure that the results we want are the first ones. We do that by sorting in order by time, with the most recent times first.

The counted view is implemented using a RequestHandler, which is the same as ChatRoomPage apart from two lines. I copied ChatRoomPage and renamed the copy to ChatRoomCountViewPage. The modified get method is shown below:

Download persist-chat/pchat.py

```
class ChatRoomCountViewPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
           <html>
             <head>
               <title>MarkCC's AppEngine Chat Room (last 20)</title>
             </head>
             <body>
               <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
               <p>(Current time is %s; viewing the last 20 messages.)</p>
           """ % (datetime.datetime.now()))
        # Output the set of chat messages
❶       messages = db.GqlQuery("SELECT * From ChatMessage " +
                               "ORDER BY timestamp DESC LIMIT 20").fetch(20)
❷       messages.reverse()
        for msg in list(messages):
            self.response.out.write("<p>%s</p>" % msg)
        self.response.out.write("""
           <form action="/talk" method="post">
          <div><b>Name:</b>
           <textarea name="name" rows="1" cols="20"></textarea></div>
           <p><b>Message</b></p>
           <div><textarea name="message" rows="5" cols="60"></textarea></div>
           <div><input type="submit" value="Send ChatMessage"/></div>
           </form>
         </body>
       </html>
       """)
```

There are only two real changes:

❶ In the query itself, we've specified the sort order as descending, so that the twenty most recent posts to the chat will be the first ones in the query result (ORDER BY time DESC), and we limited it to twenty results (LIMIT 20).

❷ The query produced the messages in descending order by time, with the most recent message first. When our users read a chat, that's not the order that they're going to expect: when you're reading a chat, you want the chat to appear in natural order, which means that the most recent message should be at the end. So we need to reverse the order of the query result before we print it.

Adding the Time-Limited View

Adding in a view that selects a subpart of the chat based on time is more complicated than the count-limited view. It requires adding a comparison to the query, and it runs into one of the biggest limitations of GQL: in GQL queries, you can't do any computation. You can't use expressions like x+1. Every computation needs to be done in Python code outside of the query and then inserted into the query.

In addition, the places you can use parameters are limited. In general, you can only use parameters in the WHERE clause of a query.

To really get the sense of those two restrictions, we need to see some parameters in GQL. A parameter is basically a slot in a query where we can inject a Python value.

For example, if we wanted the number-limited view to support different numbers of messages, it might seem natural to use a parameter in the query: ChatMessage.gql("ORDER BY time DESC LIMIT :1", 20). :1 is a parameter for the query, which will be replaced by the first unnamed parameter following the query string, in this case, 20. Unfortunately, we can't do that: you can't use the parameter in the LIMIT clause. However, we can still work around that by using Python string substitution: ChatMessage.gql("ORDER BY time DESC LIMIT %s" % 20).

We can use a parameter for the time-limited view because the relevant parameter is part of the WHERE clause. To do the time-limited view, we have to do some time arithmetic. If we want to show the messages posted in the last five minutes, we'll say that in the query as something like, "All messages whose timestamp is larger than now minus five minutes."

In Python, we can say "now minus five minutes" using the datetime module: datetime.now() - timedelta(minutes=5). To use it in a query, we just need to inject it using a parameter. So we wind up with:

```
ChatMessage.gql("WHERE timestamp > :fiveago ORDER BY time",
                fiveago=datetime.now() - timedelta(minutes=5))
```

And that's all it takes. Just copy ChatRoomPage, rename it to ChatRoomTimeViewPage, and replace the query with the fragment above, and you've got it.

Parameters can be either numbered or named. If they're named, specify their value using a named parameter to the Python call, as we did above. If they're numbered, you just specify them in order. For example, we could use a named parameter in the time-limited view query like ChatMessage.gql("WHERE timestamp > :1 ORDER BY time", datetime.now() - timedelta(minutes=5)). For any given query, it's a good idea to use either all named parameters or all positional—mixing the two is likely to lead to confusion. (In fact, I think that most of the time, you should use named parameters exclusively; the only disadvantage is that it's a tiny bit more typing, but it makes your code clearer, and in case of errors, it makes the errors easier to understand.)

Of course, to be able to see and test this, we need to modify the WSGIApplication to direct queries to our two limited views. Our application now has three views: the full conversation view, the time-limited view, and the count-limited view:

Download persist-chat/pchat.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomPage),
                                  ('/talk', ChatRoomPoster),
                                  ('/limited/count', ChatRoomCountViewPage),
                                  ('/limited/time', ChatRoomTimeViewPage)])
```

We don't yet have a nice way of moving between the views, and their implementations have a silly amount of duplication. We'll look at how to clean that up in the next chapter. But for now, we've got something that works.

Resources

The Python Datastore API. . .

. . . http://code.google.com/appengine/docs/python/datastore/

The official Google App Engine datastore documentation.

Chapter 5

Google App Engine Services for Login Authentication

Most web applications that you build with App Engine must to be able to keep track of users. You need to be able to let users log in and perform tasks based on the permissions you grant them. In this chapter, we'll look at how to manage users and keep track of who is issuing which request. To do that, we'll use an App Engine service, which is a piece of the App Engine API that is independent of the webapp framework.

5.1 Introducing the Users Service

We've made a nice start on our chat system, but it's awfully limited. We started out with a design sketch that allowed multiple chat rooms. Unfortunately, we can't make that design work very well at the moment. In that design, a given user could subscribe to multiple chats, and the chat application kept track of which subscriptions different users were subscribed to. But our chat application doesn't have any way of keeping track of who is making a particular request, so we don't know which chats to show to the user.

Taking care of logins and authentication isn't just something that you need for an application like a chat system. It's something that you'll probably need to do in every App Engine application that you write.

For things like login, which are ubiquitous and so necessary that they'll appear in nearly every application, App Engine provides APIs called *services*. A service is a module provided by the App Engine implementation

that is accessible to every App Engine application, regardless of what framework you use for building your application. Even if you decide to use something like Django for your application, you can still use all of the App Engine services.

The easiest way to set up authentication is to use the Google App Engine users service to piggyback on Google accounts. If your application uses Google email addresses as its main identifier, then it can work with Google logins. You can build your own authentication service or use another third-party authentication service if you want; there's nothing forcing you to use use GAE's login service. But it's very convenient, and for most applications, it's probably sufficient. Using other services won't be very different—the basic mechanics will be similar.

5.2 The Users Service

Google logins are supported by the users service. The users service keeps track of a currently logged-in user and provides capabilities to allow your application to provide login and logout pages.

User Objects and the Current User

The easiest thing to do with the users service is to retrieve the user object for the currently logged-in user. You can always retrieve the user by calling users.get_current_user(). If there's no user logged in from that client, the call returns None. If there is a logged-in user, you get back a Python object with three instance methods:

nickname() A textual name associated with the user. This will eventually be user configurable, but at the moment, it just returns the part of the email address before the @ symbol. In my case, it's markcc.

email() The user's email address. In my case, it's markcc@gmail.com.

user_id() A permanent identifier for the user. Treat this as an opaque string. The users can change their email address or their nickname whenever they want, but their user_id will always be the same. This isn't useful for display purposes, but if you want to record permanent information about a user—such as the set of orders placed by a user on a commerce site—you can use this as an identifier that will work no matter what.

Letting Users Log In

We need a login page for our users. Fortunately, we don't need to design our own login page—after all, every login page is pretty much the same. The users service provides a mechanism for generating a login page automatically.

The standard users service login is intended to act as an *interstitial*: that is, a page the users get along the way to what they really want to access. In our chat application, users will first get a welcome screen; from that welcome screen, they enter the chat room to see the ongoing conversation. But we want them to be logged in before they get access so we know who they are. If they're not logged in and they ask to enter the chat room, they'll be sent to a login page. As soon as they're done logging in, they'll be sent directly to the chat page.

With that in mind, the way that the users service works is that you ask it to log in the user and you provide it with a target page. After users successfully log in, it automatically redirects them to the target page.

Our chat is very typical of this style. We can build it to work this way by adding a bit of logic to the get method of the chat page's RequestHandler.

At the beginning of get, we check if the user is logged in by calling users.get_current_user(). If that returns a logged-in user, we go ahead and render the page. If not, we create a login page using the chat page itself as the redirect target for successful logins. That sounds a bit hairy, but it's really not as complicated as it sounds: the call to allow users to login and then redirect back to the chat page if they're successful is just self.redirect(users.create_login_url(self.request.uri)).

In other words, we're performing a redirect that tells App Engine to send the users to a login page. The login page is generated by the users service with users.create_login_url. And when the users have successfully logged in, we redirect them to the chat page. We don't even need to remember the URL for the chat page, we just use the URI of the original request, which is accessible in a request handler as self.request.uri.

5.3 Integrating the Users Service into Chat

Now that we know how to do logins, we can integrate users and log in into our chat application. To do that, we need to make a series of changes to the chat application:

1. Modify the chat page to require users to login.
2. Use the logged-in user object to set the user field of messages in chat and remove the user field from the chat message entry form.
3. Modify the POST message handler to use the logged-in user.

We've already seen how to do user logins in Section 5.2, *Letting Users Log In*, on the previous page. Integrating that functionality into our chat page request handler is just more of the same kind of code. The other change in the chat page request handler is removing the name field from the form. The updated request handler is shown below:

Download login-chat/pchat.py

```
class ChatRoomPage(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()
❶   if user is None:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      self.response.headers["Content-Type"] = "text/html"
      self.response.out.write("""
        <html>
        <head>
          <title>MarkCC's AppEngine Chat Room</title>
        </head>
        <body>
          <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
          <p>(Current time is %s)</p>
        """ % (datetime.datetime.now()))
      # Output the set of chat messages
      messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp")

      for msg in messages:
        self.response.out.write("<p>%s</p>" % msg)

      self.response.out.write("""
         <form action="/" method="post">
         <p><b>Enter new message from:  %s
         <p><b>Message</b></p>
         <div><textarea name="message" rows="5" cols="60"></textarea></div>
         <div><input type="submit" value="Send ChatMessage"/></div>
         </form>
       </body>
       </html>
❷      """ % user.nickname())
```

The login code is at ❶ and follows exactly the pattern we described earlier. If the users are already logged in, the chat application renders

the chat for them. If they're not logged in, it redirects to a login page and returns them to the chat page after they've logged in.

The other change is at ❷, where we removed the username entry field from the form and replaced it with a prompt line that uses the nickname fetched from the current logged-in user object.

Changing the POST handler is exactly the same. All we need to do is add a copy of the get_current_user line and use it in the call that creates the ChatMessage, as shown below:

Download login-chat/pchat.py

```
def post(self):
      if user is None:
    self.redirect(users.create_login_url(self.request.uri))
❶ user = users.get_current_user()
  msgtext = self.request.get("message")
  if user.nickname() is None or user.nickname() == "":
    nick = "No Nickname"
  else:
    nick = user.nickname()
  msg = ChatMessage(user=user.nickname(), message=msgtext)
  msg.put()
  sys.stderr.write("****** Just stored message: %s" % msg)
  # Now that we've added the message to the chat, we'll redirect
  # to the root page,
  self.redirect('/')
```

Now that we've got the ability to provide logins, we can start making our chat application much more interesting. We can build things like multiple chats, subscriptions, and other features, such as the ability to establish which users are connected, direct private messaging between users, and so on. Of course, nothing ever comes for free. When we start adding these kinds of features, the HTML that we use to render our user interface gets complicated, and we need to constantly generate the same boilerplate HTML code. That's laborious, error-prone, and just plain annoying. In the next chapter, we'll look at how to add subscriptions and multiple chats to our application and how to use a facility called templates to make it easy to generate the HTML of the UI without errors, inconsistencies, or other kinds of grief.

Chapter 6

Organizing Code: Separating UI and Logic

In all of the code we've written so far, we've been rendering the user interfaces by writing Python code that prints out HTML. Doing that is laborious, awkward, and easy to screw up. In this chapter, you'll learn about templates, which organize the code of a Google App Engine application by separating the code that renders the user interface from the code that implements the application logic. With templates, instead of rendering the HTML user interface code using Python print statements, you'll write it directly as a marked-up form of HTML. Your application logic will stay in Python, which will invoke the templates when it needs to generate a page.

6.1 Getting Started with Templates

Back in Chapter 3, *A First Real Cloud Application*, on page 37, we sketched out a pretty advanced chat system. We designed it to support multiple simultaneous chats taking place in different chat rooms. Users could participate in multiple chats at the same time by subscribing to different chats.

But since then we've been building applications that have only one chat. In the last chapter, we took care of one of the aspects of a full chat application: recognizing a logged-in user. We needed that function both to save users the trouble of typing in their usernames every time they say anything and to keep track of who is participating in which chat. We have the ability to keep track of users, but so far it's just a

convenience for the users. We're not managing multiple chats; there's no information about the users that we need to keep track of or protect.

We want to start implementing some more interesting features. To do that, we'll be introducing several new views. As we've seen in the previous chapters, adding new views can be painful because writing code to generate the HTML is clumsy and error-prone.

To make things both easier and more maintainable, we'll learn to use a facility called *templates*. Templates provide a flexible, easy-to-use system for creating the HTML code that describes our user interface. It does this by allowing us to separate the logic of our system from the appearance. The logic we'll continue to write in Python. The appearance we'll write in annotated HTML template files.

There are many template languages. The Google App Engine webapp framework includes one from the open source Django project, so that's what we'll use. If you prefer another, go ahead and use it: just put the Python files that you need into your project, and it should work.

Why Learn Another Language?

Like all web applications, cloud applications build their user interfaces by generating HTML that will be rendered by a browser. It's convenient, because it makes it easy to create all sorts of attractive interfaces on the fly. Web browsers have a very flexible, consistent, pleasant platform for creating user interfaces, and you can take advantage of everything the browser provides by using HTML. Especially with the upcoming HTML5 standard, you can create beautiful interfaces using HTML.

The problem is that generating HTML correctly can be arduous, and it's easy to make mistakes. HTML has a very verbose syntax and uses some of the same quoting characters as most programming languages. That means you must be very careful how you write the code—and no matter how careful you are, it's still easy to make mistakes.

Templates help solve that problem. They let you split your code programs into two parts: the computation and the interface. The computational part is where you do the work and produce the data that you want to render in the web page. In the computational part, you're mostly manipulating data—sometimes, you might end up rendering some of it as HTML, but for the most part, the computational part doesn't do rendering.

The interface is all about generating HTML. It takes the data produced by the computational part and renders it as HTML. In most applications

a large part of the HTML is fixed. In our chat application, every page rendering has to generate the basic HTML structure, the page headers, the message entry form, and so on. Only the content of the message list changes: everything else is exactly the same. That means that there are a ton of print statements that do nothing but output fixed strings that form a piece of the page HTML. The HTML is embedded in the code, and that embedding is, at best, awkward.

With templates, you write the computational part of your code in Python (or whatever language you're using). If you need to do any HTML rendering as part of your computation, it's done using HTML strings embedded in the Python. For the interface part of your code, you write it in HTML and use special metasyntax to embed any computation that's needed to insert dynamic content in the HTML file.

In fact, you could write your entire application using templates. But just as it's awkward, painful, and error-prone to render the HTML from standard Python code, it's awkward, painful, and error-prone to try to use complicated programming logic inside of templates.

Template Basics: Using Templates to Render Chats

The first thing we can do with templates is separate the logic of retrieving chat messages in the datastore from the logic of how we render a page containing those chat messages for displaying a chat. To do that, we'll take the chat page of our application, write it as a template, and then change the Python code to use the template. A really simple version of our chat display page as a template is shown below:

Download template-chat/chat-template.html

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
❶   <h1>Welcome to {{ title }} </h1>
❷   <p> Current time is {% now "F j Y H:i" %}</p>

❸   {% for m in msg_list %}
❹   <p> {{ m.user }} ( {{ m.timestamp }} ): {{ m.message|escape }} </p>
    {% endfor %}
    <form action="/talk" method="post">
      <p><b>Message</b></p>
      <div><textarea name="message" rows="5" cols="60"></textarea></div>
      <div><input type="submit" value="Send ChatMessage"/></div>
    </form>
  </body>
</html>
```

A Django template is a text file with a special non-XML markup syntax embedded. Right now, we're going to use it for generating HTML, but Django templates aren't limited to HTML. You can use them for anything that's built using text: CSS, XML, even Python code!

The Django template system uses the curly braces for marking its syntax. In our example, most of the contents of the template file are just plain XML. Whatever is enclosed in curly braces will be replaced when the template is actually used. In this first example, we only use a couple of elements of Django syntax:

❶ The first template construct we use is a variable reference. In Django, variable references are written by surrounding an identifier with double braces—so {{ title }} is a reference to a variable named title. When the template is used, it will be replaced by the contents of the named variable. A variable reference can also use dotted identifiers; the parts after the dot are references to fields of a Python object. We'll see an example of that a little later.

❷ Next we see what Django calls a *tag*. A tag in Django is something like a function call in a regular programming language. Tag invocations are surrounded by {% and %}. The first word inside the invocation is the name of the tag. The rest are parameters. In this case, we want to insert the current time. Django provides a tag, now, which will be replaced by the current time. It takes a series of parameters that specify how to format the time:

F The textual name of the current month

j The numeric day of the month

Y The numeric value of the year

H The current hour

i The current minute in two-digit format

There are other characters you can use in a date format—look at the Django template documents for the full list. Anything that isn't a letter in the now format string is included in the replacement text, so the spaces and colon in the parameters to now will be included in the text. As a result, the date is rendered in this form: Jul 19 2009 02:45

❸ Here we've got a for-loop implemented as a Django tag. In a Django template file, tags can have bodies. Think of it as similar to XML: in

XML, all tags can have a set of parameters called *attributes*, which are part of the tag itself. More complex tags can also have *content*, which is some other mixture of text and HTML tags that are embedded between the start and end of the complex tag. Django template tags are similar: simple tags take one line and define everything through parameters listed in the tag itself; more complex tags have content, which is everything between the tag and its corresponding end. In Django, the end of the body is marked by {% endTAG %}.

In this example, the tag is a loop. When the template is used, the evaluator will loop over each value in the specified list and produce one copy of the tag body for each element in the list. This loop iterates over the messages in the chat room: for each one, it produces a copy of the body with the variables replaced using a different message from the list of messages in the chat room.

❹ We'll talk about this in more detail in Chapter 17, *Security in App Engine Services*, on page 260, but any time you get input from a user and then insert it into a user interface view, you need to be paranoid. It's really easy for a malicious user to do all manner of evil things with your application by inserting JavaScript into their inputs if you aren't extremely careful. In Django templates, just run user inputs through the escape filter, using | escape.

Now that we have the template for our user interface page, we need to invoke that template from our Python code. An updated version of the ChatRoomPage, which uses the template, is shown below:

Download template-chat/tchat.py

```
class ChatRoomPage(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()
    if user is None:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      self.response.headers["Content-Type"] = "text/html"
      messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp")
      template_values = {
        'title': "MarkCC's AppEngine Chat Room",
        'msg_list': messages,
        }
❶     path = os.path.join(os.path.dirname(__file__), 'chat-template.html')
❷     page = template.render(path, template_values)
      self.response.out.write(page)
```

To use a template, you need to get the pathname of the template file and then call template.render.

Getting the pathname requires a bit of a trick in Google App Engine. As we keep seeing, in the cloud you have less control over your environment than you're used to in a traditional application. Your application and its data are *somewhere*, but you don't know where. The App Engine server determines where it's going to put your data, and it can move it at any time, without warning. In order to access a file, you need to ask App Engine to tell you where the directory containing your application files is located. App Engine uses the Python metavariable __file__, which is a reference to the source file for the current module. App Engine's Python environment preserves the basic directory between files, so if a data file is in the same directory as a Python source code file in your local development environment, then that file will still be in the same directory as the code on the App Engine server. So even though you don't know the full name of the directory where App Engine put your deployed code, you can find the data file because it's in the same directory as the deployed code. To find the template file, we get the directory containing our Python code using standard Python tricks with the __file__ metavariable. Just like any other file, os.path.dirname(__file__) gives us the directory, and we get the pathname of the template by joining that with the template name.

Once we've got a reference to the template file, we call template.render. The first parameter to the call is the template and the second is a Python dictionary. The keys in that Python dictionary will become the variables that can be accessed in the template. Since we reference title and msg_list in the template, those are the keys that we put into the dictionary.

6.2 Building Related Views with Templates

When you're building an application like our chat system, you typically have multiple views that look similar but not exactly the same. For example, we want to have multiple chat rooms with one view for each chat. So we're going to need an index view to select a chat room, plus a view for each of the different chats. The index view and the chat views should have similar appearances; the chats should be identical except for the name of the chat room.

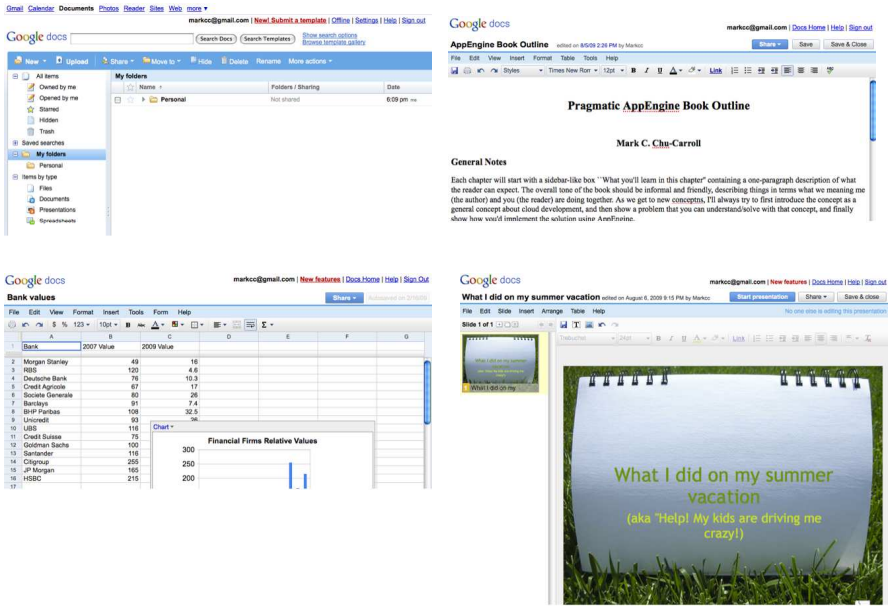


Figure 6.1: Common Styling in Google Docs

Almost every good web application has a unified look and feel across its pages. A good web application isn't just a collection of web pages: it's a cohesive program that provides a service to the user. Each of the pages that make it up provide a part of that functionality. By giving all of the pages of your application an appearance that is distinct from other applications and which is shared by all of the parts of your application, you provide the user with the feeling that they're using a consistent, well-designed application. For example, take a look at the Google Docs thumbnails in Figure 6.1. Docs presents itself as an office suite. It starts with an initial view that looks like a Windows file browser. When you open a document, it opens a new browser window to let you work on that document. Every Docs window has the same basic layout; they all have the Google Docs logo in the upper left corner, a set of control links in the upper right, a document title, and decorations using a family of shades of blue. You can tell at a glance that a window contains something from Google Docs because it *looks* like Google Docs.

The Problem with Copies

I recently had an encounter with a real-world example of the copy problem. The site where I write my blog went through a system upgrade. Before the upgrade, the base-path of all of the URLs was http://.../cgi-bin/MT/; after the upgrade, the base-path was changed to http://.../mt/. It turned out that there were multiple places where the base-path was hardcoded. During the upgrade, most of them got fixed but a couple got missed. As a result, article, comment, and administrative links all broke and needed to be fixed. It took *two weeks* to track down all of the copies and get everything working properly. During those two weeks, the site was a mess for both the writers and the readers. If they'd set up a single function that computed URLs and then just called that function whenever they needed to generate a URL, they could have changed one line of code in one place, and nothing would have broken.

Don't let this happen to you: reuse code in place instead of copying it!

The look and feel that defines your application is created by a combination of common page layouts and shared styles. The styles are defined by cascading style sheets. Both the basic structure and the shared CSS can be defined as templates, and then individual pages can tailor the details to fit particular requirements using subtemplates, all while maintaining the defaults from the master template shared by everything in the application.

These views will have common features like headers, navigation bars, and logos that appear on all of your pages. You can also use stylistic elements, such as particular fonts, text styles, and color schemes to make the pages of your application share a look and feel. All of the elements are shared between the different views. The pages have their own specific content, providing the page functionality, but the style of the pages is all the same and comes from the same code.

There's a basic rule in programming: putting multiple copies of the same thing in different places is a bad idea. Eventually you'll need to change something, and when you do, it's too easy to miss one of the copies. We'd really like to be able to do all of the common page parts once and reference them from the specific pages.

Template Inheritance

One of the most powerful features of Django templates is designed to solve exactly this problem. It's called *template inheritance*. You can define the broadest common structure of all of the pages in your website as a master template and then define the individual pages as variations on that template. You can even create whole hierarchies of templates that become progressively more specialized.

Let's take advantage of that. We'll create a master page layout with a logo on top and a couple of customizable sections. The master template for our chat application is shown below:

Download multichat/master.html

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>

  <body>

    {% block navbar %}
        <h3>Available Chats</h3>
    <div id="navbar">
      <ul>
        {% for c in chats %}
        <li><a href="/enterchat?chat={{c}}">{{ c }}</a></li>
        {% endfor %}
      </ul>
    </div>
    {% endblock %}


    <h1>Welcome to {{ title }} </h1>
    <p> Current time is {% now "F j Y H:i" %}</p>

    {% block pagecontent %}

    <p> This is template text. If you're seeing this in a page rendered
      by chat, something is wrong.</p>


    {% endblock %}

  </body>
</html>
```

The master template isn't intended to be used directly. If it was, it would generate a web page containing the message, "This is template text. If

you're seeing this in a page rendered by chat, something is wrong." It provides a basic format that other templates can build on.

The basic page layout is pretty much the same as what we've been doing all along. The only change visible to a user is that I've added a new logo to the top of the page. But I've also added a bunch of block tags to the template. A block tag identifies a section of the template contents that can be replaced by subtemplates. We could use master.html as the template for our main chat application. To take the basic view of the chat application and build it to use this new master template, we'll create a new subtemplate like the following:

Download multichat/basic.html

```
❶ {% extends "master.html" %}

❷ {% block pagecontent %}
<p> All chat messages as of {% now "H:i" %}</p>

{% for m in msg_list %}
<p> {{ m.user }}@{{ m.timestamp }}: {{ m.message|escape }} </p>
{% endfor %}

{% endblock %}
```

We declare it as a subtemplate by using the extends tag, which must be the first thing in the file. Then we put a block tag for the block we want to override. The result will be a template that consists of the contents of master.html, but its pagecontent block will be replaced by the message rendering loop that we've been using.

So far, templates look nice because they've given us a clean way to write the HTML separately from the Python code. We've been able to separate the UI and the application logic with nothing but a parameter set passed between them. But that's just scratching the surface of what we can do using templates! Template inheritance is an incredibly useful and powerful mechanism. We'll be using them constantly in the rest of this book for creating better and better interfaces for our application.

Customizing Chat Views Using Templates

Let's use what we've learned about templates to make our chat application better. One thing that we could do is to improve the Last 20 view. In the current version, it shows a very verbose form of the date and time when a message was sent. But since most of the time we're going to be showing an active chat, most of the text in those timestamps is going to

be identical. It takes up space, forces messages to take up more lines of the display, and generally makes things harder to read. Ideally, we don't want to include full timestamps, but we do want some indication of time. We can change the display of messages in the Last 20 view, so that instead of showing the full timestamp, it just annotates each message with the elapsed time since it was sent.

On the computation side of things, we'll need to modify the Python code so that it computes the elapsed time and adds it to the messages that we pass to the UI template: in the UI code, we'll need to create a new, modified showmessage block to display the elapsed time instead of the timestamp.

First, we need to update the computation part of the application to add the timestamp. It's pretty straightforward: just use Python's standard timedelta class:

Download multichat/tchat.py

```
class ChatRoomCountedHandler(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()
    if user is None:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      self.response.headers["Content-Type"] = "text/html"
      messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp "
                             "DESC LIMIT 20")
      msglist = messages.fetch()
      for msg in msglist:
        msg.deltatime = datetime.datetime.now() - msg.timestamp
      template_values = {
        'title': "MarkCC's AppEngine Chat Room",
        'msg_list': messages.fetch(),
        }
      path = os.path.join(os.path.dirname(__file__), 'count.html')
      page = template.render(path, template_values)
      self.response.out.write(page)
```

The only change in this code is the addition of the loop after the GQL query, which adds the timedelta field to the message objects. The interesting thing about this little bit of code is that we've added a field to a datastore object. But since it's not a field that was declared as a datastore property, it has no effect on the stored object. Even if we were to call put on one of the messages that we modified by adding a time delta, it wouldn't change anything about the stored object.

Now that we've updated the computational part of our application so that it computes and stores the time since the message was sent, we need to update the interface by replacing the showmessage block. We do that by creating a template like the following:

Download multichat/count.html

```
❶ {% extends "master.html" %}

❷ {% block pagecontent %}
<p> Last 20 messages as of {% now "H:i" %}</p>

{% for m in msg_list %}
<p> {{ m.user }}: {{ m.message|escape }} ({{ m.deltatime }} seconds ago)</p>
{% endfor %}

{% endblock %}
```

❶ We start by declaring that this is a template extension of master.html. That means the generated page will look like master.html, except where we specifically override blocks.

❷ Here we override the pagecontent block from the master page. It will replace the original block with this one, which renders a list of messages.

6.3 Multiple Chat Rooms

Now that we know how to use templates, we can easily put together a range of different views. Let's use that knowledge and change our chat application to make it more useful. In our original application sketch, we wanted to support multiple chats with subscriptions. Let's get started on that by providing multiple chats; we'll worry about the subscription part later.

Updating the Logic for Multiple Chats

Since we want to support multiple chats, we need a way to keep a list of the available chats. Later on, we'll add an administrative view, which we'll be able to use to manage the list of chats. But for now, we'll just hard-code it. We don't need to worry about inconsistent updates, because it's never going to be updated. It will reset to the same value every time a chat message is initialized.

In addition, we'll add something to the chat messages so they know which chat they belong to. That's pretty trivial: just add a datastore

field to the class. We'll also need to modify our POST handler to make it set the chat field, but we'll see how to do that when we set up the chat pages. The modified ChatMessage and the hardcoded chat list are shown below:

Download template-chat/tchat.py

```
class ChatMessage(db.Model):
    user = db.StringProperty(required=True)
    timestamp = db.DateTimeProperty(auto_now_add=True)
    message = db.TextProperty(required=True)
    chat = db.StringProperty(required=True)

CHATS = ['main', 'book', 'flame' ]
```

We also need to make the post method that creates a chat message, so that it sets the chat field of the chat message. We'll do that by adding a field to the request that triggers the post. In the POST handler, we retrieve the chat field from the request and add it to the chat initializer. The code for the modified POST handler is shown below.

Download template-chat/tchat.py

```
class ChatRoomPoster(webapp.RequestHandler):
  def post(self):
    user = users.get_current_user()
    msgtext = self.request.get("message")
    msg = ChatMessage(user=user.nickname(), message=msgtext, chat="chat")
    msg.put()
    # Now that we've added the message to the chat, we'll redirect
    # to the root page,
    self.redirect('/')
```

Building the Multiple Chat Landing Page

When someone first comes to the chat application, we want them to get a landing page—a generic front page. For our application, that front page shows users the last twenty messages posted to *any* chat. From this page, they can see which chats are active and then select one from the navigation bar. From the landing page, they can't post messages—since they haven't selected a chat, we don't yet know which chat to post it to.

Now, we'll fill in the first real page, the landing page. We're not going to change the toolbar. For the content, we'll do a simple view based on what we did in past versions. This is shown in the following HTML:

Download multichat/landing.html

```
{% extends "master.html" %}

{% block pagecontent %}

{% for m in msg_list %}
 <p> <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
 {{ m.message|escape }} </p>
 {% endfor %}

{% endblock %}
```

That's exactly the body that we used in our original chat template in Section 6.1, *Template Basics: Using Templates to Render Chats*, on page 72, with one addition: the chat message displays the chat that it's a part of, in bold, at the beginning of the line.

To render that, we create a new request handler. It's pretty much like the request handlers we've done so far, such as the one we used for ChatRoomCounted above.

This new chat landing page, though, has something different because of its navigation bar: it's got a list of chats, which the users can select. When users click on one of those, they get sent to a specific chat room. The next thing we need to do is create those chat pages.

The Chat Page Template

For the actual chat pages, we're going to be clever. Up until now, each time we've wanted to change anything, we've created a new request handler. But the chat pages are all identical except for the name of the chat—and in fact, later we're going to want to be able to create and destroy chat pages on the fly. So we're going to use a combination of a chat-page template and some clever URL-handling logic in the Python code so that we only have one handler class for all of the chats. The Python code will use the request URL to figure out what chat room is being requested.

The basic chat template is the same old message-rendering loop that we've done time and again, but this time, we put the name of the chat up in the page header and remove it from the messages—after all, there's no reason to keep repeating it, since all of the messages are part of the same chat! Here's the template:

Download multichat/multichat.html

```
{% extends "master.html" %}

{% block pagecontent %}

{% for m in msg_list %}
 <p> <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
         {{ m.message|escape }} </p>
{% endfor %}

    <form action="/talk?chat={{ chat }}" method="post">
      <p><b>Message</b></p>
      <div><textarea name="message" rows="5" cols="60"></textarea></div>
      <div><input type="submit" value="Send ChatMessage"/></div>
    </form>

{% endblock %}
```

It's almost the same as what we did before, except that we added a parameter to the POST request generated by the entry form at the end of the page, which we then use to include the name of the chat from which a message was posted.

The code is where the cleverness comes in. We must to do two things: we need to validate the request, and we need to customize the output based on the URL used in the request.

That validation step is something new. Until now, we've relied on Google App Engine to take care of the validation for us. We mapped each specific URL onto one specific request handler, so we always knew that an invalid request would generate an error. But now we're going to be mapping multiple requests onto a single handler: every request to view a chat is going to be handled by the same RequestHandler. Since we're eventually going to be able to add and remove chats, there's no fixed list of chats that we can hard-code into the app.yaml file or the WSGIApplication instance. When we get a request to view a particular chat or to post a message to a particular chat, we need to check that the chat in the request is valid.

Download multichat/tchat.py

```
class GenericChatPage(webapp.RequestHandler):
  def get(self):
❶   requested_chat = self.request.get("chat", default_value=None)
❷   if requested_chat == None or requested_chat not in CHATS:
      template_params = {
          'title': "Error! Requested chat not found!",
          'chatname': requested_chat,
```

```
            'chats': CHATS
            }
        error_template = os.path.join(os.path.dirname(__file__), 'error.html')
        page = template.render(error_template, template_params)
        self.response.out.write(page)
    else:
❸       messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                               "ORDER BY timestamp", requested_chat)
        template_params = {
            'title': "MarkCC's AppEngine Chat Room",
            'msg_list': messages,
            'chat': requested_chat,
            'chats': CHATS
            }
        path = os.path.join(os.path.dirname(__file__), 'multichat.html')
        page = template.render(path, template_params)
        self.response.out.write(page)
```

❶ We get the name of the chat that the user requested by decomposing the URL. The new URL form has three parts: the hostname, the resource identifier, and a query. The query consists of a set of name/value pairs. In our case, the query will be one pair: the query parameter named chat and the requested chat name. So, for example, to see the chat Random, the URL would be http://markcc-chatroom-one.appspot.com/enterchat?chat=Random.

Examining the elements of an HTTP request is a common task in Google App Engine programs, so webapp supplies an extensive library of methods for examining and manipulating URLs. To get at query parameters from a request URL, use the method get, which takes the name of a query parameter and an optional default value to return if the query doesn't include that parameter.

❷ As we discussed before, we need to check to make sure that the chat requested by the URL is valid. We do that by checking it against the global list of known chats. If the requested chat doesn't exist, we render an error page. The error page is generated from a template derived from our master, so it looks like an error generated by our application, not just a typical "page not found" error.

❸ If the chat room is known, we follow the pattern we've used all along and render the chat. The only difference from what we've done before is that in the query, we select only messages whose chat field matches the chat selected by the user.

We've got a generic handler for rendering and posting messages to all chats. What's left? We need to update the handler for posts to get the

chat from the message. That's easy: we just need to add one line to our POST handler—chat = self.request.get("chat")—and then add chat=chat to the parameters for the ChatMessage constructor call.

Just one piece remains. We have request handlers for all of our requests and templates for all of the pages we need to render. We need to change the application code that maps incoming requests to the appropriate request handler. To do that, all we have to do is change the WSGIApplication record for our app. Now it needs entries for the landing page, the generic chat page, and the post page. The updated code is shown below:

Download multichat/tchat.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomLandingPage),
                                  ('/talk', ChatRoomPoster),
                                  ('/enterchat', GenericChatPage)])
```

In this chapter, we've taken a big step forward in terms of the functionality of our chat application. We separated the program logic from the user interface using templates, and we started to use a common page structure, defined using a master template and template extensions, to provide all of the pages of our application with a common appearance.

Unfortunately, despite the added function our application is still ugly. It doesn't really look like an application—it looks like a bunch of web pages. The page layouts are sloppy and generic. In the next chapter, we'll look at cascading style sheets (CSS), which let us describe how to define and lay out a really nice-looking application in a web page, and how to use CSS with templates to turn our chat application from an ugly-but-functional system to something much better.

References and Resources

The Django Template Language: for Template Authors. . .
. . . http://docs.djangoproject.com/en/dev/topics/templates/
The official Django template documentation.

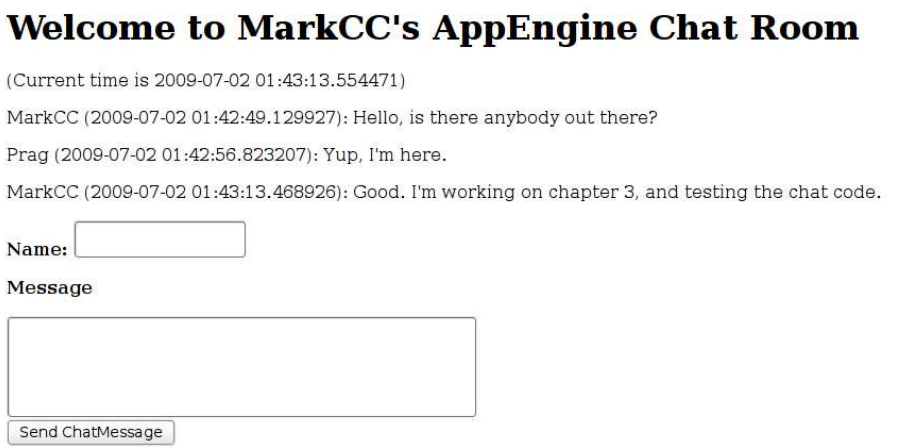
Google App Engine Template Documentation. . .
. . . http://code.google.com/appengine/docs/python/gettingstarted/templates.html
Google's documentation on using Django templates in App Engine applications.

The Django Book 2.0 . ..... http://www.djangobook.com/en/2.0/
An online version of a book on the full Django framework, including a detailed presentation of the template language.

Chapter 7

Making the UI Pretty: Templates and CSS

In the last chapter, we went a long way toward filling in the missing functionality of our application. But, to be blunt, our application is still ugly (or at least, extremely bland):



In fact, it looks quite a bit worse than a typical web page, because these days pretty much every web page uses at least some styling. We need to do something to make it look good, make it look less like a web page and more like an application.

In this chapter, we're going to pull our application together, formatting and styling it using a combination of templates and CSS.

7.1 Introducing CSS

We need to piggyback on the work done by the people who've designed our web browsers. None of what we're going to do to make things look better is specific to App Engine. In fact, it's not really even specific to cloud programming. We're going to use standard HTML-based formatting techniques. The main difference in what a cloud application is doing is that we're using them to render a UI for an application, rather than just present a pretty web page. The techniques are the same; the goal is different. It's going to be a bit on the awkward side: HTML and CSS weren't originally designed for building UIs. The functionality that we're going to rely on was really hijacked by early cloud application builders. It does the job, but it takes some getting used to.

To understand it, you need to know about where it came from. In the early days of the Web, people used to tweak HTML to try to produce UIs. By creating elaborately structured documents of tables nested in tables nested in frames, they were able to create something that looked OK—provided you were using the right browser version on the right operating system and with the right screen size. But the resulting HTML was incredibly complicated and extremely difficult to maintain and wasn't even portable between browsers.

That strategy is clearly unmanageable. Trying to combine the content of a page with the way it should be rendered just makes a mess. What was needed was a way of separating things: to let the appearance of a page be something separate from the content of the page. The solution was something called *Cascading Style Sheets (CSS)*.

CSS allowed web developers to separate structure from appearance. HTML is used to describe the structure of a page—you mark up a page based on its structural elements: sections, paragraphs, and lists. CSS is all about appearance—it provides a way of taking the structural elements of an HTML document and describing how they should look.

CSS separates style from structure. The idea behind CSS is roughly analogous to what we did in the previous chapter by separating the application logic from the UI: CSS allows us to separate the structure of the document from its appearance. The structure is written in HTML, and the appearance is described using CSS. There are three good reasons for making this separation:

Separation of concerns.
: Separation of concerns is a fancy term for a general software engineering principle: entangling different concepts in one piece of

code always makes things difficult. As I described above, trying to interleave the structure of an HTML document with the way that it's displayed created nonportable, unmaintainable messes. This is a principle that we'll keep coming back to: separating the application logic from the page rendering in the last chapter, separating structure from appearance in this chapter, and separating rendering the basic UI from the data in the next chapter.

Reusability.
A style isn't usually specific to a single web page. All of the pages on a site, or all of the views that make up an application, are usually styled in the same way. Separating the style information lets you write it once and then reuse it in all of your pages instead of generating the same boilerplate style information over and over again. Each new page just needs to include one line specifying the CSS document that describes its style.

Flexibility.
Users may want to change attributes of the style of your page. For example, users with visual impairments may want to switch to a larger, easier-to-read font or increase the color contrast between different elements. When the style sheet is kept separate from the document, it's easier for the users to tell their browser to replace it with their own specific style.

7.2 Styling Text Using CSS

CSS is based on a very simple concept: you specify a structural element, called a *selector*, and for that selector, you specify a list of property/value pairs. For example, if we wanted to make our chat room's background blue and underline the text in the headers, we could use the CSS below:

Download css-chat/snippets.css

```
body {
        background-color: #8888FF;
}

h1 {
        text-decoration: underline;
}
```

This code includes two CSS statements. The first uses the selector body so it will apply to the entire content of the HTML document. Inside the

style, it specifies one property, background-color, and defines the color value using a hexidecimal RGB code for a medium blue.

The second statement is more specific. Its selector is h1, so it only applies to top-level headers. It specifies the text-decoration property, which is used to embellish text with things such as shadows, underlines,and strikethroughs. This statement specifies that level one headers should be underlined.

To apply this to our chat pages, we need to save it in a file and then add a snippet to our HTML page template to tell the user's browser to retrieve the CSS file and use it to render the page. If we put the CSS into a file named chat-style.css, we could apply the style to the page by adding a stylesheet link to the head part of the HTML document, as in the following:

Download css-chat/snippets.html

```
<head>
  <title>{{ title }}</title>
  <link rel="stylesheet" media="screen"
        type="text/css" href="chat-style.css"/>
</head>
```

With these basics in mind, let's build a version of our chat room using styles. Along the way, we'll start to see how to use CSS selectors in a more flexible way.

Let's start simple. Make the background of the entire page dark blue. The welcome header will be a large, attractive font in white text with a medium-blue background:

Download css-chat/snippets.css

```
h1 {
  font-family: 16px Helvetica, sans-serif;
  color: #FFFFFF;
  background-color: #0000A0;
}
```

The only new thing in this step is how we specify fonts. Fonts are a bit complicated because different browsers on different operating systems have varying sets of available fonts. Instead of specifying a single font, we can specify a series of fonts in order of preference. If the first font is available on the browser rendering the page, it will use that font. If not, it will try the next, and so on. If none of the fonts listed in our style file are available in the browser, it will use its own default. What our code does is say that the font should be rendered in 16 point size, and

that our font preference is Helvetica; if Helvetica isn't available, then the browser should use its default sans-serif font.

We also specified the background-color property for both the *<body>* element and the *<h1>* element. That's an extremely simple use of cascading. Styles, both from multiple style sheets and from multiple statements within the style sheets, follow a set of rules about how to resolve things when there are multiple declarations. The complete rules are fairly complex, but the general idea is this: the most specific CSS declarations always take precedence. CSS in the HTML file takes precedence over a linked style sheet; a page-specific linked style sheet will take preference over a site-default style sheet, and a style property set on a nested element will take preference over the same style property set on an enclosing element. So the background of the h1 element will take precedence over the background of the body element.

Next, we want to update the navbar. It's going to be on lots of pages, so we don't want it to take up much space, but we want it to be noticeable. We'll make it have small black text with a white background. To do this, we'll need to change both the HTML and the CSS. For the HTML, we'll just change the navbar block to the following:

Download css-chat/snippets.html

```
{% block navbar %}
<p id="navbar">
{% for c in chats %}
<a href="{{ c.url }}">c.name</a>
{% endfor %}
</p>
{% endblock %}
```

We changed the navbar so that it's a horizontal bar of chat room names; we also wrapped it in an annotated *<p>* tag. The tag includes the attribute id=. IDs are one of the mechanisms added to HTML specifically for working with CSS. They allow us to write a CSS rule that affects one specific element in an HTML document. Since our navbar is unique—we know that there will only ever be one navbar in a chat room—we can use an identifier for it. Then we can style the navbar using a selector #navbar. It doesn't matter what element that ID is attached to: it could be any element in an HTML document. So we can change the navbar from a paragraph to a div or a span or a list—and we don't need to change the CSS at all. This is the CSS for styling our navbar:

Download css-chat/chat.css

```
#navbar {
        font-family: 8px Helvetica, sans-serif;
        color: #000000;
        background-color: FFFFFF;
}
```

The CSS here is very straightforward. We use the #navbar ID selector and declare the appropriate properties.

Now let's get to the rendering of the actual chat. In the chat, each message is rendered as a paragraph using a <p> tag. We want some <p> tags to be rendered as messages from other users and some to be rendered as messages from the user viewing the page, and we want all of the message paragraphs to be rendered differently from other paragraphs on the page.

Once again, HTML comes to the rescue. There's a way to mark tags on the page that allows us to specify certain instances of a particular XML tag that should be rendered in a particular way. You can annotate any HTML tag with a class= attribute, and then you can write selectors that apply to *any* tag that declares a particular class= value or to a specific tag (like <p>) that declares a class. In our chat application, we'll do the specific one first. The CSS styles can specify a style for unmarked <p> tags and a special style for marked tags. According to the usual CSS "most specific rule," the style for the specific tags will overrule the style for the generic tags. So we'll write two styles for the two types of chat messages. For messages sent by the user viewing the page, we'll draw the text in plain white. For messages sent by anyone else, we'll draw the messages in yellow with a darker background. The CSS to do that is shown below:

Download css-chat/chat.css

```
p.sentbyme {
        color: #FFFFFF;
        font-family: 9px Helvetica, sans-serif;
}

p.sentbyother {
        color: #DDDDFF;
        font-family: 10px Helvetica, sans-serif;
        background-color: #000080;
}
```

As you can see, the selector for a <p> with the attribute class= set to sentbyme is written p.sentbyme.

To use that CSS, we need to modify either the HTML page template or the Python application logic in order to cause it to attach different classes to messages sent by the user viewing the page versus messages sent by other users. We make this decision by asking what's being changed. Is this a change in the application logic or just in the application's appearance?

In this case it's just appearance, so we'll do it with the template. We'll render the chat page using a template extension that includes a conditional test to decide which class to use for each message. The template is shown below:

Download css-chat/distinct-messages.html

```
{% extends master.html %}

{% block pagecontent %}

{% for m in msg_list %}

❶   {% ifequal msg.sender m.user %}
    <p class="sentbyme">
    {% else %}
    <p class="sentbyother">
    {% endifequal %}

     <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
        {{ m.message | escape }} </p>

{% endfor %}

      <form action="/talk&chat={{ chat }}" method="post">
        <p><b>Message</b></p>
        <div><textarea name="message" rows="5" cols="60"></textarea></div>
        <div><input type="submit" value="Send ChatMessage"/></div>
      </form>

{% endblock %}
```

At ❶, we use a new bit of Django templates. The ifequal tag takes two variables and checks to see if, according to Python, they're equal. If so, it outputs the HTML text between the ifequal and the else; otherwise, it outputs the text between the else and the end of the ifequal block. So our code checks to see if the sender of a message is the same as the name of the user viewing the page. If they are the same, it generates a <*p*> tag specifying the sentbyme class; otherwise, it generates a <*p*> tag specifying the sentbyother class.

We can often dispense with the tag names in the CSS. We can specify a style rule that applies to *any* tag with a particular class. If we were to leave out the p in the style rules above so that the selectors were just .sentbyme and .sentbyother, those style rules would apply to any tag that had the class. That can be a very useful thing for two reasons:

1. There are properties that we might want to use in many places in our document. For example, we might want to be able to flag parts of our page that describe errors. The way that we'd normally want them to appear would be in bright red. So we could create a class error. Then anywhere in our output where there was error text—whether it was a header, a paragraph, or a small section of boldface text—we could add class="error" to the tag.

2. When building an application, we can experiment with different layouts. For example, we started with our navigation bar as a bulleted list and then changed it to a horizontal list. We can set decorative properties of the user interface elements (such as colors and font styles) using a class, and then when we change the way that we write them in HTML, we won't need to change our CSS.

7.3 Page Layouts Using CSS

We've seen how CSS makes things look more visually attractive by controlling fonts, colors, and decorations. But for building a user interface, we're still missing something incredibly important—layout. To make a UI that is both attractive and usable, we need to control where things are on our screen. Leaving the layout of our user interface up to the layout engine of the user's browser really isn't an acceptable option. Browsers render simple HTML to be good for reading a web page, not for automatically producing polished user interfaces.

Our user interface is composed of a collection of boxes. A mockup of what we'd like it to look like is shown in Figure 7.1, on the following page. It's got a welcome header, which is a box the full width of the screen; a navigation bar running vertically up the left side of the screen; an area showing the chat transcript; and, beneath the transcript, the entry form. Each of those elements are, basically, a rectangular region, and we want them to be laid out in a specific way.

We must define what the boxes are and how they should be laid out. From what we've done so far, you can probably guess what's coming: there's an HTML element for describing what's in a box (the structure) and CSS properties for describing how to lay it out (the appearance).

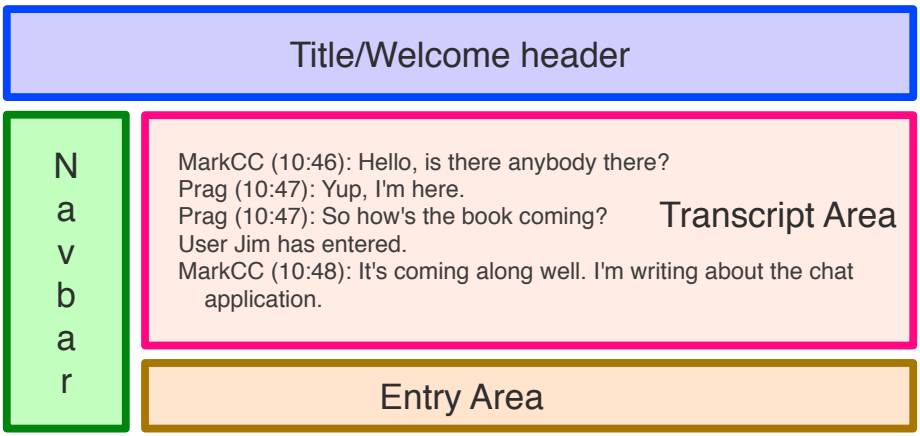


Figure 7.1: The block structure of the chat user interface

Structuring Documents with div Elements

The HTML <*div*> element can contain any collection of other HTML tags and elements, including other <*div*>s. Its only purpose is to describe a structure made up of a collection of other elements. Without CSS altering layout and appearance, you can't even see where the <*div*>s are in a document—by default, they have no visual properties at all. They just produce a box that you can then reference via CSS selectors.

Let's update our basic page template so it's structured using <*div*> elements:

Download css-chat/master.html

```
<html>
 <head>
  <title>{{ title }}</title>
  <link rel="stylesheet" media="screen"
        type="text/css"
                href="chat-style-layout.css"/>
 </head>

 <body>

<div id="header-block">
    <h1>Welcome to {{ title }} </h1>
    <p> Current time is {% now "F j Y H:i" %}</p>
</div>
```

```
<div id="navbar-block">
  {% block navbar %}
    <div id="navbar">
      <ul>
        {% for c in chats %}
        <li><a href="{{ c.url }}">c.name</a></li>       {% end for %}
      </ul>
    </div>
    {% endblock %}
</div>

<div id="content">
  {% block pagecontent %}

    <p> This is template text. If you're seeing this in a page rendered
      by chat, something is wrong.</p>


    {% endblock %}
  </body>
</div>

<div id="entry-form">
  {% block entry %}
  {% endblock %}
</div>

</html>
```

This is really just the master template from the last chapter, except that we took each of the blocks from our UI sketch in Figure 7.1, on the previous page, surrounded the HTML for that part of the UI in a <*div*> element, and tagged it with an id= attribute.

Flow-Based Layout

We have the HTML for our applications view structured into a collection of boxes. Now we need to write the CSS to lay it out. CSS provides a huge number of properties that we can use to manage the way that the page gets laid out to make it look like a natural user interface.

The catch (and there's always a catch, isn't there?) is that layout is very complicated. It's just the nature of the beast: we're trying to describe how to lay things out in browser windows that could be any size, on computer displays with different resolutions, and using different layout engines in different browsers.

We describe layout using *flow constraints*. The basic idea is that without CSS layout properties, browsers lay the elements of an HTML page out

by flowing text into the page. An HTML page starts with a rectangle representing a blank line. Text is flowed into that line until it's full. Once it fills up, a new line is created beneath it, and then the text is flowed into that box. So we wind up with the page laid out by flow text into rectangular regions of the screen: left to right and then top to bottom. To alter layout, we jump into that process. For example, we can interrupt it by positioning a box in a specific position on the screen, and the rest of the page content will flow around it.

The way we describe how to lay things out on the screen is going to be based on flow. Our HTML can't just be a collection of * *s in any old order, which we'll then position on the screen using layouts—the order in which things appear is going to have a major effect on where they're going to wind up and what our user interface is going to end up looking like.

CSS gives us two real tools for building a layout: floats and clears.

Floats

A *float* is a * * (or other HTML element) whose CSS style includes a float attribute. In layout, floats (true to their name) float between the sides of the page. The way that they work is very simple: the float gets a vertical position by following the standard flow layout. But then, instead of sitting wherever it landed, it gets floated to the side, and other elements can then be flowed around it.

That's pretty confusing, so let's look at an example. Let's put together a little mockup of our chat UI and see what it looks like without using floats, and then we'll add the floats and see how it changes. We'll do mockups of a navigation sidebar and a chat view area. Here's the HTML:

Download css-chat/flow-mockup.html

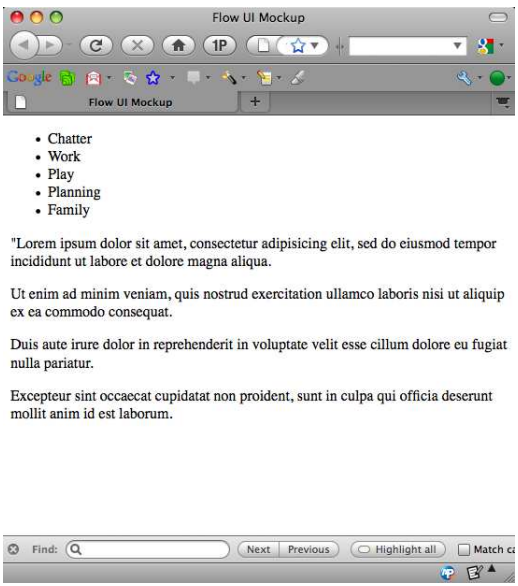
```
<html>
  <head>
    <title>Flow UI Mockup</title>
    <link rel="stylesheet" media="screen"
          type="text/css" href="flow-mockup.css"/>
  </head>

  <body>
    <div id="sidebar">
      <ul>
        <li>Chatter</li>
        <li>Work</li>
        <li>Play</li>
```

```
        <li>Planning</li>
        <li>Family</li>
      </ul>
    </div>

    <div id="body">
      <p>"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
        eiusmod tempor incididunt ut labore et dolore magna aliqua. </p>
      <p> Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat. </p>
      <p> Duis aute irure dolor in reprehenderit in voluptate
        velit esse cillum dolore eu fugiat nulla pariatur. </p>
      <p> Excepteur sint occaecat cupidatat non proident, sunt
        in culpa qui officia deserunt mollit anim id
        est laborum.</p>
    </div>
  </body>
</html>
```

It generates a page that looks like this:



We can add make the sidebar float using CSS:

Download css-chat/flow-mockup.css

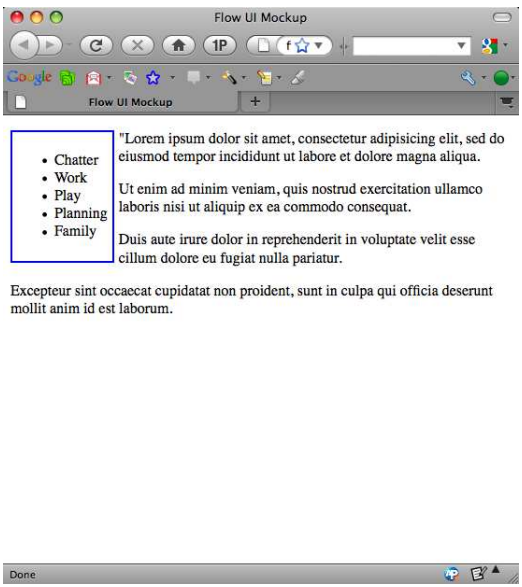
```
#sidebar {
❶       float: left;
❷       border: 2px solid #0000FF;
❸       padding: 5px;
        margin-right: 5px;
}
```

❶ The float property can be set to either left, right, or none. left, as in our example, means that the element should be floated to the left; right means that the element should float over to the right; none allows you to "unfloat" something that inherited a float property from its class.

❷ The border property lets you draw a border around the edges of that <*div*>. It's got the format "width style color." style describes what the border outline should look like: it can be solid, dotted, dashed, double, grooved, ridge, inset, or outset. For this, we'll use a simple solid outline.

❸ Small amounts of can make things look better. CSS allows you to use two kinds of space around things: margins and padding. Margins are space added outside of a <*div*> box; padding is space added inside the box.

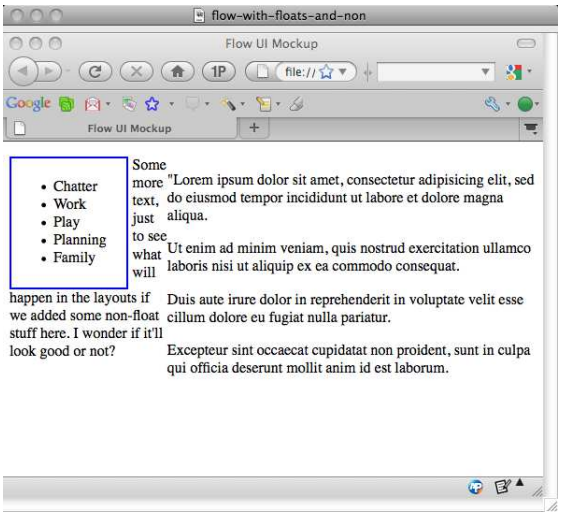
The result of using this style is:



Clears

As you can see in the screenshot, the text of the body block flows around the sidebar. It appears next to the sidebar, up to the point where the sidebar ends, and then it flows to the left margin.

We could prevent the transcript text from flowing around the sidebar by making it into a float as well. If we did that naively—just setting float: right;—what we'd get would look like this:



By default, floats are kept separate, and when it comes to positioning them, each float is treated as if it were the full width of the page. Since the transcript is a float that floats right and the navbar is a float that floats left and they're both treated as full-page-width floats in the flow layout, they won't appear side by side; they'll be vertically stacked in the layout. What we need to do in order to get them to position nicely is to give them widths. Since we're doing this inside a web browser, the page-width is variable—so we'll usually describe positions and widths using percentages. We can use absolute measures, but most of the time it's better to use relative ones. In order to position the elements in our mockup the way we want, we'll need to set widths. Here's the updated stylesheet:

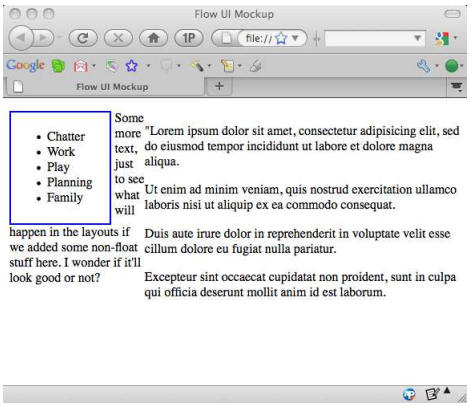
Download css-chat/flow-twocol.css

```
#sidebar {
        float: left;
        border: 2px solid #0000FF;
        padding: 5px;
        margin-right: 5px;
        width: 20%;
}

#body {
        float: right;
        width: 70%;
}

p.allclear {
        clear: both;
}
```

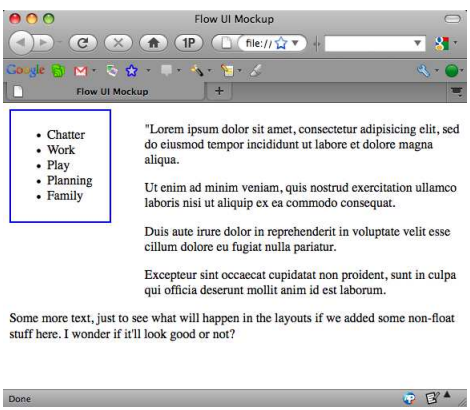
and the corresponding page:



This looks good for what it is, but if we tried to add more to the UI, such as the entry block at the bottom—as in our mockup—it would end up flowing around the two floats. If the navbar was shorter than the transcript, other things would flow to the left around it, just as in our original mockup.

What we need is a way of saying, “Don’t flow around this.” That’s what a *clear* is for. If we add clear: both; to our CSS for any element, we’ll get what we wanted. You can say clear: left; which will clear left-floats; right; which will clear right-floats, or both; which will clear all floats.

Let’s add a class allclear to the <*p*> tag for the additional text and add p.allclear { clear: both; } to our style sheet. The result is in shown here, exactly as we wanted:



7.4 Building Our Interface Using Flowed Layout

Now we have a basic idea of how to put our interface together. We just need to polish it a bit and write the CSS to make it look the way we want. Once we understand how CSS works, getting a respectable-looking interface is pretty easy—and getting from that to something that's really terrific isn't difficult, but it's time-consuming. We will want to spend some time tweaking various options, shifting margins, changing colors, rearranging elements, and so on. It takes time, but it's worth the effort: the difference between something that looks terrific, like Gmail, and something that looks mediocre (like the current version of our application) is only the amount of time spent polishing the CSS.

Getting back to our chat app, we've got four basic elements to our interface. There's a welcome bar at the top of the application. Beneath that, there's a navigation bar over to the left, and beside it, there's the chat transcript. Beneath those, taking up the full width of the window, is a pane containing the new-message entry form. Here's the CSS:

Download css-chat/app.css

```
body {
  background-color: #8888FF;
}

#header-block {
  font-family: 16px Helvetica, sans-serif;
  color: #FFFFFF;
  background-color: #0000A0;
  border: 2px ridge #0000F0;
}

#navbar-block {
  float: left;
  width: 20%;
  font-family: 8px Helvetica, sans-serif;
  color: #000000;
  background-color: FFFFFF;
  border: 2px ridge #0000F0;
  padding: 4px;
  margin-right: 4px;
}

#transcript-block {
  padding: 4px;
  float: right;
  width: 75%;
  font-family: 8px Helvetica, sans-serif;
  background-color: 444444;
```

```
  color: #FFFFFF;
  border: 2px ridge #0000F0;
}

#entry-block {
  clear: both;
  border: 2px ridge #0000F0;
  margin-top: 4px;
}

p.sentbyme {
  color: #FFFFFF;
  font-family: 9px Helvetica, sans-serif;
}

p.sentbyother {
  color: #DDDDFF;
  font-family: 10px Helvetica, sans-serif;
  background-color: #000080;
}
```

There's nothing complicated here: we just put the things we've been talking about together. We're using colors, padding, borders, floats, and clears to create the user interface structure that we want. We can test the CSS by using a fake file—that is, a simple HTML file that follows the same structure as the pages that will be generated by our application's templates. The fake file contains these elements:

Download css-chat/fake-ui.html

```
<html>
  <head>
    <title>MarkCC's AppEngine Chatroom</title>
    <link rel="stylesheet" media="screen" type="text/css" href="app.css"/>
  </head>

  <body>
    <div id="header-block">
      <h1>Welcome to MarkCC's AppEngine Chatroom </h1>
      <p> Current time is March 13, 2011 8:39</p>
    </div>

    <div id="navbar-block">
      <p><b>CHATS</b></p>
      <ul>
        <li>Work</li>
        <li>Play</li>
        <li>Write</li>
        <li>Misc</li>
      </ul>
    </div>
```

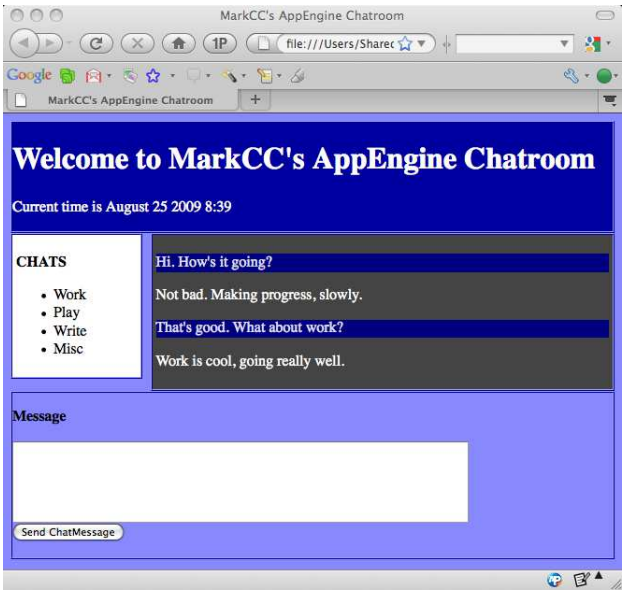


Figure 7.2: Faked chat interface for testing CSS

```
    <div id="transcript-block">
      <p class="sentbyother">Hi. How's it going?</p>
      <p class="sentbyme">Not bad. Making progress, slowly.</p>
      <p class="sentbyother">That's good. What about work?</p>
      <p class="sentbyme">Work is cool, going really well.</p>
    </div>

    <div id="entry-block">
      <form action="/talk&chat=thischat" method="post">
        <p><b>Message</b></p>
        <div><textarea name="message" rows="5" cols="60"></textarea></div>
        <div><input type="submit" value="Send ChatMessage"/></div>
      </form>
    </div>
  </body>
</html>
```

The rendered result is shown in Figure 7.2. Once we're happy with how the faked one looks, we can hook it up to the live application by adding the stylesheet link line to the master template and uploading it to App Engine.

7.5 Including CSS Files in App Engine Applications

There's a bit of trickiness to using CSS in App Engine Python code. App Engine draws a distinction between executable code (dynamic content), and data (static content). CSS isn't executable: in the context of an App Engine cloud application, it's just a static data file. So you can't just reference it the way that you reference your templates. Your application will generate a HTML page containing a <*link*> tag referencing it, but then it's up to the client's browser to send a request for the CSS.

Since it's a separate HTTP request, we need to tell App Engine how to handle the request. For requests that were handled by our program, we put url: entries into the app.yaml file. We do exactly the same thing for CSS. I like to put CSS files into their own directory, so a typical app.yaml file for me looks like this:

Download interactive/app.yaml

```
application: markcc-chatroom-one
version: inter
runtime: python
api_version: 1

handlers:
- url: /(.*\.css)
  static_files: static/\1
  upload: static/(.*\.css)

- url: /(.*\.js)
  static_files: js/\1
  upload: js/(.*\.js)

- url: /.*
  script: chat.py
```

This file has a new entry for anything ending in .css. Any request for x.css will be remapped to static/x.css. We'll see this in action in the next chapter. We just put a CSS link to app.css in our application and then put the actual CSS file in static/app.css.

As we've seen in this chapter, styling an App Engine application to make it look good is a big job. We need to interact with the browser layout algorithm to get things positioned the way we want, but with the capabilities given to us by HTML and CSS, we've been able to get to the point where our chat application looks good.

In the last couple of chapters, we've been going through a process of separation of concerns: we've separated storage from application state,

rendering from application logic, and now page structure from appearance. In the next chapter, we'll continue this process using a web technology called AJAX to separate out user interface controls from the other aspects of our program, and in the process we'll make our application behave more like a traditional desktop application. Users won't need to do things like hit the Refresh button to see new chat messages—our interface control layer will take care of that automatically.

References and Resources

CSS: the Definitive Guide http://oreilly.com/catalog/9781565926226
An excellent book that provides a detailed description of CSS, selectors, style attributes, and all of the other things that you'll need.

The Art and Science of CSS. . .
. . . http://pragprog.com/titles/stp-ascss/the-art-science-of-css
Another textbook providing a very nice guide to how to use CSS to produce visually pleasing web applications.

CSS Tutorial http://www.w3schools.com/css/
An online CSS tutorial with interactive testing of different CSS styles and attributes.

Layout Gala. .....http://blog.html.it/layoutgala/
An online resource containing HTML and CSS templates for thirty different web element layouts.

Chapter 8

Getting Interactive

In this chapter, we're going to make a huge leap forward in terms of how our application behaves. Instead of relying on the user to constantly hit Refresh to see new messages, we're going to make our chat service interactive. In order to do that, we'll learn about the following:

- JavaScript, the programming language embedded right in your web browser. You can create interactive elements in your interface by embedding JavaScript programs in an application page.
- The document object model, which represents HTML as a programmable object, allowing you to change parts of your UI using web page content that you can alter using JavaScript programs.
- AJAX (asynchronous JavaScript and XML), a technique that lets you send commands and requests to a server without reloading a page, allowing you to handle actions and updates in your interface without having to do reloads.
- The model-view-controller (MVC) paradigm, a powerful standard architecture for separating the components of an interactive application.

8.1 Interactive Web Services: The Basics

So far, our chat room has been frustratingly static. When we look at a chat in our browser, we see only the messages posted the last time the user manually loaded the page, either by posting new messages or by clicking the Refresh button in the browser. That's not really how we expect applications to work. We expect them to be dynamic—constantly updating themselves as the data underlying them is updated. In a chat

room, we expect to see chat messages posted by other users as soon as they post them—we don't expect to have to manually ask the application if there are any new messages!

The problem is we don't have any way to make things interactive. We built our application using a client-server request-response model. In fact, we didn't really *build* a client at all. We implemented a server and used the prepackaged functionality of the web browser as a client. In the current version of our cloud-programming model, the browser is the client, and all it does is render content. If we want to make our cloud service work like a real application, we need to add code that runs on the client. The key to building an interactive UI is a technology called AJAX, which lets us use JavaScript code that runs inside the browser, providing rich client functionality.

How does it work? First, modern browsers include an interpreter for JavaScript, a programming language that can be embedded in HTML. With JavaScript, you can embed programs in your application's pages that respond to user actions. So when a user clicks on a button, you can handle it immediately in the browser.

Second, your JavaScript program can manipulate the entire HTML page or any portion of it as an object, called a *DOM* (document object model) object. When you change the DOM object for your page, that immediately changes what the user sees in their browser. With JavaScript and the DOM object for your application's pages, you can create applications that immediately respond to user actions.

Enough theory—let's try a bit of programming. We're not going to do anything too fancy yet. In the last chapter, we changed the display of chat messages to distinguish between different messages. What we're going to do now is add a button that turns that on and off. When the users first load the application, all of the messages in the chat transcript will be displayed in the same font. When they click the button, the display will change so that messages sent by other users are highlighted.

We're going to handle this step with CSS classes. We'll place a CSS class on messages from other users to distinguish them from messages sent by the user viewing the page. When the user clicks the color-change button, we'll walk through the DOM for the chat messages and change attributes of those messages to make them appear differently. We'll have three different CSS classes for messages: self, other, and other-

colored. When the user first loads the page, all of the chat messages will be tagged with either self or other. When the user clicks the Highlight button, we'll go through and change all of the other messages to other-colored. In order to be able to recognize the messages that we need to change, we'll tag them with a name= attribute. Then we'll be able to change the styling of messages by searching for elements with the attribute and changing their class= attribute.

Before we put things together, let's start with the JavaScript, which will find the chat section of the page, and then walk through that structure looking for <*p*> tags that have the other class. To make it easy to find the appropriate section, we'll attach an ID to the <*div*> for the chat transcript section.

Download interactive/twiddle.js

```
❶ <script type="text/javascript">
  function highlightMessages() {
❷   var chatBlock = document.getElementById("chat-transcript");
❸   var chatMessages = doc.getElementsByName("other");
    for (c in chatMessages) {
      // Change the class attribute to be "other-highlight"
      for (a in c.attributes) {
        if (a.name == "class") {
❹         a.value = "other-highlight";
          break;
        }
      }
    }
  }
  </script>

❺ <input type="button" value="Highlight Others"
         id="HighlightButton" onclick="highlightMessages()"/>
```

❶ Embed JavaScript code in an HTML page by putting it inside a <*script*> tag.

❷ To change the styling of other users' messages, we must find the elements whose class= tag needs to be changed. We'll start by finding the <*div*> element. In JavaScript, we can always access the DOM object for the document being displayed using the global variable document. The DOM provides lots of methods for finding and manipulating the elements of the page. Right now, we'll use getElementById, which returns us the DOM object for the element that's tagged with an id= element that has the value chat-transcript.

❸ Next, we want to find the set of messages nested in the transcript with their name=; attribute set to the value other: the set of messages sent by other users. Again, the DOM provides a method to do exactly what we want—getElementsByName—which returns an array of the elements that have a particular value for their name attribute.

❹ Now, finally, we can update the class= attribute to highlight the messages. For each element in the array of messages to update, we need to find its class= element and then update it. To change something in the DOM, we just alter the properties of the objects directly—we can update the class= attribute just by assigning a new value to its value property.

❺ Now all we need to do is hook the JavaScript function into the UI. Create a button element in the *<form>* part of the page, and connect the function to the button by putting a call into the onclick= attribute of the button.

On the server side, we need to make a very small change. Up to now, we haven't been applying name attributes to messages in the transcript. Doing that just requires a one-line change to the template.

8.2 The Model-View-Controller Design Pattern

When we started our chat application, the code was pretty simple: we had a couple of message handlers in Python, and those handlers printed out HTML for the interface. As we expanded the chat application and made it more powerful, flexible, and attractive, we've added new techniques and languages for managing the complexity of the various bits and pieces of our program. But now we're getting to the point where the number of bits and pieces is starting to get confusing.

Now, our application consists of the following:

- Server-side message handlers written in Python
- Templates for HTML pages written using Django
- User interface interaction code written in JavaScript
- User interface layout management written in CSS

All of this has been manageable so far because our code has always followed a basic structure. We've built everything by focusing the struc-

ture of our code around individual HTML pages. We wrote request handlers, and in each request handler, we generated a full HTML page for rendering on the user's browser. That's given us an organized architecture for our application.

But now we're about to take another step toward making our application even more interactive by using AJAX. With AJAX, we're no longer going to be writing code in which each page is generated completely by one handler. In order to keep things organized and maintainable, we must think about the architecture of our system: we need a disciplined way of organizing all of the pieces that combine to become our cloud application.

Fortunately, cloud applications are well suited to one of the oldest and most powerful design patterns for user interfaces—model-view-controller.

If you hark back to the early days of graphical user interfaces, you'll wind up looking at Smalltalk, which is where most modern GUI ideas got their start. Windows, buttons, mice, and menus all came from Smalltalk. In Smalltalk, programmers built their interfaces using a three-part design pattern. More than thirty years later we're still using that design, and it's an almost perfect match for the way that we'll build user interfaces for cloud applications. The structure of an MVC application is illustrated in Figure 8.1, on the next page.

In MVC, the interface has three components:

The model. The application logic of the system. The model is implemented around the basic concepts of the data and operations that the application is intended to perform. In the model, you don't really consider the interface at all: the model is completely separate and only works in terms of the underlying application data. For cloud applications, the model is the server code. The server can't really talk to the client; it needs to wait for the client to send it requests. It's totally decoupled from the user interface.

The view. The view is the visible user interface: the display elements, entry boxes, and so on. In a cloud application, the view is written in HTML. It's got no real behavior of its own: it's just a bunch of display elements and the code (in HTML and CSS) that's needed to render it in an attractive way. The view really doesn't care about the application logic or data: it's capable of rendering things when it's told to.

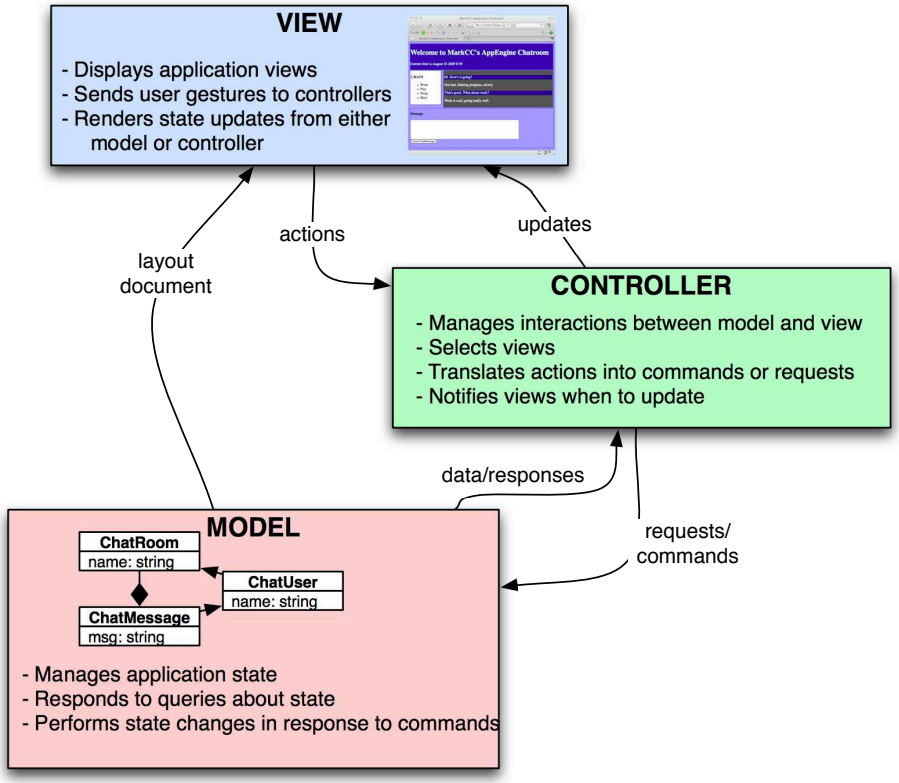


Figure 8.1: The structure of a cloud MVC application

The controller. The controller is the bridge between the client and the server. It's the fundamental piece that we've been missing so far. The controller takes interface actions that are produced in the view and translates those into operations that can be performed in the model, and it takes content data produced by server and translates that data into a form that the view can render. In the cloud, the controller is the JavaScript code that's executed inside the user's browser.

MVC is a natural way for us to build cloud applications because the concepts of the MVC model fit well with the set of pieces we'll build for our applications. The different languages we need to use to build a good cloud app force us to separate the application into at least three components: server code (in Python or Java), rendering code (in HTML

and CSS), and interaction code (in JavaScript). MVC provides us with an easy way to understand what functionality belongs in which components. In addition, it brings with it a long history of designing applications using those three basic components, which we can take advantage of in designing our cloud applications. Of course, the cloud is different, so the architecture isn't *exactly* the same as the old Smalltalk UIs, but the basic idea holds. The main difference is that in cloud MVC, there's less direct contact between the model and the view. In classic MVC, when data in the model is modified, the model sends updates directly to the view. In cloud MVC, once the view is created, the model doesn't get to talk to it directly, but instead it needs to send all of its updates to the controller, which then updates the view.

If you think about it, MVC is really nothing but the next step in the process we've been following all along. As our application has gotten more complicated, we've been decoupling things—separating components. We've separated rendering from computation by putting the rendering in templates and the computation in Python. We've separated style from content by putting the content in HTML and the style in CSS. And now we're just adding another layer of separation in exactly the same fashion. Our client is going to have both a user interface (content and style) and executable code: we're separating those into the view (in HTML and CSS) and the controller (JavaScript).

As we continue to develop our chat application, we're going to keep MVC in mind. It's the best basic structure to describe the way a cloud application can be divided into logical pieces. That's what we did earlier in this chapter: we separated chat into the model (the server, with its datastore storing the chats, and query handlers, which manage interaction with clients), the view (the HTML and CSS documents that provided the basic UI), and the controller (the JavaScript that did all of the interaction).

8.3 Talking to the Server without Disruption

At the beginning of this chapter, we used JavaScript to create dynamic interactions in the user interface without needing to talk to the server. That's useful, but it's also very limiting: we can only build features that use data in the HTML document being displayed by the user's browser. We can't do anything that needs more data.

Of course, we want to be able to do things that rely on getting data from the server. We can get data from the server with what we know, but always within the standard synchronous cycle of client-server interaction of the browser. We write client code that sends requests to the server, the server responds to those requests, and when the client's browser receives the response, it renders it. Without doing something special, we can't avoid that basic request-response-render process.

The request-response-render cycle really isn't enough. We want to write our chat application so that it automatically updates the display whenever there's a new chat message. But our chat application waits for the user to do a manual refresh. We could write a piece of JavaScript with a timer, which automatically refreshes once a second. But every time that refresh happened, it would be very disruptive to the users. Their interfaces would go blank and then redraw. If they started typing a new message, that message would be lost when the browser refreshed. That's absolutely not what we want.

What we need to do is talk to the server asynchronously. That is, we need to write some code that runs separately from the usual request-response-render cycle. This technique is called *AJAX*: asynchronous JavaScript and XML. It uses a JavaScript construct called an XMLHttpRequest. XMLHttpRequest is, frankly, rather an ugly hack. Like a lot of the tools that we use for programming the client-side of a web-based cloud app, it's basically an ad hoc thing that someone slapped together because they really needed it, rather than something carefully designed as part of a toolkit for building UIs. Once people starting using it, it became a standard, and now we're pretty much stuck with it. (As we'll see in Section 9.3, *RPC in GWT*, on page 135, the AJAX can be wrapped up and hidden under the covers so that you don't need to deal with the ugliness, but it's always best to understand what's really going on.)

The key to XMLHttpRequest is that it's asynchronous, which means when a request is sent, the sender doesn't wait for a response. Normally, when we write code that makes HTTP requests, we do it with a function call: we call a method to send the request, and the return value of that function call is the response to the query. The code *blocks*—it does nothing but wait until it gets a response. But an XMLHttpRequest doesn't block. Instead, when we create an XMLHttpRequest, one of the parameters is a function called a *callback*. When the client's browser receives a response to the request, it invokes the callback function. So the client never blocks—it just continues along, and the user gets a

XMLHttpRequest Is Not an XML HTTP Request

XMLHttpRequest is incredibly poorly named. In reality, it's not a request, and it's got nothing to do with XML.

XMLHttpRequest is a request *manager*—an object used to create requests, send those requests, and handle the responses it receives.

The most important thing about XMLHttpRequest is that it's capable of generating and sending a request outside of the normal browser request-response-render cycle, and it can handle the requests asynchronously, so your program doesn't need to wait for a response.

very smooth user interface without any interruptions or refreshes. As soon as a response is received, the callback is invoked, and the user interface is updated with the latest information.

This description sounds very abstract. But it's really not difficult with App Engine, and the best way to show that is by writing code. We'll go straight to what we really want—a dynamically updated chat view.

To do that, we need to write a bunch of pieces:

1. A server request handler that serves the user interface frame without any chat data
2. A server request handler for fetching chat data (new messages)
3. A JavaScript component that requests updates from the server and then adds messages to the transcript whenever it receives new data.

The Model: Chat's Request Handlers

The first thing we need to do is build a request handler on the server. We're going to split our request handler into two pieces by separating the handler that sends the user interface to the client from the handler that sends the data to the client. The first handler, which sends the UI to the client, basically sends the view and the controller to the browser, so that it can run as the client for our application. Then the controller will send requests for data to the server, which will be sent by the second handler.

The basic request handler is trivial. We'll take the same template we've been using and create a specialization of it that emits a blank transcript area. You can see the template in Section 8.3, *The Chat View*, on page 120. The handler will just be the same handler we've been using, only with that template.

Download interactive/chat.py

```
class InterfaceServerHandler(webapp.RequestHandler):
  def get(self):
❶   requested_chat = self.request.get("chat", default_value="none")
❷   if requested_chat == "none" or requested_chat not in CHATS:
      template_params = {
          'title': "Error! Requested chat not found!",
          'chatname': requested_chat,
          'chats': CHATS
          }
      error_template = os.path.join(os.path.dirname(__file__), 'error.html')
      page = template.render(error_template, template_params)
      self.response.out.write(page)
    else:

       messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                              "ORDER BY timestamp", requested_chat)
       template_params = {
          'title': "MarkCC's AppEngine Chat Room",
          'msg_list': messages,
          'chat': requested_chat,
          'chats': CHATS
          }
       path = os.path.join(os.path.dirname(__file__), 'interface.html')
       page = template.render(path, template_params)
       self.response.out.write(page)
```

For the data, we're going to do something a bit differently. Up to now, we've had the client operating in a *stateless* mode—that is, the client has never sent requests to the server that are based on anything that it remembers. But in interactive operation, as the client runs, it's going to keep sending requests to the server, and we want to send it only new messages—that is, messages that were posted after the last time that the client got data from the server. So the request is going to include a time, and the response is only going to send messages that were posted after that time.

We also need to include a time in the response sent to the client. After all, we're in the cloud: our client and our server are on different computers, possibly in different parts of the world. Their clocks might be set differently, and there's a time delay between when the server sends

a response and when the client receives it. In order to make sure that the client doesn't miss any messages, we want to know what time the *server* thought it was when it sent the last message to the client.

The code to do all that is pretty straightforward. We'll just send an XML document. The top-level element will be a our own <*ChatUpdate*> element, which will include a time= element. Inside the <*ChatUpdate*>, we'll put the HTML <*p*> tags containing the chat messages. And as usual, we'll use a Django template to generate the XML. The template is pretty straightforward:

Download interactive/update.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ChatUpdate chat="{{ chat }}" time="{{ time }}">
{% for m in msg_list %}
<p>({{ m.chat }}) {{ m.user }} ( {{ m.timestamp }} ): {{ m.message|escape }} </p>
{% endfor %}
</ChatUpdate>
```

The request handler that fills in the template is also very straightforward:

Download interactive/chat.py

```
class DataRequestHandler(webapp.RequestHandler):
    def get(self):
      requested_chat = self.request.get("chat", default_value="none")
      messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                             "ORDER BY timestamp", requested_chat).fetch(20)
      template_params = {
          'msg_list': messages,
          'chat': requested_chat,
                  'time': self.request.get("time", default_value="0"),
          }
      path = os.path.join(os.path.dirname(__file__), 'update.xml')
      page = template.render(path, template_params)
      self.response.headers["Content-Type"] = "application/xml"
      self.response.headers.add_header("Access-Control-Allow-Origin", "*")
      self.response.headers.add_header("Access-Control-Allow-Methods",
                                       "GET, POST, OPTIONS")
      self.response.out.write(page)
```

❶

That's it for the model part. As you can see, the model really doesn't need much specialization in order to work in interactive mode. The only thing you need to do is separate the process of serving the interface from the process of serving the data.

The Controller: JavaScript on the Client

In the server discussion in the previous section, I said that we'd send the JavaScript controller code with the user interface in the initial handler, but in the code, all that I showed was a script line that would include the JavaScript. Now we're going to implement the controller. Let's jump right into the code:

Download interactive/js/asynch.js

```
function SetUpAndSendRequest(time, chat) {
❶   var request = new XMLHttpRequest();
❷   var transcript = document.getElementById("chat-transcript");

❸   request.onreadystatechange = function() {
      // Wait until the request is done. Done == ready state 4.
      if (request.readyState != 4) {
        return;
      }
❹     var xmlData = request.responseXML.documentElement;
      if (xmlData != null) {
            transcript.innerHTML = "";
        messages = xmlData.getElementsByTagName("p");
❺       for (var x = 0; x < messages.length; x++) {
          transcript.innerHTML += "<p>" +
              messages[x].childNodes[0].nodeValue + "</p>";
        }
❻       newtime = xmlData.getAttribute("time");
❼       SetUpAndSendRequest(newtime, chat);
      } else {
        transcript.innerHTML +=
            "<p>Sorry, but there was an error updating this chat</p>";
      }
    }
❽   request.open("GET", "latest?time=" + time + "&chat=" + chat, true);
    request.send();
}

function init() {
  SetUpAndSendRequest(0, "book")
}

window.onload = init;
```

❶ First, we set up a few global variables that we'll need. The most important one is the XMLHttpRequest. In addition, we set up global references to the main application URL and to the part of the HTML page that contains the chat transcript.

❽ It's time to start the actual request code. The first thing we need to do is open a request. The XMLHttpRequest object isn't a request;

it's an object we use to create and send requests. To make it send a request, first open a new request, telling it what the parameters of the request are.

❸ Now we get to the heart of the asynchronous part of AJAX. We're not going to wait for a response; we provide a function the XMLHttpRequest object invokes when it gets its response. In this case, we put the callback function inline.

The main callback for XmlHttpRequest is called onreadystatechange. It is called when there is *any* change in the status of the request. There's a lot of power in being able to respond to all of the different states that a request can pass through, but that's beyond the scope of this book. There are lots of excellent books on AJAX—if you're going to do a lot of interactive cloud development, you should really read one of them! A few suggestions for good ones are in the resources at the end of the chapter.

Here, our program will do something with the result of the request when we've received the complete result. That's called ready state 4. The first thing we need to do in the callback is make sure that we're at state 4. If not, the callback returns. It will get called again the next time the state changes.

❹ When the callback function is finally invoked in ready state 4, we know we received the latest chat data from the server, so we can update the chat view. We grab the data from the response using the responseXML field of the XMLHttpRequest. All of the chat messages will be put into <*p*> tags, so we retrieve them.

❺ Update the chat transcript by appending the new messages.

❻ At this point, the user interface has been updated with the latest chat messages. But we need to reissue the XMLHttpRequest so when more messages are posted, the display updates again. We don't want to re-fetch any messages that we've already displayed. So we grab the time when the response was sent.

❼ Finally, we send the request by invoking this function, effectively giving us a loop: SetUpAndSendRequest creates and issues a request via XMLHttpRequest and registers a callback function to update the user interface when the new data is received. The callback re-invokes SetUpAndSendRequest. And so on.

That's the basic pattern—we use JavaScript to create an XMLHttpRequest, which issues the request and then sets up a callback.

The Chat View

The view for our new interactive MVC chat service is the easiest part. It's just a template extension almost identical to what we've done before:

Download interactive/interface.html

```
{% extends "master.html" %}

{% block script %}
<script src="asynch.js" language="javascript">
</script>
{% endblock %}

{% block pagecontent %}
<div id="chat-transcript">
</div>

<form action="/talk?chat={{ chat }}" method="post">
  <p><b>Message</b></p>
  <div><textarea name="message" rows="5" cols="60"></textarea></div>
  <div><input type="submit" value="Send ChatMessage"/></div>
</form>

{% endblock %}
```

❶ Our new template extension must include the JavaScript source file containing the controller code. The <*script*> tag can be used for either inline code or for code from an external source file; here, we include it from the source file.

❷ We need to create an empty <*div*> with the transcript id, which is what the JavaScript will use to insert the chat messages. The included JavaScript contains the code needed to start the interactive process, and it takes care of rendering the chat messages.

With those pieces in place, we're pretty much done. We've got a chat system in which the client is continually shaking hands with the server to get the latest chat messages and updating the transcript view with new chat messages as they're posted. We need to update our app.yaml filename to make sure that it includes all of our new template files. We also need to update the application object in Python to route requests to the right handler. We're pretty close to being done with basic chat: we've got multiple chat rooms, user login, full interactivity, and a pretty interface. We've set up a solid architecture for dividing up the parts of our system, both in terms of separating data from code and in terms of separating the model, the view, and the controller.

We're ready to move on to some more complicated things! Google App Engine doesn't limit you to writing your code in Python: you can use other languages, too. Currently, the main alternative is Java. In the next chapter, we'll look at how to use Java instead of Python for our application. Then we'll get into some advanced topics like advanced data storage, security, and debugging.

8.4 References and Resources

XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest/
The official W3C standards document describing the JavaScript XMLHttpRequest.

AJAX Tutorial. http://www.w3schools.com/Ajax/Default.Asp
An online interactive tutorial on AJAX.

Ajax: the Definitive Guide. http://oreilly.com/catalog/9780596528386
An excellent textbook on AJAX.

Part III

Programming Google App Engine with Java

Chapter 9

Google App Engine and Java

We've been doing all our development of App Engine services and applications in Python. For lots of applications, Python is a terrific language. In fact, for many developers, there's no reason to ever look at anything but Python. For the stuff we've been doing so far, it's excellent: it's nice and light, and its lightness allowed us to build a cloud application one layer at a time, seeing all of the details of how the inner plumbing of a cloud application really works.

To be honest, I prefer something that can catch my silly mistakes at compile-time, instead of waiting to see a stack trace in my browser when things go wrong. For example, while I was writing the Python code for this book, I ran into a problem where I accidentally passed a string from a request handler as a parameter to a template, which was expecting a list of strings. The result was an ugly stack dump in my browser window. In my opinion, there's no good reason to deal with an error like that at runtime when it could have been caught ahead of time.

And that brings us to Java. App Engine supports two main programming languages: Python and Java. Python is the lightweight dynamic language; Java is the heavy artillery. And in App Engine, you build Java applications using a toolkit called GWT, which is (without any exaggeration) a work of genius.

Even if you're a Python fan, there are some good reasons to consider using Java for your cloud applications, such as the following:

Strong typing. Strong typing can catch many kinds of programming errors. Depending on your programming style, strong typing can make your life much easier by catching many of your errors when

Static versus Dynamic Languages

One of the great debates among programmers is about static versus dynamic languages. It's a debate that will go on forever because both sides have good points. The basic difference has to do with when errors are detected. In a dynamic language, errors aren't caught until the bad code is executed. For example, in a dynamic language like Python, if you write a method call like `x.foo()`, and x doesn't have a "foo" method, you won't get an error message until that statement actually executes.

In a static language, you need to declare types for things. Then, using the information provided by those declarations, errors like the undefined method in the example above can be caught at compile time.

It's a trade-off: in dynamic languages, you don't need to write type declarations to prove to the compiler that your program is correct. That's very convenient, and it can lead to a style of programming in which your code is much simpler—and simpler code is less likely to have hard-to-find errors.

On the other hand, static languages catch a lot of mistakes for you. They force you to be more rigorous about how you write your code in order to make sure that it passes the compiler, and that process causes you to produce better-designed code.

Personally, I fall on the static language side of things. I find that the extra work of dealing with the type system saves me a huge amount of effort in the long run. Most of the silly mistakes that I make get caught by the compiler and never cause problems at runtime. In fact, my own preference is for *very* strongly typed languages, like the functional language ML. ML's type system is incredibly expressive and incredibly strict, much more so than more familiar static languages like Java and C++. But in return, my ML programs almost never have runtime errors. Nearly all of the mistakes that I make end up getting reflected as inconsistencies in types. I've written thousand-line programs in ML and had them work without a single error on the first run after spending days working with the compiler to get rid of the statically detected type errors.

you compile your program. This is particularly valuable in an environment like the cloud, where it's harder to debug your program. You can't just fire up a debugger and probe it. You can't add print statements to find where things went wrong. Anything that helps you catch problems ahead of time can be a huge time-saver.

Style. As you'll see later in this chapter, developing a cloud application in Java has a very different style and structure from Python. For some developers, the style of Java development in App Engine can be much more comfortable than Python.

Tools. Google released a set of plugins for the free Eclipse IDE for building Java/GWT App Engine services and applications. Eclipse is an absolutely amazing tool, and the App Engine plugins make everything easier. (You can use Eclipse with Python, but there's no specific App Engine support, so it ends up being pretty painful.)

In this chapter, we'll take a look at developing cloud applications using GWT. We'll do that by taking our chat application and porting it to Java/GWT. We'll go through a compressed version of our journey so far, looking at how to do what we've already done, this time in Java.

9.1 Introducing GWT

There's one reason for using Java that completely outweighs all of the others—GWT. GWT is amazing. It lets you write your entire cloud application in Java. The server side is compiled in the usual way for Java—compiled into Java bytecodes that are executed on the JVM. On the server side, it's a nice framework, but it's not particularly special. But then there's the client.

GWT lets you write your client as a Java program. You write the client in Java almost like a traditional GUI application: you build a UI from a collection of widgets using layout managers, attach event handlers, and so on—absolutely typical GUI code. But GWT translates that GUI code into HTML and JavaScript: instead of compiling Java to Java bytecodes, it compiles Java to JavaScript source code, which then executes on the client. And for all of the AJAX stuff in which the client and server need to communicate, GWT can generate remote procedure calls. It's not a totally automatic process, but it's vastly easier and more robust than writing JavaScript AJAX code manually. (To be honest, my first reaction when I heard about this was, "They're out of their minds; that's ridiculous!" which goes to show you why I'm not rich and famous.)

Because of the way it's set up, building an application in GWT is different from what we did in Python with webapp. Our first example is going to have a beautiful UI, and we don't need to wait to get to how to set up templates and floats with CSS—we'll just dive right in and let GWT do what it does best.

Programming in GWT is, in many ways, much more like programming an application with a traditional desktop GUI framework. You define your UI almost the same way you would for a traditional desktop app, and GWT takes care of generating most of the HTML, CSS, and JavaScript that's necessary for making that app work. Most of Google's recent applications (including things like Wave) are implemented using GWT.

To start looking at GWT, download the App Engine SDK for Java. I'm not going to walk through it in detail, because it's basically the same process that you used to download the Python SDK in Chapter 2, *Getting Started*, on page 20. In addition to the basic framework, you can also install a set of plugins for Eclipse, which provide an excellent programming environment. I highly recommend downloading Eclipse and the App Engine plugins. The ability to use Eclipse for App Engine development is one of the best reasons for working with Java! Eclipse is free, and it's really easy to set up. The downside to GWT is that there's a lot of *metadata*—that is, a lot of extra files that tell GWT what to do with the Java source, things like which parts to compile to JavaScript for the client, which parts to set up as a servlet bundle for the server, and so on. Maintaining all of those files can be painful, but the Eclipse tooling is a huge help. You *can* program in GWT without using Eclipse, but you really shouldn't. From here on, I'm going to assume that you're using Eclipse with the GWT plugins.

GWT constitutes a very different approach to building a cloud application. In Python and webapp, everything was focused on the server. Of course, we built client UIs, but we did it by focusing on what the server needed to do to generate the UI on the client. The process centered on building request handlers and the CSS and templates that the request handlers needed. GWT is almost exactly opposite: in GWT, you focus on the client. You build a client UI using a framework that looks like a traditional client application. When your client needs something from the server, you make a remote procedure call (RPC) to invoke it; GWT takes care of most of the work of turning that RPC into an AJAX call.

With that in mind, let's start building a GWT application.

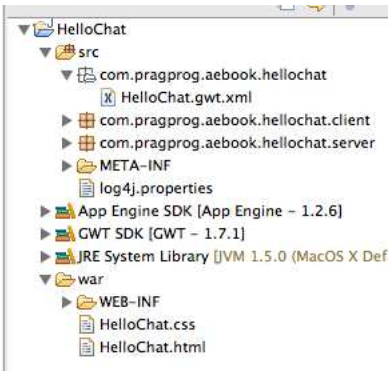


Figure 9.1: The GWT project directory structure in Eclipse

9.2 Getting Started with Java and GWT

To begin, we'll look at something like a basic "Hello, World" program. The GWT tools for Eclipse automatically build a project skeleton, which is a basic GWT "Hello, World"; so instead of writing our own, we'll just let Eclipse do it and walk through the pieces, seeing how it's all put together. In Eclipse, select New from the File menu. In the dialog that comes up, pick New Web Application Project. Then fill in the resulting dialog box with a project name and the name of the Java package you want to use for your Java code. I selected HelloChat as the project name and com.pragprog.aebook.hellochat for the Java package name.

The starter application sets up a page that prompts users for their names; when a user enters a name, it pops up a dialog box saying hello.

The Structure of a GWT Application

A GWT application consists of a set of *modules*. A module is a GWT package consisting of Java code, JavaScript, HTML files, images, data definitions, and whatever else you need in a web application. The directory structure that you get when you create a GWT/App Engine project in Eclipse is based on the structure of the GWT module that it implements.

To begin with, let's look at that directory structure. You can see the structure in the Eclipse package browser in Figure 9.1. Inside the App Engine project, there are a collection of GWT libraries plus two main

components: a source directory named src and a target directory named war. war stands for "web archive": the deployable application that you upload to App Engine is a war file.

The source directory itself is also divided into three parts: a module declaration, a package for the client-side Java code, and a package for the server-side Java code.

The server package, com.pragprog.aebook.hellochat.server, is deceptively simple, consisting of one almost trivial source file, because GWT is going to automatically generate the server-side plumbing.

The client side has a three files. One of them, HelloChat.java is the main body of our application. The other two, GreetingService.java and GreetingServiceImpl.java are part of the setup for a GWT remote procedure call. These files contain the declarations that GWT needs in order to allow us to do AJAX client/server applications without explicitly setting up XMLHttpRequests. We'll look at how those files work in Section 9.3, *RPC in GWT*, on page 135.

The way that these pieces fit together is determined by the GWT module declaration.

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/HelloChat.gwt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.7.1//EN"
    "http://google-web-toolkit.googlecode.com/.../gwt-module.dtd">
❶ <module rename-to='hellochat'>

  <inherits
❷       name='com.google.gwt.user.User'/>

  <inherits
❸       name='com.google.gwt.user.theme.standard.Standard'/>

  <entry-point
❹   class='com.pragprog.aebook.hellochat.client.HelloChat'/>
</module>
```

❶ The fundamental unit of code in GWT is a module, which consists of a collection of things: Java code; resources like CSS, HTML, or image files; and GWT customizations, like Java to JavaScript compiler extensions. This line declares the module that will contain our application. The rename element is part of GWT's URL handling: GWT will tell the server to set this module up at a URL path ending with hellochat.

❷ Modules in GWT can inherit things from other modules. It works pretty much like object-oriented inheritance. Our application is a submodule of com.google.gwt.user.User, which is the standard module for an application with a user interface. Most of the basic functionality of GWT—the UI widgets, the remote procedure call plumbing, and the basic server-side servlet infrastructure—are inherited through this declaration.

❸ Part of the reason GWT defines modules in addition to using class inheritance in the Java code is because there are a lot of resources in a GWT module besides code. A module can include things like CSS. The inherit statement pulls in the CSS files that define the look of the UI widgets in our application. We can change the look of our application by inheriting from a different style module.

❹ The Java code for a GWT application starts with an *entry point*. An entry point is, pretty much, the GWT GUI equivalent of a main function. In the module file, you declare entry points for code you want executed in your GWT application. In this case, the entry point is the class HelloChat.

Setting Up the UI in GWT

Within a GWT module, the user interface frame is defined by an HTML file. The HTML file isn't considered source code, so it doesn't get put into the src directory. It's a static resource—a file that contains information that will be used by the code. So the HTML file ends up in the war directory. Let's take a look at its contents:

Download workspace/HelloChat/war/HelloChat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
❶  <head>
     <meta http-equiv="content-type" content="text/html; charset=UTF-8">
     <link type="text/css" rel="stylesheet"  href="HelloChat.css"/>
     <title>Web Application Starter Project</title>
     <script type="text/javascript" language="javascript"
❷            src="hellochat/hellochat.nocache.js"></script>
   </head>

   <body>
     <h1>Hello World</h1>
❸    <table align="center">
       <tr>
❹        <td colspan="2" style="font-weight:bold;">Please enter your name:</td>
       </tr>
```

```
        <tr>
❺         <td id="nameFieldContainer"></td>
          <td id="sendButtonContainer"></td>
        </tr>
      </table>
    </body>
</html>
```

❶ The HTML frame file is a standard HTML file. It starts off with the usual HTML stuff: the doctype declaration and the head block with the usual meta tags.

❷ This is the most important line of the entire file! What makes the HTML file into a GWT application frame is this include line. It pulls in the JavaScript file that's going to be generated by GWT, containing all of our application code.

❸ As I'll explain in more detail later, you can do layout in the UI using either static structures defined in the HTML file or dynamic structures defined in Java code. For our application, that HTML frame defines a static structure for the main UI page. The easiest way to do that is using HTML tables. (We could also do it using CSS floats, as we saw in the Python code, but if we want to do dynamic layout, it would be much better to let GWT take care of it.) So we set up a two-column table: one column for the text entry box and one for the Send button.

❹ The HTML static structure can include static content as well as static structure. As usual, if we can separate things like static content from program logic, we should. So we use the static frame here to insert a title line and use the HTML table layout controls to make it span both columns of the layout.

❺ Now we get to something interesting. What we're doing here is creating an empty box in the UI. The <*td*> tag creates a box in the HTML layout, but it's empty—there's nothing inside of the tag. In our Java code, we'll insert something, referencing it using its id= tag. We create two boxes this way: one for the text box and one for the button.

Now we can get to some code. As we saw above in the module declaration, the application has one entry point. The full entry point method is pretty long; it incorporates both the creation of the UI elements, setting up event handlers, and setting up remote procedure calls for the client/server communication. Let's look at it in pieces. We'll start with

the part that builds the main UI—that is, the main page that prompts the users for their names:

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
❶ public void onModuleLoad() {
❷         final Button sendButton = new Button("Send");
          final TextBox nameField = new TextBox();
          nameField.setText("GWT User");

          // We can add style names to widgets
❸         sendButton.addStyleName("sendButton");

          // Add the nameField and sendButton to the RootPanel
          // Use RootPanel.get() to get the entire body element
❹         RootPanel.get("nameFieldContainer").add(nameField);
          RootPanel.get("sendButtonContainer").add(sendButton);

          // Focus the cursor on the name field when the app loads
❺         nameField.setFocus(true);
          nameField.selectAll();
```

❶ An entry point class is a container for the GWT equivalent of a main function. Conceptually, it really is like the main program in a non-GUI tool. But in Java, everything needs to be enclosed in a class, so we must create a skeleton class around the actual main. In a typical GWT application, this is the *only* method that's defined on the entry point class—it's just an overcomplicated wrapper for a single method. The real main function is the onModuleLoad method of the entry point. As the name suggests, this is what gets executed when the GWT module is loaded by the client. Inside this method, we create the UI widgets, lay them out, and set up the event handlers.

❷ The first thing we do inside of onModuleLoad is create the UI widgets. For basic cases, it looks pretty much like the way we'd do it if we were building a non-browser UI. We create a button and a text box where the users will enter their names.

❸ The first place that things start to look different from a traditional non-browser UI is in the management of the style attributes of the widgets. In a typical GUI toolkit, there are a set of methods to call for various style attributes. For example, in the Mac OS Cocoa widgets, we could modify the gradient of a button using a call like [button setGradientType: NSGradientConcaveWeak]. In GWT, that's all done using CSS: we'd set a CSS attribute to create a gradient image for the button background, and we'd add the line

background: url("images/gradient.png") to the CSS style block for .gwt-Button. The only call for managing style is one that sets up a connection to a CSS style. The style name is translated by GWT into a CSS class= attribute. It might seem a bit strange at first, but it's really nice in practice: it helps maintain that separation of concerns—you really shouldn't clutter your code with visual style stuff, and you should have all of the style stuff in one place. The way GWT uses CSS gives you a really convenient way of doing that.

❹ Now we get to layout. GWT provides you with a GUI context that's basically the contents of the browser page, called the RootPanel. To access the root panel directly, call RootPanel.get(). We can also do part of our layout using HTML, as in this example. If the application's main HTML page contains elements that are named with an id= attribute, we can access those elements using get(name). In this case, the root page for our application did provide elements for pieces of our application. This is pretty typical of GWT style: we've got a choice between doing things like layout statically (by doing it in HTML) and doing them dynamically (by writing layout code in Java). In general, when the layout is pretty much fixed (like it is in this case), it's easier to write an HTML table and just fill it in from Java. To create something on the fly, like the dialog box we'll see in a few minutes, use a GWT layout manager. In the static layouts, we can place a layout box on the page by calling get and then inserting a GUI widget into it using add(widget).

❺ Finally, when the UI loads, we'd like it to work so that if the user starts typing, it will show up in the text box. We do that by setting the *focus*: the focus is the widget on the screen that receives UI events like keystrokes. Users can set the focus by clicking the mouse inside of a widget, but it's annoying to be forced to do that when there's only one place where it makes sense for the focus to be. So we set it to focus on the text entry box. We also have it automatically select the placeholder text that we put into the box, so if users start typing, their text will replace the placeholder.

That's it for the basic building of the GUI We're left with two other important pieces. Our application is going to get a name from a user and send it to the server. The server puts that name into a hello message and sends it back to the client to display in a pop-up dialog box. What we still need to do is put together the client/server communication and the dialog box. We'll look at the client/server communication

in the next section. First, we'll look at the dialog, which is more GWT UI work, but instead of using a static layout from an HTML file, the dialog is fully dynamic:

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
// Create the popup dialog box
❶ final DialogBox dialogBox = new DialogBox();
dialogBox.setText("Remote Procedure Call");
dialogBox.setAnimationEnabled(true);
❷ final Button closeButton = new Button("Close");
// We can set the id of a widget by accessing its Element
closeButton.getElement().setId("closeButton");
final Label textToServerLabel = new Label();
final HTML serverResponseLabel = new HTML();
❸ VerticalPanel dialogVPanel = new VerticalPanel();
dialogVPanel.addStyleName("dialogVPanel");
dialogVPanel.add(new HTML("<b>Sending name to the server:</b>"));
dialogVPanel.add(textToServerLabel);
dialogVPanel.add(new HTML("<br><b>Server replies:</b>"));
dialogVPanel.add(serverResponseLabel);
dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
dialogVPanel.add(closeButton);
❹ dialogBox.setWidget(dialogVPanel);

❺ closeButton.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
                dialogBox.hide();
                sendButton.setEnabled(true);
                sendButton.setFocus(true);
        }
});
```

❶ First, we need to create the dialog box. This is a popup, so it's not contained in the browser frame. That means that we can't just grab the RootPanel; we need to create a freestanding widget. GWT provides a convenient widget for that: DialogBox is a free-standing window frame that can embed any GWT widget—we just create its contents and insert them. Since it's a window, it has a title bar and we can set its contents using its setText method.

❷ We want the users to be able to get rid of the dialog box whenever they want, so we create a Close button, which we'll add to the dialog box frame later. As usual, we can set the attributes of the widget with CSS. In this case, we do it by diving down directly to the HTML. Given any widget, we can get the XML element corresponding to that widget by calling getElement(). Then we set its ID to allow a CSS style to reference it using the setId() method of the XML element.

After the Close button, create another couple of widgets. There's a Label, which is a piece of noneditable text embedded in a widget. Then there's something interesting—an HTML widget, which is a wrapper for a chunk of literal HTML text. Whatever is inside of the HTML widget is rendered directly into the HTML page for the UI. That's useful for embedding things like styled text, where it's often easier to just use HTML markup around a piece of text than it would be to do the programmatic manipulation to produce the same effect.

❸ Now we're going to lay out a series of elements. Since we don't have a static HTML frame, we need to specify how to lay them out using GWT. The layout is just a bunch of stuff stacked vertically. GWT has a widget for doing that—the VerticalPanel. We just add the widgets of the UI to the panel in order. Notice the HTML markup here: there's some text we want to show in boldface. Instead of creating a Label widget and setting its style attributes to make it bold, we can just wrap the text in <*b*> tags.

❹ We've got the UI elements laid out in a VerticalPanel. All we need to do is tell the dialog box that the panel is what it should show: we do that by setting the dialog box's widget. Now the visual parts of the box are all done. The box starts off invisible: stand-alone widgets like this don't actually appear on the user's screen until we explicitly tell them to. As we'll see later, we can do that with a dialog box by telling it where it should appear. Most of the time, that's in the center of the browser window—so the dialog will be made visible by calling its center() method.

❺ With the basic UI set up, we can finally look at how to handle events in GWT! It's pretty much the same as in Java's Swing library. Create a handler object, and attach it to the appropriate widget using an addXXXHandler method. In this case, we're attaching the handler that closes the dialog box when the user clicks its Close button, so we attach a ClickHandler object. In its onClick method, we make the dialog box invisible and enable the entry area of the main page.

9.3 RPC in GWT

Now we get to the complicated part.

As I mentioned before, AJAX code is not written explicitly in GWT. Instead, we write something called a *remote procedure call* (RPC). An RPC is something that looks almost like a normal method call, but under the covers, it's translated by the system into a request sent from the client to the server. The return value of the RPC is the response sent from the server back to the client.

Just like any other RPC system, there's a client side and a server side in GWT. We can look at the code for them separately; it's up to the GWT RPC system to string them together.

If you've done any distributed programming, Google-style RPC is probably not what you're used to. Traditionally, RPC tries to appear as much like a traditional function call as possible. In other words, if we want to provide an RPC for a factorial function, the function implementation would look like a traditional function declaration, and an invocation of it would look like a traditional invocation. For example, Java has a native RPC layer, where we define a remote object by an interface, and then we can invoke methods on an object of the interface type.

We could define a factorial service as a Java interface:

```
public interface Fact extends Remote {
    int fact(int n);
}
```

Then in code that uses it, we'd acquire a handle for the remote interface and invoke it directly with something like this:

```
int j = f.fact(n);
```

Wrapped around that, we'd have some plumbing in the server to make the object available so that the client can get a handle for it, and we would likewise have code on the client to get hold of a handle on the remote object. But the invocation itself looks like a normal, local invocation.

There's a problem with this approach: communication is slow. A remote procedure call can easily take two orders of magnitude more time than a local call. In a traditional call, the code is stuck waiting until it gets the response back from the server. That's a huge waste of time, and it can create unacceptable delays in our user interface.

Google therefore uses something called *asynchronous* or *continuation passing* style for remote calls. The call itself returns nothing. Instead, it takes an extra parameter, which is a function to invoke on the result of the RPC whenever it's received.

For example, imagine we have a factorial service. In traditional style, what we want to do is this:

```
System.out.println("Foo = " + Math.log(3 * f.fact(n)));
```

In continuation passing style, that would be something like this:

```
f.fact(n, new AsynchCallback<int>() {
        public void onSuccess(int result) {
            System.out.println("Foo = " + Math.log(3 * result));
        }
    });
```

All I did was take the original code that was going to use the return value of the RPC and wrap it up in an object that I could pass to the remote call. All that does is start the process. My code can go off and do other things while the RPC is translating the call to a message, sending it, waiting for the server to send a respond, and then translating the response message back to a result value. Whenever the result comes back, the callback will be invoked, and the result of the call will be processed.

It can feel a bit unnatural when you're used to imperative programming. (Functional programmers do this all the time, even when they're not doing distributed programming.) It takes some time to get used to, and even then, it sometimes seems a bit twisted. But overall, the advantage of not having to wait for the RPC outweighs the problems. It makes applications more responsive to the users, and that's the most important thing.

Client-Side RPC in GWT

No matter how great a toolkit is, communication is never simple. We need to be able to deal with translating parameters into a format that can be passed as a message, and we need to be able to deal with delays or even failures due to the network. To cope with that, GWT uses a very Google-ish idiom. At Google, we have some very distinctive and stylized ways of handling certain basic problems. In particular, we have a way of doing remote procedure calls using an asynchronous response handler. It's pretty foreign to people who haven't done a lot of distributed programming in the Google style. In general, GWT is really terrific about making plumbing invisible. However, in this case they decided not to.

GWT was originally developed for Google to use internally, and Google engineers spend so much time working in this style that it becomes natural. It's a bit strange, but it's really the simplest way to solve many of the traditional problems with RPC. With asynchronous RPC, we don't need to write multithreaded code on the client, but we still get code that responds to updates as soon as they become available and that doesn't block while it's waiting for a response.

At any rate, that's more than enough background. Time to get down to the nitty gritty and look at what's involved in writing RPCs in GWT. For our hello-world program, there's one remote call. It sends a user's name to the server and gets back an HTML fragment containing a greeting addressed to that user. In GWT terms, that's going to be a *service* provided by code on the server. The first thing we do in GWT is write a synchronous interface for the service methods. In our hello-world application, it's called the greeting service. Somewhat surprisingly, the greeting service interface is written in the client package (Remember, GWT is always client-focused; the client is going to be the user of the interface, so it's located in the client package.) The basic synchronous greeting service interface is shown below:

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/GreetingService.java

```
package com.pragprog.aebook.hellochat.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

❶ @RemoteServiceRelativePath("greet")
❷ public interface GreetingService extends RemoteService {
❸         String greetServer(String name);
}
```

❶ GWT service interfaces can be annotated with information about how they'll fit into the URL structure of the application. The application has a root URL, and all of the application addresses will have that root URL as a prefix. This annotation specifies the path of this service relative to the root URL. So if the application is at http://gwt.appspot.com/foo, then this service would be at the URL http://gwt.appspot.com/foo/greet. Service interfaces should always specify their relative path this way.

❷ The service interface declaration must extend he GWT interface com.google.gwt.user.client.rpc.RemoteService. Using this superinterface specifies that our interface is intended for service in a GWT application and that GWT should translate it to JavaScript.

❸ This is the synchronous method declaration. Any parameters to a service method must extend either java.lang.Serializable or the GWT specific variant com.google.gwt.user.client.rpc.IsSerializable. One thing that's very important to understand here is that GWT using the Java synchronization interface to mark classes that will be passed in an RPC does *not* mean that it uses Java serialization. The use of either IsSerializable or Serializable is just a marker to tell GWT that it needs to generate code to serialize and deserialize the type. The actual format that GWT uses is not even close to compatible with Java's standard serialization.

The method declaration itself is completely standard: it's just a normal interface method declaration.

In addition to the synchronous interface, we also need to write an asynchronous interface. This is one of the places where I'm frankly mystified why the GWT team didn't provide some automatic support. To generate the asynchronous interface, we just write another interface, which is a pure boilerplate translation of the synchronous interface. It must have methods with exactly the same name as the methods in the synchronous interface; each method must have return type void; and each method adds a parameter at the end, which is an AsyncCallback and whose type parameter is the return type of the synchronous method. For example:

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/GreetingServiceAsync.java

```
package com.pragprog.aebook.hellochat.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

/**
 * The async counterpart of <code>GreetingService</code>.
 */
public interface GreetingServiceAsync {
        void greetServer(String input, AsyncCallback<String> callback);
}
```

The asynchronous interface doesn't have any explicit connection to the synchronous interface, and it doesn't have to use any special annotations or inherit from any special class. It's really only used by the client—the actual plumbing to map between the asynchronous and synchronous interfaces is generated by GWT. The only purpose of the asynchronous interface is to provide the call interface that the client will use.

Server-Side RPC in GWT

The server side of GWT RPC is amazingly simple. We just implement the synchronous client interface using a class that extends RemoteServiceServlet. For our greeting service, that implementation is as follows:

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/server/GreetingServiceImpl.java

```
package com.pragprog.aebook.hellochat.server;
import com.pragprog.aebook.hellochat.client.GreetingService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
 * The server-side implementation of the RPC service.
 */
@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet implements
                GreetingService {

        public String greetServer(String input) {
                String serverInfo = getServletContext().getServerInfo();
                String userAgent = getThreadLocalRequest().getHeader("User-Agent");
                return "Hello, " + input + "!<br><br>I am running " + serverInfo
                                + ".<br><br>It looks like you are using:<br>"
                                + userAgent;
        }
}
```

The @SuppressWarning annotation is a little unusual. It's there because serialization in later generations of the Java virtual machine uses a version identifier for each class file. If we have a class that implements java.lang.Serializable, Java generates a warning if we don't provide it with a version identifier field. Service implementations always implement Serializable, because RemoteServiceServlet inherits from the standard servlet class, which implements Serializable. Since GWT doesn't use the version identifier, including it in the code would just be pointless clutter; the annotation prevents the compiler from generating a confusing warning message.

And that's all we need to do on the server side. Of course, it gets more complex when we want to do persistence using datastore. But we'll save that for the next chapter.

So now we've got the full RPC for our hello application in place. It's a lot cleaner than the way we did AJAX back in Python: we have a well-defined interface, and we can invoke it just by making a method call. We don't need to worry about creating XmlHTTPRequests, parsing parameters, checking status codes, or any of the messy and error-prone things we needed to do in Python.

9.4 Testing and Deploying with GWT

Now that we've got a basic GWT application, let's run it. Just like in Python, there are two ways of running a GWT application in App Engine: a local mode, where the application runs on your machine, and a deployed mode, where the application runs in the App Engine cloud.

The local mode in GWT is very different from the local mode in Python. In Python, local mode was nice for testing without deploying, but it didn't really add much in the way of support for debugging. But with GWT, in the local mode, both the client and the server run in Java, and we can use all of the capabilities of Eclipse to debug the GWT application. That makes a huge difference: we have full access to breakpoints, traces, stepping, and all of the other Java debugging tools.

To run in local mode using Eclipse, go to the Run menu and pick Run As.../Web Application. GWT will open a local simulated browser environment to display the client and start a local Tomcat web server to execute the server.

To deploy it to App Engine, go to the package explorer view and right-click the project. In the project menu that comes up, there's a Google submenu. Just select Deploy to App Engine, and your program will be live in the App Engine cloud. If there's any information about the project that it needs, it will prompt you to fill it in the first time you run the deploy command.

We finished our first basic GWT application. It isn't as full-featured as our Python chat application yet, but it's got a better UI and a cleaner communication layer. In the next chapter, we'll take what we've learned about GWT and use it to build a Java version of our chat application. In the process, we'll learn about the Java interface to the datastore and about the restrictions that GWT puts on server-side Java code.

Chapter 10

Managing Server-Side Data

In the previous chapters, we examined the pieces of a basic GWT application in App Engine. We built a GUI, set up a simple RPC, and strung all of the plumbing together to get an application working. In this chapter, we're going to implement our chat program using GWT. We won't spend much time looking at how to build the GUI—there are entire books on building GUIs in GWT, and GWT's own documentation of its UI classes is excellent. What we'll spend most of our time on is *datastore*, the mechanism that lets us work with persistent data in App Engine. Just like in Python, we need to do some extra work to make classes persistent and queryable. Now we'll look at how to do that using Java.

We'll also touch on some other issues of the server-side plumbing in the Java side of App Engine. Specifically, App Engine puts some restrictions on what code can do and how it can run in the App Engine cloud environment, and we'll examine what those restrictions are and what effect they have on how we write the server side of our App Engine applications.

10.1 Data Persistence in Java

If you look back at Chapter 4, *Managing Data in the Cloud*, on page 53, you'll remember that we needed to do some extra work in our Python code to store data and to make things work correctly in the cloud. The same thing is true in Java. Unfortunately, this is one of the places where the static typing of Java makes things a bit more cumbersome. The basic back-end datastore used by App Engine is exactly the same as in Python, but making it work with Java takes a bit more thought.

It's not difficult, but there's a bit more boilerplate that we need to put into our code. As usual, Eclipse can take care of a lot of that for us. Let's get started without Eclipse's help, though. We'll write everything by hand, so that we understand all of the details.

In typical Google style, what App Engine does to make datastore work with Java is grab a standard Java API—Java Data Objects (JDO)—and pick out a useful subset of its functionality. JDO is a hugely complex, bloated API (typical of standards). But there's a kernel of goodness to it. App Engine uses that kernel.

Below, we'll see how JDO persistence works for describing persistent objects and for storing, querying, and retrieving them.

Storing Java Classes

In Python's datastore interface, we created a persistent class by adding attributes to the class object. In Java, we'll do something similar—but in Java, add attributes are added through the use of annotations in the class declaration. It's easiest to describe by example, so we'll take our chat message and translate it into a data object:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/ChatMessage.java

```
package com.pragprog.aebook.persistchat;

import java.util.Date;

import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

import com.google.appengine.api.datastore.Key;

❶ @PersistenceCapable(identityType = IdentityType.APPLICATION)
public class ChatMessage {

    public ChatMessage() {
    }

    public ChatMessage(String sender, String msg, String chatname) {
        this.senderName = sender;
        this.message = msg;
        this.chat = chatname;
    }
```

```
❶       @PrimaryKey
        @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
            private Key key;

❷       @Persistent
            protected String senderName;

        @Persistent
            protected String message;

        @Persistent
            protected String chat;

        @Persistent
            protected long date;

        public Key getKey() {
            return key;
        }

        public String getSenderName() {
            return senderName;
        }

        public void setSenderName(String senderName) {
            this.senderName = senderName;
        }

        public String getMessage() {
            return message;
        }

        public void setMessage(String message) {
            this.message = message;
        }

        public String getChat() {
            return chat;
        }

        public void setChat(String chat) {
            this.chat = chat;
        }

        public long getDate() {
            return date;
        }

        public void setDate(long date) {
            this.date = date;
        }
    }
```

Let's take a closer look at this example:

❶ For a Java object to be stored in the App Engine datastore, its class must be declared as persistent. In JDO, you do this by attaching a @PersistenceCapable annotation. In full JDO, we need to declare an identity type. App Engine only supports IdentityType.APPLICATION, but because Java's type system requires annotations to match their declaration, the field must always be declared.

❶ To be able to store, retrieve, or search for a particular object, the object must have a unique key that is used to identify it. In Python, when we defined a persistent object, the framework automatically added an invisible key field that was used by datastore. In Java, everything needs to be declared statically. So we need to manually insert the key declaration in the source code and annotate it as being a key. We mark it as a key using the @PrimaryKey annotation, and we tell it that the way to compare primary keys is through object identity using the valueStrategy = IdGeneratorStrategy.IDENTITY attribute of the @Persistent annotation.

Most of the time, we'll use a key object like this, which is initialized automatically and which we won't usually use directly. As we'll see later, we can do some customization of keys, but there's usually no need.

❷ Each field of a persistent object that should be stored needs to be annotated with @Persistent.

There are some restrictions on the data objects in Java. They're mostly requirements to keep the tangle of object pointers manageable:

- When a persistent object contains another persistent object as a field, it owns that object, and no other persistent object is allowed to have a reference to the owned object. This means that we sometimes need to save objects.
- When a persistent object contains a collection of other persistent objects, it owns all of the objects in the collection.
- We're limited to the basic Java collection classes. We can't use arrays, and we can't use any of the extended collection types. We can use concrete classes like ArrayList, LinkedList, HashSet, TreeSet, Stack, and Vector. We can also use the more abstract interfaces, like List, but when we save and then reload an object, we can't guarantee that the type of list will be the same in the restored

object (meaning that we might save something that uses a LinkedList and get back something that uses an ArrayList). Since that can have pretty dramatic performance implications, I recommend explicitly using the concrete collection types for JDO fields.

- We can use Java serializable types in data objects by marking them with the annotation @Persistent(serialized=true). Be warned that their behavior is a bit different than you'd normally expect from Java. For example, suppose we had two copies of a serializable object in a list field of a persistent object. If we saved that object and then loaded it, the two copies would not be guaranteed to be ==, and whether they would be equals() is dependent on how the equals() method is implemented for our class.
- Fields of type String are not allowed to be any longer than 500 bytes. We can store longer strings using the Text class from the datastore package for the field, but we won't be able to perform queries based on the value of the field.
- If we're *not* using Eclipse, we need to add an extra compilation step called *code augmentation*. During augmentation, the App Engine JDO implementation adds code to classes based on the persistence annotations that allow them to be stored and queried by the datastore. We need to make sure that the augmentation process gets executed each time we recompile our Java sources. (Eclipse automatically includes JDO augmentation into the project build, so it's taken care of—yet another reason to use Eclipse!)

10.2 Storing Persistent Objects in GWT

In Python, storing things in the datastore was incredibly simple. We created a persistent object, and then we called its put method. Presto!—it was stored. In Java, we need to do a bit more work. Again, it's a bit of static boilerplate. Java provides us with a lot of advantages, but it does require a lot more boilerplate.

To be able to store and retrieve objects, we need something called a PersistenceManagerFactory. The factory is very expensive to create, and we don't want to reinitialize it every time we process a request; instead, we set it up so it is created when our application is loaded into a server in the App Engine cloud. And we want it created in a nice, centralized place to be sure that anyone who needs a PersistenceManager knows where to find the one instance of the factory. There's a natural solution

to that—the singleton design pattern. We'll create a singleton class that statically creates a single instance of a PersistenceManager, which can then be accessed by anyone who needs it.

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/Persister.java

```
package com.pragprog.aebook.persistchat.server;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

public final class Persister {

  private static final PersistenceManagerFactory pmfInstance =
          JDOHelper.getPersistenceManagerFactory("transactions-optional");

  private Persister() {}

  public static PersistenceManagerFactory get() {
    return pmfInstance;
  }

  public static PersistenceManager getPersistenceManager() {
    return get().getPersistenceManager();
  }
}
```

Now any client code that needs to use a persistence manager can just invoke Persister.getPersistenceManager(). With a persistence manager, we can store an object o by calling PersistenceManager.makePersistent(o), followed by PersistenceManager.close().

This is a lot more trouble than the Python o.put() call: we need to set up a persistence manager factory, allocate a PersistenceFactory, and call close() when we're finished. Fortunately, there are benefits. The interface through the PersistenceFactory provides support for transactions. From the time that we allocate a persistence manager until we call its .close() method, everything we do is part of an atomic unit—that is, either it all succeeds or it all fails. Every object that we store, every change that's made to a persistent object, will either all be stored or none of it will. That's the beauty of transactions: we get the safety and security of a relational database. The boilerplate code can seem annoying, but there are advantages.

We're ready to post a new message. We need to create an RPC service, which is how the client is going to tell the server when it gets a new message to post. (Again, it's worth pointing out that in GWT we imple-

Transactions

You will constantly hear about *transactionality* when you talk to anyone about distributed applications. Here's why.

Transactionality prevents corruption of data. Without transactionality, if you're interrupted while storing something—because of a network glitch or a crash of some computer involved—you could end up with your data storage in an inconsistent state.

For example, imagine that you're writing an online store. You create an order record, which instructs your shipping department to ship an order to a customer; then you create a billing record that tells your bank how to collect the payment for the order. If your system crashes between the time you store the shipping record and the time you store the billing record, you could ship a product without ever collecting a payment!

You want the two steps to be *atomic*, which means that either both are successfully stored or neither are. The atomic unit in which everything gets stored successfully or nothing gets stored is called a *transaction*.

Java datastore provides a way to collect multiple storage operations into a single transaction.

ment the post operation as an asynchronous RPC—which is exactly what it is, in terms of the operation of our program—instead of getting tangled in the mess of the XMLHttpRequest.) Our RPC service needs two methods: one for posting a new message and one for fetching messages. The basic interface is shown below:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/ChatSubmissionService.java

```
package com.pragprog.aebook.persistchat.client;
import java.util.Date;
import java.util.List;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
import com.pragprog.aebook.persistchat.ChatMessage;

@RemoteServiceRelativePath("chat")
public interface ChatSubmissionService extends RemoteService {
  List<ChatMessage> postMessage(ChatMessage messages);
  List<ChatMessage> getMessages(String room);
  List<ChatMessage> getMessagesSince(String chat, Date timestamp);
}
```

We turn it into an asynchronous interface as usual:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/ChatSubmissionServiceAsync.java

```
package com.pragprog.aebook.persistchat.client;

import java.util.Date;
import java.util.List;

import com.pragprog.aebook.persistchat.ChatMessage;
import com.google.gwt.user.client.rpc.AsyncCallback;

public interface ChatSubmissionServiceAsync {
    void  postMessage(ChatMessage messages,
                      AsyncCallback<List<ChatMessage>> callback);

    void getMessages(String chatroom,
                     AsyncCallback<List<ChatMessage>> callback);

    void getMessagesSince(String chat, Date timestamp,
                          AsyncCallback<List<ChatMessage>> callback);


}
```

We implement that with a class in the server package. The implementation is very straightforward. We use the Persister that we just implemented to get a PersistenceManager for the operation, to make the message object persistent so that it will be saved as part of a transaction, and then to close the persistence manager, which will execute the transaction. Finally, we use the other service method, invoked directly from the server, to provide the user with an updated list of the messages in the chat.

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/ChatSubmissionServiceImpl.java

```
public List<ChatMessage> postMessage(ChatMessage message) {
        PersistenceManager persister = Persister.getPersistenceManager();
        persister.makePersistent(message);
        persister.close();
        return getMessages(message.getChat());
}
```

If we want to save any other objects as part of the same transaction as the chat message, we just add more calls to makePersistent to store them before the call to close.

10.3 Retrieving Persistent Objects in GWT

In Python, we needed to use an SQL-like query language called GQL in order to retrieve objects from the datastore. In Java, we do the same thing, but because we're working as part of a standard Java persistence framework, we use its query language rather than the custom language built for Python. The standard query language for Java Data Objects is called JDOQL. Like GQL, it looks a lot like SQL.

To be honest, we don't really *need* to use JDOQL. If we know the key for an object that we want to retrieve, we can fetch it using the PersistenceManager method getObjectById. For example, if x were the ID of one of our chat-message objects, we could retrieve it this way:

```
PersistenceManager pm = Persister.getPersistenceManager();
try {
   pm.getObjectByID(ChatMessage.class, x);
} finally {
   pm.close();
}
```

The catch is, obviously, that we need to know the key. There are definitely cases where we're retrieving an object for which we either know the key or we can figure out what it is. (We'll look at how to do things like that in Chapter 13, *Advanced Datastore: Property Types*, on page 191.) There's a trade-off here: fetching by key is a lot faster than fetching by query. But for our chat application (and for many similar applications), the speed of executing a query is so small compared to the cost of network communication for the enclosing request that it's not significant. So for our purposes right now, retrieving by key isn't particularly useful. For applications that do a lot of datastore interaction during the processing of a single query, the trade-off would be very different, and fetching by key could have a serious performance impact. Trade-offs like this abound in cloud programming: you need to be aware of just which of many factors is really important for performance.

There *is* one part of the previous code that's important: the use of try...finally. As I mentioned, persistence objects aren't lightweight. There are a lot of resources associated with one, and the longer it hangs around, the more cruft it's liable to accumulate. You must be sure that it gets closed so the resources can be reclaimed. Without the use of try...finally, if any of your code between the pm.getObjectById(...) and the close() encountered an error or threw an exception, the close() call could get skipped, which would be *very* bad. This is more of an issue

with retrieves than with stores because when you start a transaction to store something, you generally know all of the things you're going to store, so you rarely do any computation that could generate an error. And if you did, you don't want the transaction to be committed by a close()! But with retrieves, you often do things iteratively: you retrieve one object, which gives you the information you need to identify other objects that you want to retrieve. So the safety of the try...finally block is important for retrieval.

Most of the time—and specifically for our chat application—we'll use JDOQL queries to describe what we want to retrieve from the datastore. To retrieve all of the chat messages for a particular chat room, the JDOQL query would be this:

```
select from ChatMessage
where chat=desiredRoom
parameters String desiredRoom
order by date
```

Our JDOQL query (in fact, most JDOQL queries) has four parts:

select from ChatMessage
: The select clause specifies the set of objects to search in the query. It looks like the select clause from a SQL query and basically does the same thing. A SQL query selects a set of table rows that match some filter, and the select clause says what table to select from. A JDOQL query selects a set of objects that match some filter, and the select clause says what class to select from. We want to retrieve a set of ChatMessages, so we select from ChatMessage.

where chat=desiredRoom
: The where clause is much like the one from SQL: it provides a predicate (that is, an expression that will only be true for the objects we want to retrieve). We want to retrieve the messages from a particular room. The actual value of the room whose messages we want to retrieve is a parameter named desiredRoom.

parameters String desiredRoom
: The parameters clause doesn't have any equivalent in SQL, but it should. In most SQL libraries, we have to specify parameters using some really painful and awkward syntax. In JDOQL, the parameters clause declares a list of typed variables, and wherever those variables are used in the query string, they'll be replaced by the parameter values from the query invocation. So we say that desiredRoom is a parameter of type String.

order by date

The order by clause is again the same as in SQL: it specifies what order the objects selected by the query should be returned in. We want to see the chat messages in the order in which they were posted, so we do it by date.

JDOQL also has an alternative syntax: instead of using a string for the query, we can compose it programmatically. We'll use the programmatic version in the code below.

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/ChatSubmissionServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatMessage> getMessages(String chat) {
        PersistenceManager persister = Persister.getPersistenceManager();
        try {
                Query query = persister.newQuery(ChatMessage.class);
                query.setFilter("chat == desiredRoom");
                query.declareParameters("String desiredRoom");
                query.setOrdering("date");
                return (List<ChatMessage>)query.execute(chat);
        } finally {
                persister.close();
        }
}
```

This is fairly straightforward. @SuppressWarnings is an artifact of the way that Java handles typed lists; because Java uses something called type erasure to simplify compilation of typed collections, it can't verify that the cast to a typed list is valid. Since the compiler can't guarantee that you're not making a mistake, it generates a warning to let you know that there *might* be an error. The SuppressWarnings annotation basically tells the compiler, "Shut up, I know what I'm doing!" Aside from that little change, this is identical to the query we looked at above, translated into the programmatic form. It's better this way because it separates the different elements of the query and makes the code easier to understand.

10.4 Gluing the Client and the Server Together

Now we just need to tell App Engine how to glue the client and server code together so the chat app client can call the RPC methods that allow it to store and retrieve messages from the datastore. That's done using the App Engine web.xml, which is located in the war/WEB-INF directory of our App Engine project. This file declares the servlets that are part of our application, tells App Engine where to set them up on the server side, and tells GWT how to find them.

Download workspace/PersistChat/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Servlets -->
❶ <servlet>
  <servlet-name>chatServlet</servlet-name>
    <servlet-class>com.pragprog.aebook.persistchat.server.ChatServiceImpl
        </servlet-class>
  </servlet>

❷ <servlet-mapping>
    <servlet-name>chatServlet</servlet-name>
    <url-pattern>/chat/chat</url-pattern>
  </servlet-mapping>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>Chat.html</welcome-file>
  </welcome-file-list>

</web-app>
```

❶ Tell App Engine what servlets need to be deployed on the server for the application to run. Each servlet is given a name, which is associated with the class that implements that servlet.

❷ For each servlet specified in the servlets clause, give App Engine the URL where the application should run the servlet.

In this chapter, we built up the infrastructure of our chat application: we've got the client interfaces for making RPCs to post new chat messages and to retrieve the messages. We built the servlet implementation of the RPC methods using the App Engine JDO interface to the datastore. By doing that, we learned the basics of how to both store and retrieve objects from the datastore.

In the next chapter, we're going to return to UIs and look at how to build a really great-looking GUI for our chat application using the RPC services we just built. As we do, we'll explore the available GUI widgets. We will cover how to create GUI layouts in GWT, update the data being displayed in the UI without doing a full page-refresh, and respond to user actions.

Finally, we'll look at some more sophisticated ways of using the datastore. You can do a lot of really interesting things with datastore, but it's also got some rather peculiar limitations. We'll look in-depth at both the power and the limitations of the datastore.

10.5 References and Resources

The Java Datastore API. . .
. . . http://code.google.com/appengine/docs/java/datastore/
The official App Engine java datastore documentation.

Java Data Objects . . . . . http://java.sun.com/jdo/
The JDO standard documentation. JDO is the basic technology that the Java interface to the App Engine datastore is based on.

The Java Persistence API—A Simpler Programming Model for Entity Persistence. . .
. . . http://java.sun.com/developer/technicalArticles/J2EE/jpa/
An article with an overview of the Java persistence API used by App Engine.

Chapter 11

Building User Interfaces in Java

Now we've seen some of the basics of how to do things in Google App Engine using Java. We learned the basics of how to use the datastore for persistence, and we examined the general structure of how to build a complete Java cloud application using GWT. In this chapter, we'll look in more detail at how to work with GWT to build user interfaces for cloud applications.

11.1 Why Use GWT?

Before we dive in, it's worth mentioning that you don't *need* to use GWT to build App Engine services using Java. You can use any Java web application development framework you want—subject to a few limits imposed by the App Engine runtime. (Java programs running in App Engine can't use threading, locking, or runnable objects, which can be limiting.)

But you *can* use most of the common Java-based web frameworks. You could use a standard servlet environment, or Struts, or even Grails. GWT isn't an essential part of App Engine, so why am I focusing it?

The answer is partially just because I like it. I've played around with a lot of different toolkits for building web and cloud-based applications. My experience is that GWT is by far the best. The worst pain-points, the places that causes the greatest amount of trouble, are all generated automatically by GWT using a well-tested standard code generator.

Let me expand on that a bit. What GWT does for you is make writing the user interface for a cloud application as natural as writing a user interface for a traditional application. It gives you tools for testing and

debugging that make it easy to debug even where your code crosses the boundary between different languages—like your Java server and your client UI, which is really running in JavaScript or HTML5.

And that's the most valuable thing about GWT. You see, the biggest problem with cloud programming in general is that it's painful and difficult to test and debug. In a cloud application, you need to deal with the client program (HTML + JavaScript), plus whatever programming language the server is implemented in, plus XML and HTTP. That's a lot of complexity. But that's not the worst of it. You don't just need to deal with those languages: you also need to deal with the boundaries between them. You don't just need to write JavaScript code on the client: you need to write JavaScript code that knows how to generate HTTP requests and how to parse the XML response and generate JavaScript objects from them. Your server doesn't just need to have code to perform its basic operations: it needs to know how to parse HTTP requests to figure out what operation a client wants it to perform, and it needs to take the result of that operation and translate it into XML. And both the client and the server need to agree perfectly on what the requests, responses, and XML encodings are. Any problem with any piece of the code—particularly with code in the boundary regions between languages—-can cause subtle, hard-to-find problems. I have no idea how many hours of my life I've wasted tracking down errors involving trivial mistakes in XML formats or HTTP message headers.

The most important thing about GWT isn't that it lets you write your UIs using widgets like a native UI toolkit. That's really nice and valuable. But far more important and valuable is the fact that GWT takes care of the boundaries: you write all of your code in Java, and GWT takes care of translating that Java into XML, HTTP, JSON, JavaScript, HTML, and CSS, and generates the code that bridges the boundaries between them. All of that stuff is still going on under the covers—but with GWT, you don't need to deal with it. You won't have problems with those boundaries, and the amount of time and effort that you'll save is astonishing.

11.2 Building GWT UIs with Widgets

It's time to build our own UI using GWT. We'll take our chat application and build the same basic UI that we built in Python, this time using GWT. To start, we'll concentrate on the real UI aspects—except where

necessary, we'll ignore the application logic and focus on presentation. This is typical of how you'll build a GWT-based App Engine service: you start by figuring out what you want the service to do. As we did in the previous chapter, figure out the basic data you want to manipulate and what kinds of RPC calls you'll need. Then sit down with GWT and put together the UI.

Like most UI toolkits, in GWT, you work with *widgets*. A widget is a basic element of the user interface. There are widgets for all of the basic common user interface elements you're used to: text boxes, buttons, radio buttons, drop downs, menus, and so on. There are also *container widgets* for managing layout. This whole structure is, as I've said before, one of the great reasons for using GWT: you don't need to worry about how to do the CSS to make your UI layout work. You don't need to figure out how to set up all of the JavaScript to draw your UI. You build your UI out of widgets—including container widgets—and then you let GWT do the work. You can focus on the what, rather than the how.

In a traditional GUI toolkit, you start off with a window and place the widgets in it. With GWT, you're putting your application into a web page and putting widgets into that. So in GWT, the starting point is an HTML page. You can put whatever HTML you want into the basic page; in fact, if you want to, you can do almost all of the GUI layout using HTML. Back in Chapter 9, *Google App Engine and Java*, on page 123, we did use HTML to do the basic layout. But you don't have to do that: you can use an HTML page that is totally blank and just use GWT to do all of the layout. That's what we'll do in this chapter. We still need an HTML page—the only way of loading a cloud application is through a web page. So we will use a minimal skeleton page, shown below. All this it does is set the links to load the GWT UI, using a <*link*> tag to load the CSS, and a <*script*> tag to load the UI code.

So we start with an HTML page. To tell App Engine exactly which page is the main frame of our application, we edit the web.xml file the same way we did in the previous chapter. We'd like the main page of our application to be called Chat.html, so we just edit the welcome-file-list entry:

```
<welcome-file-list>
    <welcome-file>Chat.html</welcome-file>
</welcome-file-list>
```

Once the system knows what page provides our application frame, we can set that frame up by creating the Chat.html page:

Download workspace/PersistChat/war/Chat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">

    <link type="text/css" rel="stylesheet" href="Chat.css">

    <title>AppEngine Chat</title>

    <script type="text/javascript" language="javascript"
            src="persistchat/persistchat.nocache.js">
    </script>
  </head>

  <body>
  </body>
</html>
```

There's not much to it; it's a very minimal file. The body is completely empty: we're going to populate it completely using dynamic layout code in GWT. The important things are these:

- A link to the .css page for the application. You'll usually use the same name for the CSS and HTML files, so the CSS is Chat.css.
- A script tag, which loads the GWT-generated javascript code.

With the page in place as a frame for our application, we can start on the real work. What do we want in our Chat UI? Let's look at the mockup we put together previously in Figure 7.2, on page 104. We've got a title bar up at the top; under the title bar, there's a subheader with the current time and date; then there's a box containing a list of chats and a box containing the chat messages side by side. Underneath these boxes is the entry box where we can type new messages and a button to send them.

In GWT, we can set up exactly what we just described. The basic structure of the UI is a vertical layout: a top box with the title and subtitle, a middle box with the chat list and chat messages, and a bottom box with the text entry. We want to write GWT code that will set up those widgets and arrange them into our UI.

In GWT, the client application always starts by calling the onModuleLoad method of the application's main class. onModuleLoad is basically the main function of a GWT application, and since the first thing that a GWT application needs to do is set up its user interface, put the UI construction code right there. In the rest of this section, we'll be building up the implementation of onModuleLoad.

We'll start by creating the basic frames of our UI:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
❶ final VerticalPanel mainVert = new VerticalPanel();

❷ final VerticalPanel topPanel = new VerticalPanel();
  final HorizontalPanel midPanel = new HorizontalPanel();
  final HorizontalPanel bottomPanel = new HorizontalPanel();
❸ mainVert.add(topPanel);
  mainVert.add(midPanel);
  mainVert.add(bottomPanel);
```

❶ In order to set up a UI with a series of stacked UI elements, we must create a Panel widget and insert the other elements into it. First, we create a VerticalPanel, which is the main element of the UI.

❷ Now, we want to create the vertical elements of the UI. We've got three things. At the top, we've got something that is going to vertically stack a title bar and a subtitle, so that's another VerticalPanel. In the middle, we'll have two things side by side (the chat list and the chat transcript), so that's going to be a HorizontalPanel. Finally, at the bottom, we're going to have the entry box and Send button—again, stacked vertically—so that's another VerticalPanel.

Now we want to put the title and subtitle into the top panel:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
❶ final Label title = new Label("AppEngine Chat");
  final Label subtitle = new Label(new Date().toString());
❷ title.addStyleName("title");
❸ topPanel.add(title);
  topPanel.add(subtitle);
```

❶ In the top panel, we're going to want to put two widgets, both of which only display some text. The easiest way to put text into a GWT UI is to use the Label widget. We create label widgets for both the title and subtitles. Since they're passive and don't change, we can just put the text into them when we create them.

❷ The content of the subtitle widget is just plain text. But for the title, we want to make it look different: the title should be bigger and bolder than the text beneath it.

Most UI toolkits provide some way of managing styles to set things like text size, font, color, and so on. But we've already seen a way of doing that in a web environment: CSS is a really nice, flexible way of managing style attributes. Why reinvent the wheel? Instead of defining a new API for doing styles, GWT uses CSS. You don't need to write much of it: GWT has default styles for its widgets that look good. But when you want to customize something, you specify the attribute using CSS. Write the CSS for the style you want in your applications CSS file, and then tell GWT to use a particular style for a widget by calling addStyleName("style"). We'll put the style apptitle on the title label. (There is a setStyle name, but it clears all of the other style attributes associated with the widget. GWT automatically sets a lot of attributes to make the widgets look right, and you don't want to clear those. addStyleName just adds your style to the CSS cascade, so it still inherits all of the style attributes other than the ones you specifically set.) We'll get to where to put the CSS in a moment.

❸ Once the widgets are created, we need to add them to their panel.

In the code above, we called addStyle to alter the appearance of the label widget containing our application title. What we did was insert a reference to a CSS style. This is how we alter anything about the style of our application's widgets using GWT. GWT does a really good job of setting attractive defaults, but in any real application, there will be places where you'll want to customize. CSS is a great way of doing that: CSS provides a complete set of style attributes in a standard way, and with the cascade structure of CSS, it's very easy to set up a general style for your application and customize it where necessary.

We're going to change a couple of styles, but the basic pattern is always the same. In the Java code, we add a CSS class attribute to the element whose style we want to change; then in the CSS file, we write a class entry for that. Here's what we add to the default CSS:

Download workspace/PersistChat/war/Chat.css

```
.title {
    font-size: 4em;
    font-weight: bold;
    color: #4444FF;
}
```

```
.messages {
    background: #AAAAFF;
}

.emphasized {
    font-weight: bold;
    background: #FFFF88;
}
```

We're doing three style classes. I'll describe the first one in a bit of detail, and the other two should make sense. The first CSS class is the one for the title label we just created. We want to make the title text big and bold, with a colored background. So in our CSS class, we change its font-size attribute to make it big; we change its font-weight attribute to make it bold; and we modify its background element to change the background color.

That's it; we've changed the style.

We also need to set up the contents of the other two subpanels. The basic mechanics are similar to what we've just seen, but we use some more interesting widgets. The chat list panel is particularly interesting because its contents are a set of links that are generated dynamically, based on which chats are available. That's a bit complicated, so we'll come back to that: first, let's finish looking at the basic layouts of the other subpanels. Here's the basic layout code for the middle panel:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
❶ final VerticalPanel chatList = new VerticalPanel();
chatList.setBorderWidth(2);
final Label chatLabel = new Label("Chats");
❷ chatLabel.addStyleName("emphasized");
chatList.add(chatLabel);
❸ chatList.setWidth("10em");
❹ populateChats(chatList);
// "TextArea text" is defined as a field of the class, so that
// the textarea can be referenced by handler methods.
❺ text = new TextArea();
text.addStyleName("messages");
text.setWidth("60em");
text.setHeight("20em");
midPanel.add(chatList);
midPanel.add(text);
```

❶ We're going to put the list of chats in a column on the left, so we create a vertical panel. Then we give it a visible border by setting its border width to 2 and put a label on top of it.

❷ As we did before, we're going to change the style of the label on the chat list. This time, we'll use the style emphasized, the definition of which you can see in the CSS above.

❸ Normally, GWT picks a default size for panels based on the size of the largest thing inside of the panel. We want a bit more control than that, so we explicitly set its width to 10 ems. The 10 ems figure seems random, but it's not. The text area displaying the chat transcript is going to be 60 ems, because that's a width that will fit most messages cleanly. After experimenting with different sizes, making the chat list one-sixth the width of the transcript has the nicest appearance. Making it any narrower gives it an awkward, skinny look; making it wider puts too much whitespace on the left side of the window.

❹ To populate the contents of the chat list, we need to set up some RPCs and callbacks; putting that stuff inline here where we're constructing the basic UI layout would be awkward at best, and it would violate our principle of separation of concerns. Layout of the UI is one concern; RPCs and event handling is a different one. So we just call the code that does that work. We'll look at it a bit later.

❺ And finally, we create the text area. We set its width to 60 ems and its height to 20 ems. Then we finish adding all of the widgets we've just created to the panel, and the layout of our middle section is all done!

Finally, we'll lay out the bottom portion:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
final Label label = new Label("Enter Message:");
label.addStyleName("bold");
final TextBox messageBox = new TextBox();
messageBox.setWidth("60em");
final Button sendButton = new Button("send");
bottomPanel.add(label);
bottomPanel.add(messageBox);
bottomPanel.add(sendButton);
setupSendMessageHandlers(sendButton, messageBox);
```

This is pretty much the same sort of stuff. We create the widgets and lay them out. Again, we need to set up some event handlers and callbacks; again, we keep our concerns separate and just use a call to the setup.

Now that we've got the UI laid out, the only layout thing remaining is focus management. The focus of a UI is the widget that is selected as the default target of any actions. If you start typing, the focus is whatever widget receives the characters that you type. Setting the focus is a small detail in some ways, but for a user, making sure that the focus is in the right place has a big effect in making your application work smoothly.

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
RootPanel.get().add(mainVert);

// focus the cursor on the message box.
messageBox.setFocus(true);
messageBox.selectAll();
setupTimedUpdate();
}
```

When users run a chat application, they're going to expect to be able to start typing chat messages, so we want to make sure that the widget that lets them type messages is active.

11.3 Making the UI Active: Handling Events

We've got a UI now, and if we were to try to display it, it would look pretty good. But there's no activity. It doesn't know how to respond to any actions that the user takes, and it doesn't know how to fetch data from the server. The key to making the UI active in GWT is callbacks.

What we've got so far produces a UI that looks like what we want, but it's completely passive. It doesn't *do* anything. To get it to do something, we need to set up event handlers. Event handler code is intrinsically asynchronous. It's built entirely from *callbacks*. Instead of something like a windows event loop, where you write a loop that watches for user interface actions and then makes a decision, GWT event handler code creates objects containing code to handle a particular event.

Even before we worry about handling user actions, we need to handle some activities. We left one bit of laying out of the UI—the list of active chats. The problem is that we don't know what chats exist on the server, so we can't just generate the list when we're populating the UI. We need to retrieve that list, and we need to do it without interfering with the display of the application.

This process is typical of a lot of things in App Engine using GWT. We need to send a request to the server to get the list of chats. But we

Continuation Passing Programming

This kind of callback-based programming is sometimes called *continuation passing style* (CPS) code, a term that came from the functional programming community. It's based on the idea that any program can be written in a completely asynchronous style. Wherever you have code that calls a function f and then uses its result, you can replace it by adding a new parameter to f containing a function and then invoking that function with f's result.

For example, you could think of a function to multiply two numbers:

```
def mult(m,n):
  return m*n
```

In continuation passing style, you'd write this:

```
def cpsmult(m, n, done):
    done(m * n)
```

If you think about GWT, you'll see CPS all over the place. The UI event handlers are all basically CPS code; and the asynchronous form of GWT RPC is precisely the CPS form of the procedure call.

don't want our program to just stall and show nothing in the user's window while it waits for a chat response. In a typical Java program, we'd probably get around that using threads—we'd create a Runnable object and spawn it off to take care of retrieving and populating the chat list. But App Engine provides us with a controlled, limited environment. It does not permit us to create threads, so we can't do that!

What we can do is use GWT's asynchronous calls, which involves several steps. We need to add a method to the service containing our RPC calls to get the list of chats. We'll look at how to do that in more detail later on. Then we'll create a vertical panel in the onModuleLoad method. Finally, we'll call a populateChats method, which will use a GWT asynchronous RPC to retrieve the list of chats and populate the list. The implementation is:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
/**
 * Sets up a call and callback to retrieve the list of available chats. When
 * the server responds, this will create the link widgets and add them to
 * the chatListPanel.
 *
 * @param chatListPanel
 */
public void populateChats(final VerticalPanel chatListPanel) {
    chatService.getChats(new AsyncCallback<List<String>>() {
        public void onFailure(Throwable caught) {
            chatListPanel.add(new Label("Couldn't retrieve chats: " + caught));
        }

        public void onSuccess(List<String> chats) {
            for (String chat : chats) {
                Button chatButton = new Button(chat);
                chatListPanel.add(chatButton);
                Chat.this.setupChatClickHandler(chatButton,
                        chat);
            }
            setCurrentChat(chats.get(0));

        }
    });
}
```

This is a relatively straightforward use of RPC and asynchronous operations: it's the same basic pattern you'll use over and over again in App Engine/GWT.

❶ We make an RPC call. Instead of actively waiting for a response and then returning it, we'll set up a callback object, which will be invoked whenever the result of the RPC becomes available. This pattern is used a lot and not only for RPC. Because of the lack of threading, almost anywhere we'd use threads in a typical Java program, we set up some kind of a callback in App Engine. Nearly all of the real code of our application will run in callbacks.

❷ Most of the time, the RPC should succeed. As you'll see when we write the server-side code, there's no way that our server will return an error. But in cloud programs there's always a layer between your client and your server—the network itself. And that's a potential source of errors that's beyond your control, so program carefully to make sure that you're prepared for it. In this case, we'll handle it in a really simple way: if the getChats call fails, we'll just put an error message into the chat-list widget.

❸ If the RPC succeeds as expected, then we populate the chat list panel. For each chat, we create a Button widget containing the name of the chat and add it to the panel.

❹ Then we need to set up event handlers. We want the UI to do something when we click on one of those chat buttons. To do that, we need to set handlers. We'll look at how to do that in the next section.

In order to make our application take action in response to events, we need to set up *event handlers*. Event handlers are callbacks that are invoked whenever the user does something that your program is interested in. For example, we just made a bunch of buttons for selecting chats. But those buttons don't do anything. So try this:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
protected void setupChatClickHandler(final Button chatButton, final String chat) {
    chatButton.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
            setCurrentChat(chat);
            text.setText("Current chat: " + chat + "\n");
            currentChat = chat;
            chatService.getMessages(currentChat, new MessageListCallback());
        }
    });
}
```

This attaches a handler to a button. The handler is an instance of ClickHandler. The onClick method of the handler will be called whenever the button is clicked. When the user clicks the button for a particular chat, the handler will set the current chat for the program, and it will invoke an RPC that will retrieve the messages from the selected chat.

There are more event handlers that we need to write. When the user tries to send a chat message by clicking the Send button, something is supposed to happen: the system is supposed to take the contents of the text entry box, send it to the server as a new chat message, update the transcript, and clear the entry area to get it ready for the next message.

We'll set up a callback that is invoked when users click on the Send button or when they press the Enter key on text in the message entry area:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
private void setupSendMessageHandlers(final Button sendButton,
        final TextBox messageBox) {
    // Create a handler for the sendButton and nameField
```

```
❶       class SendMessageHandler implements ClickHandler,
                KeyUpHandler {
            /** Fired when the user clicks on the sendButton. */
❷           public void onClick(ClickEvent event) {
                sendMessageToServer();
            }

            /** Fired when the user types in the nameField. */
❸           public void onKeyUp(KeyUpEvent event) {
                if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
                    sendMessageToServer();
                }
            }

            /** Send a chat message to the server. */
❹           private void sendMessageToServer() {
                ChatMessage chatmsg = new ChatMessage(user,
                        messageBox.getText(), getCurrentChat());
                messageBox.setText("");
                chatService.postMessage(chatmsg,
                        new AsyncCallback<Void>() {
                    public void onFailure(Throwable caught) {
                        Chat.this.addNewMessage(new ChatMessage(
                                "System", "Error sending message: " +
                                caught.getMessage(),
                                getCurrentChat()));
                    }

                    public void onSuccess(Void v) {
                        chatService.getMessagesSince(getCurrentChat(),
                                lastMessageTime,
                                new MessageListCallback());
                    }
                });
            }
        }
❺       SendMessageHandler handler = new SendMessageHandler();
        sendButton.addClickHandler(handler);
        messageBox.addKeyUpHandler(handler);
    }
```

❶ We're writing a handler that will be invoked for two kinds of events: when the users click the Send button and when they hit the Enter key in the message box. We need to implement an interface to handle each event: for a button click, there's the ClickHandler interface; for the Enter key, there's the KeyUpHandler.

❷ The ClickHandler interface has a method onClick; your handler's onClick will be invoked whenever the button is clicked. The button click handler and the Enter key handler do the same thing, so we abstract that into a single function.

❸ The KeyUpHandler interface has a method onKeyUp that is invoked whenever a key is pressed. The main difference is that onKeyUp will be invoked whenever *any* key is pressed, but we only want to send the message when the Enter key is pressed. So we need to do a test to check what key was pressed and only do the send when the Enter key is pressed.

❹ Here's where we do the real work. First, we create the chat message. Then, using the usual GWT asynchronous style, we invoke the GWT RPC message.

❺ Finally, we create an instance of the callback object, and we register it as the event handler for clicking the Send button or pressing Enter in the text box.

11.4 Making the UI Active: Updating the Display

We've laid out our user interface, and we've built callbacks and event handlers to make it actually do things in response to user actions. But we're still missing one key bit—updating the display. When a user selects a chat room, the UI is supposed to update so that it displays the messages in that chat room, and then it's supposed to keep updating whenever new messages get posted.

In the handler code above, in the chat selection handler (and in one or two other places), we created a MessageListCallback. That's actually the code that updates the display with collections of new messages. The MessageListCallback is used in two ways:

1. When the user selects a new chat, it's invoked to display the new messages. In that case, it retrieves a complete list of all messages in the chat room up to the current moment.
2. In order to keep the UI up-to-date, we have a scheduled callback that keeps retrieving new messages: it doesn't get the entire list of all messages in a chat; it just gets the ones that haven't been seen yet by this client. (We'll look at how it determines what has or hasn't been seen yet by the client in the next chapter.)

Updating the UI is *really* easy. Basically, all of your UI widgets have methods to change their contents, and all you need to do is convert things into strings and then add them to the widgets. That's all that you need to do.

So let's take a look at the MessageListCallback:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
public class MessageListCallback implements AsyncCallback<List<ChatMessage>> {

    public void onFailure(Throwable caught) {
    }

    public void onSuccess(List<ChatMessage> result) {
        addNewMessages(result);
    }
}

protected void addNewMessages(List<ChatMessage> newMessages) {
    StringBuilder content = new StringBuilder();
    content.append(text.getText());
    for (ChatMessage cm : newMessages) {
        content.append(renderChatMessage(cm));
    }
    text.setText(content.toString());
}

protected String renderChatMessage(ChatMessage msg) {
    Date d = new Date(msg.getDate());
    String dateStr = d.getMonth() + "/" + d.getDate() + " " +
        d.getHours() + ":" + d.getMinutes() + "." +
        d.getSeconds();
    return "[From: " + msg.getSenderName() + " at " +
        dateStr  + "]: " + msg.getMessage() + "\n";
}

protected void addNewMessage(ChatMessage newMessage) {
    text.setText(text.getText() + renderChatMessage(newMessage));
}
```

1. The callback itself is trivial: it receives the list of messages to add to the display, and it just invokes addNewMessages.

2. The implementation of addNewMessages is nearly as simple. It renders the chat messages as strings, concatenates the string containing the new messages with what was already in the transcript window, and then sets the window contents.

There's one more piece to really making the UI active the way we'd like it to. We want the UI to automatically update when other users post new messages. In a cloud environment, we can't really do something where we say, "Update whenever someone else posts a message": our client doesn't know when someone else has posted a message. There's no way for it to know: only the server knows, and the server can only

respond to requests from the client. So we create a periodic update: we set up a request that will automatically be sent to the server on a regular schedule asking it for any updates. This is a very common pattern in GWT UIs, so GWT makes it easy to do. And, of course, it's basically another callback, called a Timer. A Timer is a runnable object, which GWT will invoke on a schedule. We create and set up the timer as the last section of our onModuleLoad method:

Download workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
private void setupTimedUpdate() {
// Create a new timer
Timer elapsedTimer = new Timer() {
    public void run() {
        chatService.getMessagesSince(getCurrentChat(), lastMessageTime,
                new MessageListCallback());
    }
};
// Schedule the timer for every 1/2 second (500 milliseconds)
elapsedTimer.scheduleRepeating(500);
}
```

The creation of the timer object is absolutely bog-standard GWT-style callback code. We create a Timer object. When invoked, the timer sends a request to the server asking for updates. In more typical GWT code, it provides yet another callback—this time, one that will be invoked when the update request returns a result. When the getMessagesSince call returns, the new messages will be displayed by calling addNewMessages.

Once the Timer object is created, we just need to tell it how often it should be invoked. For a responsive UI, one-half of a second is a pretty good interval, so we tell it to invoke every 500 milliseconds.

11.5 Wrapping Up with GWT

In this chapter, we put together the basic GUI of our chat application. We examined the basic mechanisms for building GWT applications. We saw how to create widgets and arrange them in the UI. We looked at how to use CSS to customize styles of elements of the UI. We examined how to create event handlers and attach them to widgets to make the UI active. In fact, we've seen everything we need to build a complete App Engine program in Java.

In the next chapter, we'll put it all together. We'll finish filling in the gaps in our server and in the client-side logic that interacts with the server. In the process, we'll look at some more of the interesting services that

GWT provides. It should all look familiar: GWT uses a continuation-passing callback style pretty ubiquitously. Most services are provided through hooks where we can attach our own callbacks. By the end of the next chapter, our chat application will be completely built and deployed to App Engine.

11.6 References and Resources

The GWT Widget Gallery. . .

. . . http://code.google.com/webtoolkit/doc/1.6/RefWidgetGallery.html

The single most useful resource for a GWT UI builder. This site is an up-to-date list of every GWT widget, with visual examples of what the widgets look like, complete specifications of what CSS attributes you can use to customize them, and lists of the event handlers that you can use to program them.

The GWT 2.0 Developers Guide. . .

. . . http://code.google.com/webtoolkit/doc/latest/DevGuide.html

The official GWT documentation, describing everything you could want to know about GWT.

Chapter 12

Building the Server Side of a Java Application

We've built most of our chat application; what's left is the connection between the client and the server. Back in Chapter 10, *Managing Server-Side Data*, on page 141, we put together a basic RPC interface for connection between the client and the server. Unfortunately, it turned out to be not quite right once we built our client. In this chapter, we'll look at what we did wrong in defining that interface and how to fix it. We'll look a bit more in-depth at what kinds of objects and methods we really need in our server, and we'll implement the missing pieces. We'll examine we else we need in our server beyond just the handlers for client requests. And we'll finally deploy our Java chat application.

12.1 Filling in Gaps: Supporting Chat Rooms

If you were paying attention in the last chapter, you'll notice that I cheated in some of the client methods. The RPC interface that I defined in Chapter 10, *Managing Server-Side Data*, on page 141, only had two methods, but I used a couple of extras. The original RPC interface just didn't have everything that I really needed to make the application work. That kind of problem is really common when you're getting started with a system like App Engine.

Programming distributed applications is very different from traditional application program. When you're not used to thinking in terms of such an extreme separation of the system into client and server pieces, it's very easy to forget about some of the basic things that you'll need.

When we designed our original chat interface, we focused completely on how to get and post chat messages. After all, those are the two fundamental operations in a chat application.

But there's more to chat. The main activities of a user of a chat application are posting and reading messages. However, in order to post and read messages, the users need to be able to see what chats are available and pick one. We need to cover the complete life cycle of our application. We can't just focus on one or two key activities: we need to think about how our users will get to the point where they can do those activities, and we need to think about how we'll set up the infrastructure we need in order to make those activities possible.

In terms of our chat application, we're missing two important things:

- We need a way of getting the list of available chat rooms, so users can choose the one in which they want to participate.
- We need a way of *creating* chat rooms. When we first deploy our application to the server, our datastore will be completely empty. There won't be any rooms. In order to make a usable application, we need to either seed the datastore by putting in a set of chat rooms or provide some way for users to define new rooms. Either way, we must provide a call to create a chat room.

Let's get started on creating those missing pieces.

Implementing ChatRoom Classes

We need to add a couple of new RPC methods to our interface, but we're missing data types. As we just discussed, we need to be able to create and query the list of chat rooms: to do that, we must have a persistent ChatRoom class.

What should a ChatRoom object look like? To be able to create and query rooms, we don't need much: in fact, the only thing we really need is the name of the chat.

But the first time we tried to define the interface, we messed up—e left out things that we really need. Rather than rushing ahead, this time we'll be more careful and make sure we get the design of the ChatRoom class completely right. What sorts of things might we want to have in the chat list? One thing that comes to mind is a timestamp containing the time at which the last message was posted to the chat. With that, we could have the UI show users which chats are active. So we'll write

a persistent chat object that contains both the name of the chat and the timestamp of the last posted message. There's nothing particularly new about this class. It's a typical persistent App Engine object, so we won't discuss it in detail. Here's the code:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatRoom.java

```
public class ChatRoom implements IsSerializable {

    String name;
    long   date;

    public ChatRoom(String chat, long date) {
        this.date = date;
        this.name = chat;
    }

    public ChatRoom() {
    }

    public String getName() {
        return name;
    }

    public long getLastMessageDate() {
        return date;
    }

    public void updateLastMessageDate(long d) {
        date = d;
    }
}
```

Persistent Classes and GWT: Ouch

And now, we encounter one of the few places where GWT's strategy of translating Java to JavaScript automatically causes real trouble. We've got classes for chat messages and chat rooms, which we use both in RPC calls and in the server datastore persistence code. The problem is that to use a class in a persistent way, it needs to have some persistence annotations, and it needs a persistence key. But the annotations and the key aren't things that GWT can translate into JavaScript! So we can't use the persistent classes for GWT RPC.

We can't use the GWT RPC version of those classes for persistence either because the RPC classes don't have the annotations and fields needed by the datastore JDO implementation. So we need to have two versions of each of those classes: one for persistence and one for RPC. It's annoying, but it's the easiest way of working around this painful problem.

We put the RPC version of the ChatMessage and ChatRoom classes in the client package, and we put the persistent version of them into the server package. To make things easier to read, we prefix the names of the persistent version of the classes with *P*; you'll see why later, where we need to do the work to convert between the two different versions.

For now, so that you can see the difference, let's look at the persistent version of our new ChatRoom. We just saw the RPC version of it, and here's the datastore persistent PChatRoom:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/PChatRoom.java

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class PChatRoom {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    String name;

    @Persistent
    long date;

    public PChatRoom() {
    }

    public PChatRoom(String chat, long date) {
        this.date = date;
        this.name = chat;
    }

    public ChatRoom asChatRoom() {
        return new ChatRoom(name, date);
    }

    public String getName() {
        return name;
    }

    public Key getKey() {
        return key;
    }

    public long getLastMessageDate() {
        return date;
    }

    public void updateLastMessageDate(long d) {
        date = d;
    }
}
```

The Server ChatRoom Methods

Now that we have the ChatRoom class done, we can write the methods that will work with it. What do we really want to be able to do with ChatRooms?

Create rooms.
: We need to be able to create chat rooms.

List rooms.
: We need to be able to get a list of the available chat rooms.

Delete rooms.
: This one is optional; it depends on just how we want our system to behave. If we think that chats are intrinsically transient—that is, constantly created and then discarded—we'll want a way to clean up and discard chats once they're no longer being used. On the other hand, if we want chats to be a permanent record of an ongoing conversation, we shouldn't delete them.

  The way that I tend to use chat rooms is much more the latter: I consider most chats to be part of an ongoing conversation. I may stop talking today, but odds are, I'll come back tomorrow when I have more to say. Since I'm designing the application, I'll leave out Delete.

We know what methods we want, so now we can add them to the ChatService. Our new methods are shown below:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatService.java

```
List<ChatRoom> getChats();

void addChat(String chatname);
```

As always in GWT, we need to add corresponding methods to the asynchronous version of the interface:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatServiceAsync.java

```
    void getChats(AsyncCallback<List<ChatRoom>> chats);

void addChat(String chatname,
        AsyncCallback<Void> callback);
```

Implementing these is more of the same. getChats is pretty much the simplest possible JDOQL query: fetch all objects of type ChatRoom and return them, as shown in the following:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatRoom> getChats() {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(ChatRoom.class);
        query.setOrdering("date");
        return (List<ChatRoom>)query.execute();
    } finally {
        persister.close();
    }
}
```

Adding a new chat is just a bit more complicated. We need to create a chat object and make it persistent:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
public void addChat(String chat) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        PChatRoom newchat =
            new PChatRoom(chat, System.currentTimeMillis());
        persister.makePersistent(newchat);
    } finally {
        persister.close();
    }
}
```

12.2 Proper Interactive Design: Being Incremental

There's one other big problem with our original interface. We only provided a message for getting *all* of the messages in a given chat. We're building a UI where chats never get deleted—messages just keep accumulating. And we want the application to be interactive, which means it's going to constantly update the displayed list of messages. When we put those two together, we've got a formula for wasted resources and lousy performance. We're going to be constantly resending the entire list of messages. Even assuming we only retrieve the message list when we post a new message, that means on our client we're going to retrieve message one the first time we post; message one and message two the second time; messages one, two, and three the third time; then one, two, three, and four—and so on. By the tenth message, we'll have sent message one 10 times. Remember, it's going to automatically update twice a second: in one minute of using the system, we'd fetch the same messages 120 times!

This is *not* the way a cloud application should work. Imaging scaling this: an application isn't going to have one user; it's going to have hundreds of users, thousands, or more! With a thousand users, we'd be resending the same old messages 120,000 times per second. Not only is that a waste of time, it's also going to cost money! In App Engine, we pay for the resources we use. If we resend the same thing a 100,000 times a second, we're going to use up our free resources quickly, and then we'll need to start paying for bandwidth.

There's absolutely no good reason for doing that. In a real cloud application, we don't worry very much about CPU time: CPU time is cheap. But we focus on communication: communication is expensive in both time and money. Sending things over the network is incredibly slow in comparison to computing them ourselves. Most of the time, we should focus our design work on minimizing communication. It's faster and cheaper to recompute the same thing many times than it is to send it over the network once.

For our chat room, there's no reason to ever resend a message to a single client. The client can remember the messages it's already seen and just add new ones to its list. In a cloud implementation of this, what we can do is retrieve updates of the list based on time—that is, we can say, "Give me all of the chat messages since the last time I asked."

This way of working is called an *incremental* update. Instead of refetching an entire collection of data, we manage copies of the data on both the client and the server and only send minimal modifications. For pretty much any cloud application, if we want it to be both efficient and affordable to operate, we need to design it for incremental updates.

Data Objects for Incremental Updates

To code incremental updates, we usually need to build a data structures to represent how an incremental update fits into the structure as a whole. In our case, what that means is that we can't just return a list of ChatMessages from the getMessages or getMessagesSince methods: we need to return a list of messages *and* a timestamp. So we need to create a new object type that wraps those two things. It doesn't need to be a persistent object: we're never going to store it in the datastore—we generate it on the fly in response to a request from the client. But it does need to be something that can be sent over the network. It needs to be *serializable*, which means that GWT will generate code to both translate the object into a format that can be sent in a message and translate

messages back into the object. In GWT, we make an object serializable by having it implement the interface IsSerializable. This interface has no methods: it's just a marker used to tell GWT that it needs to generate code for serializing this object.[1]

With all of that out of the way, it's pretty easy to write a serializable object—just make it declare that it implements IsSerializable, and make sure that it includes a default, no-argument constructor. So our serializable object simply wraps a list of ChatMessages and a date object that we'll use as a timestamp. We want GWT to translate this. The easiest way to do that is to just put it in the client package.

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatMessageList.java

```
public class ChatMessageList implements IsSerializable {

    private List<ChatMessage> messages;
    private long time;
    private String chat;

    public ChatMessageList(String chat, long time) {
      this.chat = chat;
      this.time = time;
      this.messages = new ArrayList<ChatMessage>();
    }

    /**
     * Default 0-argument constructor for GWT serialization.
     */
    public ChatMessageList() {
        messages = new ArrayList<ChatMessage>();
        time = System.currentTimeMillis();
        chat = null;
    }

    public String getChat() {
        return chat;
    }

    public List<ChatMessage> getMessages() {
        return messages;
    }
```

1. We can actually use the standard Java interface Serializable for this. But Serializable is used by the Java virtual machine to identify things that can be serialized using the native Java serialization mechanism. GWT doesn't use Java serialization—in fact, GWT serialization doesn't even resemble standard Java serialization. So I prefer to use GWT's own marker interface to make it clear that I'm using GWT serialization.

```
    public long getTimestamp() {
        return time;
    }

    public void addMessage(ChatMessage msg) {
        messages.add(msg);
    }

    public void addMessages(List<ChatMessage> messages) {
        messages.addAll(messages);
    }
}
```

Making the Chat Interface Incremental

To start using this time-based incremental retrieval mechanism, we'll need to both modify some old methods and add some new methods to our interface:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatService.java

```
❶ void postMessage(ChatMessage messages);
❷ ChatMessageList getMessages(String room);
❸ ChatMessageList getMessagesSince(String chat, long timestamp);
```

❶ Our postMessage method originally returned the list of messages in a chat. But the reason we wanted to add a method to get the messages since a particular point in time (that is, that we don't want to send the same old messages over and over again) applies just as well to the result of the postMessage method as it does to the result of getMessages. We have two choices: we can either add a timestamp parameter to postMessage, or we can make postMessage return no value and instead have our client call getMessages after a new message is posted.

As a general rule, it's good to separate query methods (methods that retrieve values) from update methods (methods that modify values). We'll follow that rule and make postMessage have a void return value.

❷ When a user first connects to a chat room, the client should fetch all of the messages in that room. After that, it will start doing the incremental fetches. In order to make that work, the client needs to know the server time when they did their first fetch. We must modify the return type of the getMessages call so that it returns a ChatMessageList.

❸ Now we finally get to a new method: getMessagesSince.

And as usual for GWT, we need to provide an asynchronous version:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/ChatServiceAsync.java

```
void  postMessage(ChatMessage message,
                  AsyncCallback<Void> callback);

    void getMessages(String chatroom,
                  AsyncCallback<ChatMessageList> callback);

    void getMessagesSince(String chat, long timestamp,
                  AsyncCallback<ChatMessageList> callback);
```

Dealing with Time

Time-based work in the cloud is a bit tricky. Our client isn't running on the same machine as our server. In fact, we don't have *a* server; our server code might be running on lots of different machines in the cloud. There is no guarantee that the clocks used by our client and our server are in sync. The time the client thinks it last retrieved messages might not be the same time the server thinks it was. To make matters worse, the fact that things are happening on a network adds a delay, which can make timing-based things even more complicated. This is potentially a serious and painful problem—fortunately, once you understand what's going on, it's not too difficult to fix. Let's begin by examining the problem itself.

Suppose the clocks on the client and the server are perfectly in sync. We could still run into conflicts as a result of simple timing issues caused by network delays. Imagine this scenario:

09:34:58.1432 The client sends a request for messages.

09:23:58.1894 The server receives that request after a slight delay for transmission time over the network.

09:23:59.2401 The server processes the request and responds.

09:24:59.4019 The client receives the response over the network.

What time should the client use for the time of its request? Suppose that it used the time it sent the request: 58.1432. Another client could have sent message X at 58.1434. When the server receives the request, it will return a list containing X. And the next time the client asks for messages, it will ask for messages newer than 58.1432. Since X's timestamp is 58.1434, it's newer than 58.1432, and so the server will include X in its response again. That means the client will see message X twice. That's clearly not what we want.

We could use the time that the client receives the response: 59.4019. Perhaps another client sent message Y at 59.2530. Message Y will *not* be in the list of messages received by the client because it was posted after the server sent its response to the client. The client's next request will be for messages posted after 59.4019; since Y's timestamp is 59.2530, that would mean it wouldn't be included. Message Y will *never* be sent to the client. Once again, that's clearly not what we want.

It seems like no matter what we choose, we're going to miss messages. In general, the solution is to use a single clock: anything in a cloud application that relies on two different clocks is bad news. We must always work in terms of one clock. We can't work in terms of the client's clock, because client clocks are out of our control: there will be multiple clients, and there's no way to know whether they're anything close to being synchronized. But we *can* rely on the clocks in the App Engine servers; they're synchronized using the network time protocol. We can be certain that the differences between their clocks are smaller than the units that we'll be measuring time in.

The server clock—the clock provided by App Engine in our server code—should be the only clock we use. And that means we need to tell the client what time it was on the server when a request was served.

Implementing the Server Methods

Now that we've got the interface between the client and the server worked out, let's implement the server code. We implemented parts of it before, but we've changed things and added more methods. We're almost starting from scratch in our server.

We can begin with getMessages. This is nearly the same as before, only now we wrap the result in a MessageList object with a timestamp.

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public ChatMessageList getMessages(String chat) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatMessage.class);
        query.setFilter("chat == desiredRoom");
        query.declareParameters("String desiredRoom");
        query.setOrdering("date");
        List<PChatMessage> messages =  (List<PChatMessage>)query.execute(chat);
        // Get the most recent message.
        ChatMessageList result = null;
        if (messages.size() > 1) {
```

```
❶               PChatMessage lastMessage = messages.get(messages.size() - 1);
                result = new ChatMessageList(chat, lastMessage.getDate());
                for (PChatMessage pchatmsg : messages) {
                result.addMessage(pchatmsg.asChatMessage());
                }
            } else {
                result = new ChatMessageList(chat, System.currentTimeMillis());
            }
            return result;
        } finally {
            persister.close();
        }
    }
```

There's one interesting snippet in this code, which is related to the discussion about time in Section 12.2, *Dealing with Time*, on page 180. At ❶, we get the last message retrieved by the query and use its timestamp as the time. The reason that we do this instead of using the system clock is because our app isn't running on one server. There could be one server in the cloud handling a **POST** request at the same time that this server is handling a **GET** request, and that could result in a new post being added between the time the query finishes and when the getMessages implementation gets the time. So there could be a message whose timestamp is *between* the last retrieved message and the current time recorded by getMessages. In order to avoid that situation, we just use the time of the last message. This strategy gives us a consistent view of times so that our incremental message retrieval works properly.

Next, we can look at posting a message. That's where things start to get really interesting.

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
    @SuppressWarnings("unchecked")
public void postMessage(ChatMessage message) {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
    PersistenceManager persister = Persister.getPersistenceManager();
        try {
            PChatMessage pmessage = new PChatMessage(user.getNickname(),
                                                     message.getMessage(),
                                                     message.getChat());
❶           long timestamp = System.currentTimeMillis();
            pmessage.setDate(timestamp);
            persister.makePersistent(pmessage);

❷           Query query = persister.newQuery(PChatRoom.class);
            query.setFilter("name == " + message.getChat());
            List<PChatRoom> chats = (List<PChatRoom>) query.execute();
```

```
❸               PChatRoom chat = chats.get(0);
❹               chat.updateLastMessageDate(timestamp);
            } finally {
                persister.close();
            }
        }
```

We start off with some typical App Engine boilerplate: create a PersistenceManager, and make the chat message object persistent so that it will be stored in the datastore. Then we get to the new stuff.

❶ We modify the date on the message. As I explained before, we have to be very careful to only use one clock—App Engine's clock. We don't care what the client thought the date and time was; we need to use the date here on the server.

❷ Our Chat objects must include the timestamp of the last posted message. So we need to retrieve the chat object in order to update its last-message time to the time of this message.

❸ The query returns a list of chats, but we know that the list will only have one entry, so we can just grab the only entry out of the list.

❹ Now we update the last message date in the chat object. We don't need to do anything to store it: since we retrieved the message using a PersistenceManager query, it's already managed by the PersistenceManager. And that means the updates will automatically be saved at the end of the transaction.

Finally, we need the incremental getMessagesSince:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public ChatMessageList getMessagesSince(String chat, long timestamp) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatMessage.class);
❶       query.declareParameters("String desiredRoom, int earliest");
❷       query.setFilter("chat == desiredRoom && date > earliest");
        query.setOrdering("date");
        List<PChatMessage> messages =
            (List<PChatMessage>)query.execute(chat, timestamp);
        ChatMessageList msgList = null;
        // Get the most recent message.
        if (messages.size() >= 1) {
            PChatMessage lastMessage = messages.get(messages.size() - 1);
            msgList = new ChatMessageList(chat, lastMessage.getDate());
        } else {
```

```
            msgList = new ChatMessageList(chat, System.currentTimeMillis());
        }
        for (PChatMessage msg : messages) {
            msgList.addMessage(msg.asChatMessage());
        }
        return msgList;
    } finally {
        persister.close();
    }
}
```

This is almost the same as the updated getMessages, except for two changes to the JDOQL query:

❶ We added a new parameter to the query, so that we can compare the date passed to the call to getMessagesSince with the dates on the messages.

❷ We added a new clause to the filter to compare the dates.

12.3 Updating the Client

Now we've got all of the server methods we need for interacting with our client! Our application is very nearly ready to run. What we still need to do is make a couple of changes to our client so that it knows how to use our updated RPC interfaces.

There's not much to do here: basically, the only problem is that we need to update our code to adapt to the changes we made in the RPC service. Instead of getting a list of chat messages, it now gets a ChatMessageList, and now whenever it receives a ChatMessageList, it needs to update its stored last message time so that the client can use it to make incremental update requests. All we need to do for that is update our addNewMessages method:

Download workspace/Chat/src/com/pragprog/aebook/chat/client/Chat.java

```
protected void addNewMessages(ChatMessageList newMessages) {
    lastMessageTime = newMessages.getTimestamp();
    StringBuilder content = new StringBuilder();
    content.append(text.getText());
    for (ChatMessage cm : newMessages.getMessages()) {
        content.append(renderChatMessage(cm));
    }
    text.setText(content.toString());
}
```

And then, we also need to update the code that creates a timed update so that it uses the lastMessageTime in its incremental update request:

```
// Create a new timer
Timer elapsedTimer = new Timer() {
    public void run() {
        chatService.getMessagesSince(getCurrentChat(), lastMessageTime,
                new MessageListCallback());
    }
};
// Schedule the timer for every 1/2 second (500 milliseconds)
elapsedTimer.scheduleRepeating(500);
```

12.4 Chat Administration

There's one last thing to do before we're ready to run our chat application. We provided a method for creating chats on the server, but we don't have any interface for doing it. For now, we're not going to build a new UI element for adding chats. Instead, we'll write a bit of *administration code*. Administration code is code that you write to set up, clean up, initialize, or monitor things. It's code that's not there for our users to access but for us to use to manage our system.

A lot of times, you'll build a UI for your administration code. For example, if you wanted to be able to keep track of how many people were accessing chat, how many messages they posted to which chat rooms, and how frequently people accessed the system, you'd probably want to have a UI. Then you'd build another GWT user interface for doing administration; to administer things, you'd just load the URL for the administrator's UI in your browser. If you did that, you'd probably want to be careful to provide some security mechanisms, which we'll see later, to make sure that only you could access the administration UI. But there are other kinds of administration code, which is what we're interested in here. When we first deploy a server, we need to initialize some things. In our chat application, we need to initialize the set of chat rooms.

The problem is we can't just run code on the server to set things up. We're limited to the interface provided by App Engine. What we need to do is *self-detecting initialization*. That is, we need to put code in to detect when the server is not yet initialized and call the initialization method.

We do that by modifying getChats. Every time users come to our application but before they see anything else, the application will call getChats

to initialize the chat list view. So what we'll do is simple. When the application retrieves the list of chat rooms, we'll have it check to see if the result is empty. If it is, we'll call a method to initialize a set of chat rooms. We'll put the initialization method in the server implementation, but we won't declare it as an RPC. We don't want users to be able to invoke it: the only way it should ever be invoked is by the automatic invocation when the system detects that it hasn't been initialized.

The code to initialize the chat is pretty simple. We create a PersistenceManager, create the rooms, and make them persistent:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
static final String[] DEFAULT_ROOMS =
    new String[] { "chat", "book", "java", "python"  };

public List<ChatRoom> initializeChats(PersistenceManager persister) {
    List<ChatRoom> rooms = new ArrayList<ChatRoom>();
    List<PChatRoom> prooms = new ArrayList<PChatRoom>();
    long now = System.currentTimeMillis();
    for (String name : DEFAULT_ROOMS) {
        PChatRoom r = new PChatRoom(name, now);
        prooms.add(r);
        rooms.add(r.asChatRoom());
        persister.makePersistent(r);
    }
    return rooms;
}
```

To call it, we just add a test to getChats, which checks to see if the list of chats in the datastore is empty. If it is, we make it call initializeChats:

Download workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatRoom> getChats() {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatRoom.class);
        query.setOrdering("date");
        List<PChatRoom> rooms = (List<PChatRoom>)query.execute();
        if (rooms.isEmpty()) {
            return initializeChats(persister);
        } else {
            List<ChatRoom> result = new ArrayList<ChatRoom>();
            for (PChatRoom pchatroom : rooms) {
                result.add(pchatroom.asChatRoom());
            }
            return result;
        }
    }
```

```
        finally {
            persister.close();
        }
    }
```

12.5 Running and Deploying the Chat Application

We've finally finished a Java version of the chat room! Now it's time to test it. In Eclipse, you just run your application using the Run button. It takes a few moments to start up, and then it will show you a URL to use to access the test server. When you go to that URL, you'll be asked to install a browser plugin: for testing purposes, it uses a special mechanism to manage communication between the client and the server so that you can use the Eclipse debugger to trace things in both the client and the server. (In fact, it will even let you trace an RPC call from the point where it's invoked on the client to where it's actually executed on the server.) Eclipse makes debugging an App Engine program nearly as simple as a traditional application.

Before you can deploy your Java code to the App Engine servers, you need to do a complete GWT compile. As usual, Eclipse makes things easy. First, you need to do a full GWT compile. Normally, Eclipse is doing partial compilation—basically just using the Java compiler in a restricted mode. To deploy, you need to do a full compile with full translation of Java to JavaScript. Up on the Eclipse toolbar, there's a button that looks like a red toolbox with a Google *G*: just click on that, and Eclipse will do a full-blown GWT compilation.

And now, finally, our application is ready to go live! Right next to the toolbar button that you used to compile the application, there's a button that looks like the App Engine jet-engine logo. Just click on that. The first time you do that, it will prompt you for the App Engine applicatin ID, and then it will do the deployment. It takes a few minutes to do a full deployment, but once it's done, your application is live on App Engine.

So, after all of this work, what does it look like? Cast your eyes over Figure 12.1, on the following page.

Pretty sharp, eh?

Just like with the Python applications, you now have full access to all of the administrative controls on the application dashboard.

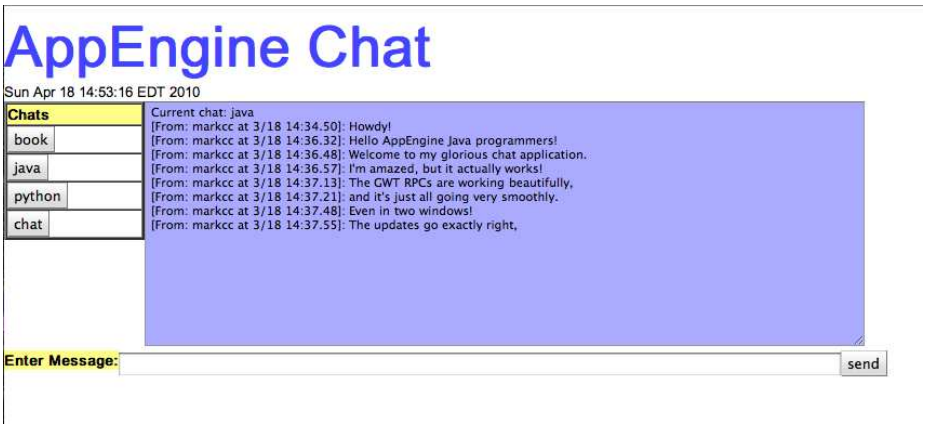


Figure 12.1: The Java chat room, deployed

Troubleshooting

When you go to deploy your application, there are a couple of problems that occur frequently but whose error messages are at best bizarre and mysterious and at worst highly misleading. Here are two of the most common strangenesses:

- When you change something about the way you store data in the datastore, you'll often get all sorts of strange error messages, such as class cast exceptions and "unable to convert data type." They're caused because your datastore contains data from before you made the change, and so the data in the datastore is inconsistent. Hopefully you'll figure out your data structures before you deploy the application. But while you're working on it, figuring things out, you'll frequently need to change things. For example, when I was working on the code for this chapter, I originally stored the date as a java.util.Date object. But that turned out to be a problem for GWT RPC, so I changed the representation of a timestamp to be a long containing the timestamp in milliseconds since the epoch.

  Those errors are not caused by any problem in your code. They're entirely the result of old data left in your local datastore. To fix it, what you need to do is empty the old data out of your local datastore. To do that, you just need to delete the contents of the directory WEB-INF/appengine-generated/local_db.bin in your Java project.

If you actually deployed the application, then you need to go to the application dashboard and select Datastore Viewer. From there, you can do a Select All and then delete all of the objects that you want to purge.

- One of the most mysterious and yet common errors in Java App Engine occurs in GWT RPC parameters. When you go to run your program, you get the error message, "*type* was not included in the set of types which can be serialized by this SerializationPolicy." This seems to imply that there's a configuration problem, but the error is very misleading. All that's wrong is that you left out a zero-argument constructor for the type. Every serializable type used by GWT must have a public, zero-argument constructor. It doesn't need to do any actual initialization; it's just used by the GWT plumbing to create a blank object that GWT will then fill in with the results of deserializing the message.

12.6 Wrapping Up the Server Side

We've covered a lot of material in the last few chapters. We've seen how to do datastore persistence in Java. We've learned a whole lot about GWT and how to design applications using its model. We've done a lot of work on how to build an RPC-based interface between the client and server and how to use that interface to produce a well-performing web application.

From here, we're going to start moving into more advanced topics. Instead of focusing specifically on Java or Python, we're going to look at a bunch of different topics: security and authentication, advanced data management, administration, and monitoring. For each one, we'll look at how to do it in both Java and Python.

Part IV

Advanced Google App Engine

Chapter 13

Advanced Datastore: Property Types

As we've been learning to program Google App Engine, we've done some basic things with the datastore. We haven't done anything particularly difficult, but even in our simple applications, datastore is at the heart of our designs. As you start building more complicated App Engine programs, you'll find that the datastore becomes even more important.

The datastore can do a whole lot more than what we've seen so far. It's an extremely powerful and flexible persistent storage system. In this chapter, we'll look at the capabilities of the datastore and experiment with some of its more advanced features.

In this chapter, we're going to start to take a different approach. We're not going to limit ourselves to either Java or Python; instead, we're going to look at both. We're also not going to be building UIs: we're going to focus on data management. One of the things we'll see when we do that is we can build services for our App Engine applications using HTTP in a way that allows us to implement each service in whatever programming language is most convenient. We can write Python services that use Java services, and vice versa, using an HTTP layer as a bridge between the languages.

13.1 Building a Filesystem Service

Throughout the advanced topics section, instead of continuing to focus on our chat application, we'll be building different kinds of services to

RESTful Programming

In this chapter, we'll be building things around a model called REST. What we'll be doing is often called RESTful programming. REST stands for representational state transfer, which is a fancy way of saying that the program is using web-based protocols the way they were originally intended—for accessing and updating resources on the web.

REST has become a buzzword—everyone wants to claim their system is RESTful and their technologies are the best way of doing REST. But it's really not complicated.

What REST means is you use the basic HTTP primitives (**GET**, **PUT**, and **POST**) exactly the way they were originally intended. To retrieve data, you always use **GET**. To store a complete object, you use **PUT**. To update data, you use **POST**. In general in REST, you don't use things like CGI parameters for **GET** or **PUT**: a URL identifies a specific resource, and since **GET** means "retrieve a resource," you shouldn't need anything more than just the URL to do it.

REST is great because of its simplicity. An lot of web-based programming has been done in incredibly sloppy ways. There are still huge numbers of web and cloud applications that use **GET** with long strings of CGI parameters to do all sorts of things besides retrieve data. In many ways, REST is almost like object orientation for the Web: it's a way of programming that's built around the fundamental entities that you're working with and the fundamental operations that they understand.

showcase the parts of Google App Engine. In this chapter, we'll use the datastore to build a filesystem-like service.

We can think of this as the infrastructure for an additional feature in the chat service. Many chat systems—Google Talk, for example—provide users with the ability to share and exchange files. Our filesystem service could be used to implement a system like that in chat.

Before we start implementing a filesystem service in Google App Engine, we need to figure out just what we want. Since it's going to be a web-based filesystem, it's not going to behave exactly like a traditional operating system local-disk filesystem.

What should a web-based filesystem look like? There's actually already a standard way of doing it, based on an extended version of HTTP called WebDAV. WebDAV is really too complicated to implement as an example, but we'll use it as a rough model. In WebDAV, every URL identifies a resource, which is basically a file. Every resource has content, which is a collection of bytes, and properties, which are maps from names to pieces of metadata. Since datastore uses property as its name for the fields of a model, we'll use attribute instead.

So for example, in a traditional local filesystem, you have metadata such as owner, creation time, and access privileges. In our WebDAV-like filesystem, all of those will be handled using attributes.

One thing that's a bit surprising at first is that in WebDAV, a directory is just a regular resource. The only difference between a directory and a non-directory resource is that the directory has an attribute that contains a list of its children resources by name.

So we'll do that. To start, we'll just write some standard Python code, and then we'll gradually turn it in to datastore. So we'll start with a rough sketch of code for a very basic pseudo-filesystem written in Python. This is not GAE code: it's just standard Python to get a sense of how we want things to behave. In my experience, this is often a very valuable first step in building a GAE application or service. When you're building cloud applications, you've got both the traditional issues of figuring out how you want your system to behave and the new issues of figuring out how to make it behave in the cloud-based environment. Doing a first sketch helps you work out the basic design issues before you start on the cloud programming.

Download filesystem/filesystem.py

```
from datetime import datetime
import string

class Resource(object):
    @staticmethod
    def MakeResource():
        return Resource(content=None, attributes={})

    def fs_put(self, content):
        self.content = content

    def fs_get(self):
        return self.content

    def fs_setAttribute(self, name, value):
        self.attributes[name] = value
```

```
    def fs_getAttribute(self, name):
        return self.attributes[name]

    def isDir(self):
        return self.getAttribute("children") is not None

    def addChild(self, name, resource):
        if self.getAttribute("children") is None:
            self.setAttribute("children", {})
        self.getAttribute("children")[name] = resource

class FileSystem(object):
    @staticmethod
    def MakeFilesystem():
        fs = FileSystem()
        fs.root = FileSystem.MakeFile("/", "root", "")
        return self

    @staticmethod
    def MakeFile(name, owner, content):
        file = Resource()
        file.put(content)
        file.setAttribute("owner", owner)
        file.setAttribute("time", datetime.now())
        return file

    def getRoot(self):
        return self.root

    def getResourceFromChild(self, child, nameElements):
        """A recursive helper for retrieving a file by path. child is the name
        of a directory which is transitively contained in the target of this call.
        nameElements is a list of path components for the pathname that come after
        the child. Each recursive call resolves one element of the path, then calls
        itself on the remainder. When nameElements is empty we've found the file."""
        if nameElements is []:
            return child
        childsChildren = child.getAttribute("children")
        if childsChildren is None:
            return None
        else:
            nextChild = childsChildren[nameElements[0]]
            if nextChild is None:
                return None
            else:
                return self.getResourceFromChild(nextChild, nameElements[1:])

    def getResourceAtPath(self, path):`
        pathElements = string.split(path, "/")
        self.getResourceFromChild(self.getRoot(), pathElements)
```

This is pretty straightforward, so I'm not going to explain it in depth. A filesystem is an object that has a root directory in it. A directory is just a resource, which has an attribute named children, which is a map from name to resource. The filesystem has a method to resolve a complex path to get to a resource.

13.2 Modeling the Filesystem: A First Cut

Looking at our non-datastore version of a filesystem, let's think about how we'd translate this to datastore persistence. The filesystem is really easy: it's a persistent object with one property, the root resource. It's eventually going to be the object associated with a servlet: all calls to access anything in the filesystem will get routed through a single servlet associated with the filesystem object. But for now, we won't worry about that piece of things; we'll just create a model for the filesystem. We won't implement methods like getResourceFromChild yet. Before we can do that, we need to understand how our filesystem is going to be represented.

Download filesystem/persistent_filesystem.py

```
class FileSystem(object):
  def __init__(self):
    self.root = MakeFile("/", "root", "")
    return self
  def getRoot(self):
    return self.root

  def getResourceFromChild(self, child, nameElements):
    if nameElements is []:
      return child
    childsChildren = child.getProperty("children")
    if childsChildren is None:
      return None
    else:
      nextChild = childsChildren[nameElements[0]]
      if nextChild is None:
        return None
      else:
        return getResourceFromChild(nextChild,
                                    nameElements[1:])

  def getResourceAtPath(self, path):
    pathElements = string.split(path, "/")
```

The resource is more interesting. A resource has two properties: its content and its attributes. The content is, obviously, a blob—a great

big bundle of bytes. We don't know, or care, whether the content has any more structure than that.

But representing a resource's attributes is a problem. The attributes are a map from names to values, and the values could be anything! The Python persistence we've seen so far needs to be able to specify a type for persistent properties, but given the way we handled attributes in raw Python, there is no single type of value used by all attributes! We can't do it that way in datastore.

There are two ways that we can get around this. We either need to find a way of describing an attribute as a storable object that we can specify to datastore, or we need to find some way of storing things more flexibly. For now, we'll try the former: we'll say that all attribute values must be strings. If users want to use something more complicated as an attribute value, that's fine, but when they go to store it in a resource object, they'll need to translate it to a string format. That's not too burdensome, because to be used in messages, they need to have some way of translating the objects into a wire format like XML or JSON anyway, and they can just use that representation.

Of course, this doesn't completely solve our problem. With the datastore models we've seen so far, we can only use atomic property values—that is, we can't store lists or maps!

Fortunately, that's not really a limit of the datastore; that's a limit of the set of features that we've looked at so far. Datastore doesn't support maps, but it does support lists, albeit with a bit of work on our part for lists of interesting values. And as we'll see later, with the help of queries, that's good enough.

But for now, we can't use a map, so we'll use a list to build our own map. Datastore models support lists. The lists are sort of limited, but they do the job. A list needs to consist of values that all have the same type. And the values need to by either Python primitive objects or else datastore keys.

In the datastore, every stored object has a unique key. Given that key, you can retrieve and update the object. So for attributes, we'll use a list of key/value pairs. The key and value will both be strings. So to retrieve an attribute for a resource, we'll search the list of attributes for the resource and return the value of the attribute if we find it.

The first thing we need to do is create an attribute type:

Download filesystem/first-persistent.py

```
class ResourceAttribute(db.Model):
    name = db.StringProperty(required=True)
    value = db.TextProperty(required=True)
```

An attribute is an object that has two fields:

name A string property. Strings can be no longer than 500 characters, but we can write queries that use the value of a string, and we can sort a query result by a string property. We might want to be able to do a query for something like "all objects which have a children property" in order to retrieve all directories, so we need to use string.

value For the value, we'll use a text property. Texts can be as long as we want, but the ways that we can use them are more limited than strings. Here we're caught in a trade-off: on one hand, we might want to be able to do something like write a query that gets all resources created by a particular user; to be able to do that, we'd need to be able to use the value of this property in a query. But on the other hand, we don't know what kinds of data users might want to store in a property, and it's not hard to imagine cases where they'd want their properties to be longer than 500 characters. So, at least for now, we'll accept the trade-off that lets us use larger property values.

With that, we can create the first version of our model for our filesystem resources. The basic structure of our model, without the methods that implement its behaviors, is shown below:

Download filesystem/first-persistent.py

```
class PersistentResource(db.Model):
    content = db.BlobProperty(default = "")
❶   attributes = db.ListProperty(item_type=db.Key)

    @staticmethod
❷   def MakeResource(creator):
        resource = PersistentResource()
        resource.content = ""
        attribute = ResourceAttribute(name="creator", value=creator)
❸       attribute.put()
❹       resource.attributes.append(attribute.key())
        resource.put()
```

❶ Our list property for the resource attributes is pretty straightforward. We've defined an attribute type that the datastore knows

how to persist, so we can create a list of keys for those objects. So we just define the list property as being a list of datastore keys. It's that simple.

❷ It's a really bad idea to provide a standard Python __init__ method for a datastore persistent type: the datastore implementation provides a default initializer, and the correct behavior of the datastore internals rely on the fact that you don't change that. So we construct things using static methods. In the static MakeResource method, we create an empty instance of a resource and then populate a couple of fields. When we create an instance of a resource model, we'll initialize one attribute. We don't really need to do this: clients aren't going to call this code directly. They can only call it through the HTTP interface that we'll build later. But to illustrate how it works, we'll initialize the creator property here.

❸ We create the attribute in a normal way—using the default constructor provided by the datastore. Then we tell the datastore to store the attribute object. We need to do that for two reasons. First, the attribute is *not* contained in the resource object. So when we store the resource, it won't store the attribute object, it will just store a key-reference to it. So the attribute needs to get stored. We need to do it before we store the resource, because to put the attribute into the attribute list of the resource, we need its key, and the key isn't initialized until the object is stored. So we create the attribute and put it to the datastore first.

❹ With the attribute stored, we can now access its key. To set the attribute for the resource, we just add the attribute object's key to the attributes list. Then we store the resource, and we're done.

Now, with our models in place, we need to look at how to implement their behaviors. The basic get and put of content are easy and straightforward:

Download filesystem/first-persistent.py

```
def GetContent(self):
    return self.content

def PutContent(self, content):
    self.content = content
    self.put()
```

That should be clear without any explanation! Now we can move on to the attributes. Let's look at retrieving the value of an attribute first:

Download filesystem/first-persistent.py

```
def GetAttribute(self, name):
    for attr_key in self.attributes:
        attr = ResourceAttribute.get(attr_key)
        if attr.name == name:
            return attr.value
    return None
```

❶ The attributes field of our resource is a standard Python list. The elements of that list are the keys for our attribute values. To find an attribute with a particular name, we'll need to iterate over all of the attributes. We can just use a standard Python loop to do that.

❷ Now, inside of that loop, we want to look at the actual attributes. So the first thing we need to do is retrieve them. When you have a key, you can retrieve it by calling Classname.get(key). You can see here why, even using keys, you can't really have a list whose elements aren't all the same type. When you retrieve a stored object by key, you need to know its type in order to call key. So here, since we know that the keys are all for ResourceAttribute objects, we can retrieve them by calling ResourceAttribute.get(attr_key).

You may be worried about the performance of doing a sequence of get operations, one for each attribute. It's really not bad. First, the number of attributes per resource is going to be small: in real filesystems that use this sort of structure, the maximum number of attributes for resources is typically around two dozen or so, with most having many fewer. This means the number of retrievals is pretty small. Also, retrievals by key are really fast—dramatically faster than retrievals by query. The exact relation varies depending on the complexity of the query, but you can generally do a lot of retrievals by key for less than the cost of a single query.

The implementation of SetAttribute is very similar:

Download filesystem/first-persistent.py

```
def SetAttribute(self, name, value):
    for attr_key in self.attributes:
        attr = ResourceAttributeModel.get(attr_key)
        if attr.name == name:
            attr.value = value
            attr.put()
            return
    newAttr = ResourceAttribute(name=name, value=value)
    newAttr.put()
    self.attributes.append(newAttr.key())
    self.put()
```

❶ Just like in getAttribute, we need to search through the list of attributes to find the one we want to change. So we start with a loop exactly like what we did in get: we iterate over the attribute keys, and for each one, we retrieve the attribute object. Then we check its key.

❷ If we found an attribute whose key matches the one we want to set, we just update the attribute object and re-store it. Since we're updating the value in place, we don't need to worry about resaving the key: the attribute key is already stored in the resource object, and we haven't changed that. Once we're done updating it, we're done, so we can return.

❸ If the property doesn't already exist in the resource, then we'll get out of the loop without having returned. In that case, we need to create the attribute and save it to the datastore. Then we can put the attribute's key into the resource's attribute list and save the updated resource. We need to save the resource in this case because we modified it by adding the new attribute.

So that's the basic core functionality of the filesystem, implemented in a way that can be used with the datastore. We need to be able to do a couple of other things. For example, we'd like to be able to ask if a given resource is a directory. In terms of our data model, a directory is a resource which contains a child attribute. That's easy to check:

Download filesystem/first-persistent.py

```
def IsDir(self):
    return self.GetAttribute("children") is not None
```

We also need to be able to do things like add a new file to a directory. How can we do that? Naively, we'd get the children attribute from the resource and then add a file to it:

Download filesystem/first-persistent.py

```
def NaiveAddChildToDirectory(self, name, resource):
    children = self.getAttribute("children")
    if children is not None:
        children.value.append(resource) # WRONG!
```

The problem is that doesn't work. As we saw earlier, things need to have specific types. The type of an attribute value is text, so we can't just put the resource object there. So how do we represent the children list? If children is really just another attribute, then it shouldn't be treated

any differently than any other attribute. But that's going to be really awkward. So what can we do?

Answering that question isn't trivial, and it brings us to one of the most important factors in designing a good datastore model: where do you put the dividing line between different objects? And how do you describe relationships between objects? In datastore, you do that with references.

Datastore Keys and References

When it comes to handling the value of the children attribute in the datastore, we've got two real choices: we can either find a way of encoding the structure of a directory as a string and then put that string into an attribute, or we can treat the children attribute as a special case. We'll look at some details of the trade-offs between those two later; but for now, we'll just go the easy route and treat children as a special case. Directory structure is a key detail in a filesystem, so while it may seem a bit hacky, it really is important enough to justify treating it specially.

But either way, we're coming up against another important factor. A file in a directory is an independent object. If you think in terms of a classical object model, a file isn't a *part-of* the directory that contains it; it's a standalone object which is *referenced-by* its parent directory. This distinction is particularly important in a system like datastore.

Suppose we made files a part of the directory that encloses them. Then whenever we retrieved the root directory of the filesystem from the datastore, we would need to load the entire filesystem out of the datastore. The root directory ultimately contains every file and directory in the system; if it just contained parts of them, they'll be stored and retrieved with it. In addition, we won't even be able to retrieve an individual file: it will only exist as an attribute within the root directory.

That's definitely not what we want.

We want our filesystem to be a collection of independent objects. In terms of a classical in-memory object model, what we want is a pointer: we don't want the directory to contain its children; we want it to point to its children.

In datastore, the equivalent of a pointer in a language like C++ is called a *reference*. A reference isn't like a C++ reference, which is really just another name for a pointer. An object-store reference is an identifier that uniquely identifies some other object in the datastore. It's almost

like a pointer to the target object in the datastore, except that instead of pointing to its location, it provides an identifier that allows the datastore to retrieve the specific object.

As we mentioned before, every object that's stored by datastore contains a unique identifier called a *key*. A reference to another object is a property whose value is the key for another object. So far, we've worked with keys in a very primitive way. But App Engine gives us another convenient mechanism for working with them. A datastore reference object is a wrapper for a key that behaves as if it's the object identified by the key itself. When you try to access a field or method of a reference object, it automatically does a retrieval itself and then forwards the call to a retrieved object. In terms of our filesystem, that means that we can return a list of all of the children objects of a particular directory without actually retrieving all of them.

Let's look at how we'd do that. We'll create a new version of our resource model type, which has a dedicated property for children. Like we did for the attributes model, the children property will be a list of objects, each of which is a name/value pair:

Download filesystem/persistent_filesystem.py

```
❶ class DirectoryEntry(db.Model):
      name = db.StringProperty()
❷     resource = db.ReferenceProperty(PersistentResourceModel)
```

❶ We need to define the model class for a directory entry. It's just a typical db.Model subclass.

❷ This declares a property that's a reference to another resource. To make a reference property, create a field with a db.Reference value and provide the constructor with a parameter that is a subclass of db.Model. When you set the value field of this class to a particular resource, what will really happen internally is that the key for that resource will be stored.

Now, we can write the resource definition. What we'll do is make children be a special case—we'll put a hard-wired children attribute into the resource, which is a list of DirectoryEntry keys.

Download filesystem/persistent_filesystem.py

```
class PersistentResource(db.Model):
    content = db.BlobProperty(default = "")
    attributes = db.ListProperty(db.Key)
❶   children = db.ListProperty(db.Key)
```

❶ In the new resource model, we include a property for the children, which is a list of keys for DirectoryEntry objects. We *don't* use references to the entries; we can't do that in a list property. And besides, conceptually, we want the entries to be contained in the resource object, which means that we will only store or retrieve them from inside of the methods of the resource class. The point of references is that they make it transparent when the actual retrieval happens. Then you can pass the reference around as if it were the real object without worrying about how to retrieve it when its value is actually needed. But here, there's only one place that these values will ever be retrieved, so the overhead of references just makes no sense.

Now we've got an interesting setup. Our resource type contains a list of directory entries, which are accessed by key, and the directory entries are references. How do we use this for implementing directories? Well, first let's look at how we need to modify the GetAttribute and SetAttribute methods to work with directory entries. Then we'll look at code that uses these filesystem objects to see how we actually use the references.

Download filesystem/persistent_filesystem.py

```
def GetAttribute(self, name):
  if name == "children":                                          ❶
    return [ DirectoryEntry.get(key) for key in self.children ]
  else:
    for attr_key in self.attributes:
      attr = ResourceAttribute.get(attr_key)
      if attr.name == name:
        return attr.value
    return None

def SetAttribute(self, name, value):
  if name == "children":
    self.children = [ de.key() for de in value ]                  ❷
    self.put()
  else:
    for attr_key in self.attributes:
      attr = ResourceAttributeModel.get(attr_key)
      if attr.name == name:
        attr.value = value
        attr.put()
        return
      newAttr = ResourceAttribute(name=name, value=value)
      newAttr.put()
      self.attributes.append(newAttr.key())
      self.put()
def IsDir(self):
  return (self.children is not [])
```

❶ In GetAttribute, we first check to see if the attribute name is children. This is sort of ugly, but it's unavoidable here due to the limitations that we're dealing with. The children attribute is a special case, so we need to check for it explicitly. If it's the one being retrieved, then we use special code that retrieves the children of the resource; otherwise, we fall through to exactly the same code that we used before. If it is retrieving the children attribute, then we use a list comprehension to go through the list of DirectoryEntry keys, retrieve the actual DirectoryEntry objects, and then return that list. So to the client of our filesystem, when they call file.GetAttribute("children"), the result is a list of directory entries. But there's a bit of cleverness hidden in there: as we'll see later, to the clients of this code, the directory entries behave as if they really contain the child resources, but in fact, they just contain references.

❷ In SetAttribute, we do the mirror image of what we did in GetAttribute. We start exactly the same way, by checking what attribute we're setting. If it's children, then we expect the value to be a list of DirectoryEntry objects, so we use a list comprehension to do exactly the opposite of get: we replace the directory entries with keys, getting a list of keys, and then we store that into the special children attribute of the resource. Since that modifies the resource, we need to save it with a put call.

Once the reference property is set up, how do we use it? Datastore does all of the work. If you just get the value of a property of a datastore instance that is a reference, the datastore will automatically retrieve it for you completely transparently. So, for example, we could iterate through the children of a directory, printing out whether each is a directory with the following function:

Download filesystem/persistent_filesystem.py

```
def RenderChildren(dir):
  children = dir.GetAttribute(children)
  for c in children:
    # c is a DirectoryEntry
❶   if c.resource.IsDir():
      print("Child %s is a directory" % c.name)
    else:
   print("Child %s is not a directory" % c.name)
```

In the function, we iterate over the DirectoryEntry values in the children property of directory. For each, we access the resource that it references

by calling the IsDir method on the resource field of the DirectoryEntry. This field is not a resource; it's a reference. But whenever we try to access any field or method of the reference—as when we call its IsDir method here—the datastore automatically retrieves the referenced object.

So far, this seems remarkably simple—almost too simple Persistence can be extremely complicated to get right, but what we've seen so far does an admirable job of hiding that complexity. However, the underlying complexity is still there.

This affects you in a couple of ways. Most immediately, the fact that datastore automatically retrieves references for us when we use them is great, but under the covers, it's still a round-trip call to the datastore. Retrieving things from a persistent storage system is not free. It's not terribly expensive and is far less expensive than doing a query. But it's not free, and the costs of doing it can accumulate and become quite significant. Hiding the retrieval from ourselves is very convenient, but it also makes it easy to accidentally do things that will make our code extremely slow.

For example, in the directory printer up above, what happens when we iterate over the directory that way? It retrieves the resource objects for every child of the directory. And doing that means retrieving the keys of every DirectoryEntry for every child of every member of the directory. And if we use those, we end up retrieving references to other resource objects. It's really easy to write a traversal routine that walks through a directory hierarchy but which ends up doing a huge number of retrievals, each independently as part of a reference access. In a case like that, we'd be much better off with just a single query that retrieved all of the objects at once. Using references without care means you can potentially end up with extremely inefficient code in ways that just aren't very obvious if you're not used to cloud-style distributed programming.

Here's an example to give you a sense of the potential impact. Early in my career, I did some work on a persistence system for C++. In that system, we did something a lot like this automatic reference retrieval. At one point, I was working on some example code for a customer and found that its performance made absolutely no sense. I was iterating over a loop, but the time cost was huge. Instead of being roughly proportional to the size of a list, it was proportional to the *square* of the size of the list. The problem turned out to be almost exactly what I described above: each time someone accessed a member of the list, it

automatically retrieved its list of children and constructed proxies (our form of references) for them. So accessing a list element actually ended up creating a list a proxies. What looked like an innocuous line of code was actually expensive because of the clever stuff that the system was doing behind my back.

In datastore, you need to be aware of what's really going on in order to understand the costs. When you access a reference parameter, what you're really doing is making a call to the datastore: db.get(key). And that can cause a very expensive retrieval operation.

Sometimes that's a real problem, but you can avoid it by working with keys explicitly. When you're working with keys, as we do with our DirectoryEntry objects, you make the retrievals explicit. In your code, it becomes very clear how many retrievals you're doing because they're not hidden from you anymore. There's a subtle balance in this kind of programming—to choose between using the simplicity of something like a reference and the explicit nature of something like a key. In general, if you're going to have a lot of objects, I'd recommend working with explicit keys. It's more effort, but it keeps you honest.

The next major question for building our filesystem is how should we handle the content? In our initial model, we stored the content in a text property. That's good, but it's not great. Text objects can get pretty big, but they're still basically just strings. That's not quite right: the content of a file in a filesystem can be arbitrarily large, and the content can be any sequence of bytes.

But there's also another problem. Every time we retrieve a file object, we retrieve the *entire* file object. What if the content is 20 megabytes, and all we want to do is check if the file is a directory, as we did in the render function we wrote before? That's incredibly inefficient. And remember: in the cloud, you're paying for things. The time it takes to load the file content you don't use isn't free. What we really want to do is create another object-type for content, so we can use a reference.

Our final object model (for this section) is the following:

Download filesystem/filesystem_blob_model.py

```
❶ class ContentModel(db.Model):
      data = db.BlobProperty()

  class DirectoryEntry(db.Model):
      name = db.StringProperty()
      resource = db.ReferenceProperty(PersistentResourceModel)
```

```
class Resource(db.Model):
❷    content = db.Reference(ContentModel)
    attributes = db.ListProperty(db.Key)
    children = Db.ListProperty(DirectoryEntry)
```

This is really close to what we had before. The main differences are as follows:

❶ We create a new class for file contents. Its only attribute is a Blob. A blob is pretty much like a text, except instead of it being a series of bytes interpreted as a sequence of characters, it's just a series of bytes that are completely uninterpreted. For a file, that's exactly what we want: the program that reads the file contents can decide how to interpret it.

❷ In the Resource class, we use a reference to the content object. With that, when we retrieve a resource object, we just get a reference to its content. When we try to access the content, then it will get retrieved.

Implementing the Rest of the Filesystem

We've taken care of most of the datastore parts of our filesystem, but we still need some glue code to put it all together. What we've done so far is implement files and directories, so we need to combine them into a full filesystem.

A filesystem is, ultimately, a collection of files and directories, with one particular special directory called the *root*. All references to files inside of the filesystem will be worked out relative to the root. The other thing that the filesystem needs to do is implied by the last sentence: it need to be able to resolve a full pathname to the specific file that it references or generate an error if it can't.

A bare minimum implementation of the filesystem, then, is the code below. The only complicated part of that is the implementation of GetResourceFromChildByList, which recursively traverses the directory hierarchy to retrieve a particular file. If you have trouble following the code, don't worry; it's not really important to what follows. All that it's doing is starting with a path like /a/b/c.txt and finding the correct resource by first looking in the root directory for a resource named a, then looking in the a resource for a child resource named b, then looking in the b resource for a child resource named c.txt, and then returning c.txt.

Download filesystem/filesystem_servlet.py

```
class Filesystem(db.Model):
    root = db.Reference(Resource)

    def GetRoot(self):
        return self.root

    def GetResource(self, path):
        path_elements = path.split("/")
        return self.getResourceFromChildByList(self.root, path_elements)

    def GetResourceFromChildByList(self, resource, path_elements):
        if path_elements is []:
            return resource
        else:
            for direntry in resource.children:
                if direntry.name == path_elements[0]:
                    return getResourceFromChildByList(direntry.resource,
                                                      path_elements[1:])
            return None
```

Implementing File Retrieval Using GET

We've worked out a basic filesystem model; the next thing to do is implement it. The datastore parts of this are pretty easy—at least at first. To do a basic filesystem, it's pretty much just a series of full-object GETs and PUTs. There's a bit of work to get the URLs mapped, but it's nothing particularly different from what we've seen before. But as you'll see, once we have the basics, we can move on to some more interesting things, like searching the filesystem based on property values.

Let's we'll start with the basics. What do we want users to be able to do with our filesystem?

1. Create a resource. This involves creating the resource object, setting its properties, storing its content, and modifying its parent resource to provide a directory entry to the new resource.
2. Retrieve a resource's contents.
3. Set and retrieve resource properties.
4. Update a resource's content.

We must map these actions onto the basic HTTP operations: GET, PUT, and POST. Retrieving a resource content or its properties is clearly a GET operation. Creating a new resource or updating its content is PUT.

In fact, for now, we really don't need POST at all: we'll use GET for all retrieves, and we'll use PUT for all stores.

Now we need to think about how to structure our URLs. If we ignore attributes, it's easy—we have simple paths in a URL, and the standard URL syntax was designed exactly for that. For attributes, it's a bit trickier. We could use CGI parameters to modify the HTTP call, or we could create a way of encoding attributes into the URL. The classic REST style calls for the latter approach. An attribute is a kind of data accessible by the service; each piece of uniquely identifiable data should have its own URL.

Attributes are bits of metadata for a resource, so their URLs should be based on the URL of the file they're associated with. In order to prevent confusion between an attribute of a file and a member of a directory, we'll prefix attributes with "~", which we won't allow as a path character in the name of a resource. Given a resource with URL R, its attribute a1 will be accessed by a URL R/~a1.

We can make a first version of our filesystem service with a GET handler:

Download filesystem/filesystem_servlet.py

```
class FilesystemResourceHandler(webapp.RequestHandler):
    def GetFilesystem(self):
        query = Filesystem.gql("")
        return query.get();

    def get(self):
❶       filesystem = self.GetFilesystem()
❷       root = filesystem.root
❸       url = self.request.path
        urlElements = url.split("/")
        # And then check if it's a content request or an attribute request.
        # if the first char of the last name element is "~", then it's
        # an attribute.
        resourcePath = None
        attr = None
❹       if urlElements[-1].startswith("~"):
            attr = urlElements[-1]
            resourcePath = urlElements[:-1]
        else:
            resourcePath = urlElements
❺       resource = filesystem.getResourceFromChildByList(root, resourcePath)
❻       if resource is None:
            self.response.error(404)
            return
❼       if attr is not None:
            result = resource.getAttribute(name)
```

```
            if result is None:
                self.response.error(404)
            self.response.out.write(str(result))
            return
        else:
❽           self.response.out.write(resource.content.data)
```

❶ First, typically for a cloud application, we need to call a retrieval function to fetch the filesystem root object. We know how to write that: it's just a simple datastore retrieval.

❷ With the filesystem, we can get the root resource—that is, the root directory of our filesystem.

❸ Then we go to the request, and from it we retrieve the path for the resource that we want to retrieve.

❹ We parse the path to see if it's a request for an attribute or for the resource content.

❺ We retrieve the resource object using the method we wrote earlier in the chapter.

❻ If the attempt to retrieve the resource fails, it returns null. So we generate a 404 response (the HTTP response for "resource not found").

❼ If the URL included an attribute name, we retrieve the attribute and return it in the content of the result message. If we don't specify the result code, it will default to the success code, and the content-type will default to UTF-8 text. Since those are both what we want, all we need to do is write the attribute value to the response stream.

❽ Otherwise, we use the datastore reference auto-retrieve to get the resource content and write it to the stream. Again, the defaults will work perfectly: when we write a blob to the response output stream, if the blob is just text characters, it will default to UTF-8; if it's got any nontext bytes, it will default to a byte stream.

Implementing File Storage Using PUT

The interesting thing about implementing file storage is that when you PUT the files, you might need to also update the parent of the file in order to create a directory entry for the new resource.

The code is pretty similar to GET. We start off by parsing the path. If the PUT is a content-put, we need to retrieve either the resource or,

if it doesn't exist yet, its parent. If it's a property-put, the resource has to exist, so we need to retrieve the resource itself. Then we do the appropriate update.

Download filesystem/filesystem_servlet.py

```
def put(self):
    filesystem = GetFilesystem()
    root = filesystem.root
    url = self.request.path
    urlElements = url.split("/")
    resourcePath = None
    attr = None
    if urlElements[-1].startswith("~"):
        attr = urlElements[-1]
        resourcePath = urlElements[:-1]
    else:
        resourcePath = urlElements
    resource = filesystem.getResourceFromChildByList(root, resourcePath)
❶   if resource is None:
        parent = filesystem.getResourceFromChildBylist(root,
                                                       resourcePath[0:-1])
        name = resourcePath[-1]
        if parent is None:
            self.response.set_status(404,
                                     "Parent dir of new resource not found")
        else:
            resource = Resource(content = self.request.body, attributes=[],
❷                               children=[])
            resource.put()
❸           dirEntry = DirectoryEntry(name=name,resource=resource)
            parent.children.append(dirEntry)
            parent.put()
            self.response.set_status(100, "Resource created")
            return
    else:
❹       resource.content=self.request.body
        resource.put()
        self.response.set_status(100, "Resource updated")
```

❶ Everything is just like GET up to the point where we try to retrieve the resource. In GET, if we couldn't find the requested resource, we returned an error. In PUT, we just try to create the resource. In order to create it, we need to try to get the parent resource. If we can't find that, we return an error. But if we can find the parent, we go ahead and create the new resource as its child.

❷ We can create a new resource by instantiating the model class. Each of the properties of the model is a named parameter to the constructor. Once it's created, we save it to datastore using put().

❸ Once the new resource is created and stored, we need to create a directory entry for it in its parent object, add it to the children of the parent resource, and put the updated parent.

❹ If the resource already exists, all we do is update its content, and then put it.

13.3 Property Types Reference

Google App Engine provides support for quite a few types of properties for datastore models. In this section, we'll look through a list of them. It's a bit boring, but it's useful to have the full list and a place to refer to for information. The property types supported by datastore fall, roughly, into two different categories. There are primitive types, the basic types essential to storing stuff in datastore, and there are utility types, which are more specialized things (like, for example, email addresses) that come up frequently enough to justify their own types in order to provide a standard method of formatting and validation.

Primitive Property Types

BlobProperty
: A blob is a "binary large object"—effectively, a chunk of bytes of arbitrary size. You can't use anything about blobs in queries; they're not comparable or sortable. The value of a blob property is an instance of db.Blob, which is a subclass of str. So you can use them from your code as if they were strings of bytes, because that's exactly what they are!

BooleanProperty
: A BooleanProperty is a simple boolean value, either True or False.

ByteStringProperty
: Basically the same thing as a blob but indexed and length-limited like a string.

DateProperty/DateTimeProperty/TimeProperty
: A timestamp object. In Python, this is implemented as an instance of datetime.datetime. There are actually three versions of this that are roughly the same: they're all datetime objects. With DateProperty, the time fields are left blank; for TimeProperty, the date fields are blank. They're sorted in chronological order in query results.

FloatProperty
: A floating point number.

IntegerProperty
: A 64-bit integer value.

Key
: A property that stores a key that uniquely identifies some other datastore object. Keys are supposed to be completely opaque; they are not sortable or comparable for anything except equality. If you absolutely must, there are methods for decomposing keys to extract information, but most of the time, if you need to do that, it's a sign there's a serious problem with your persistence design.

ListProperty
: A list of some value type that is supported by datastore. This takes the type of list element as a parameter. Unlike normal Python code, you can't have a list of mixed types; a datastore list property can only store values of a single type. There's also StringListProperty, which is just shorthand for ListProperty(item_type=basestring).

ReferenceProperty
: A reference to an object of some particular type. The type of the referenced object is a parameter to the constructor. When you dereference a ReferenceProperty, the datastore will automatically and transparently do a get to retrieve the object.

SelfReferenceProperty
: A convenient shorthand for a reference property to an object of the same type as the object containing the reference. If one of our resources had a field that was another resource, we could define it using a SelfReferenceProperty instead of a ReferenceProperty(Resource).

StringProperty
: A string value.

TextProperty
: A TextProperty is a cross between a string and a blob. It's a string type—its values are interpreted as unicode strings. But like a blob, it's got an arbitrary size, and it's value can't be used in queries.

Complex Property Types

CategoryProperty

: A string property. This is used in the context of building things like Atom feeds, where it represents an atom category.

EmailProperty

: A subclass of string that represents email addresses. Unlike most of the specialized types, there is no validation of the syntax of an email address provided by this type. It's only useful as a hint to people reading your code that the value of a property is intended to be interpreted as an email address.

GeoPtProperty

: A geographical location. The value is an XML element containing the location in terms of the standard GEORSS representation, as defined at http://georss.org.

IMProperty

: A representation of an instant messaging handle. This can be used by an Google App Engine service for communicating via instant messaging services. It's constructed from a URL identifying the IM protocol and a username. For example, my Google Talk IM would be db.IM("http://talk.google.com", "markcc@gmail.com").

LinkProperty

: A LinkProperty is a string containing a valid URL. This is designed for use in Atom feeds, but it's generally useful for any application that needs to store links. It provides full validation of the URL.

PhoneNumberProperty

: A validated string containing a telephone number. This is validated against a variety of international standard phone number formats.

PostalAddressProperty

: A string value containing a postal address.

RatingProperty

: A value for representing a user rating or ranking of something. It's an integer ranging from 0 to 100.

13.4 Wrapping Up Property Types

In this chapter, we looked deeper into the datastore. We learned about the types of properties that we can have in the datastore. We looked at the difference between containment and reference in stored objects, and we learned about keys and references, which are the main mechanisms for managing relationships between stored objects in the datastore. We used this to implement the basic structure of a filesystem service in the datastore.

There's still more to learn about datastore. To make an efficient, scalable datastore model, we need to know how to define an index and how to select the right indices to create to support the queries that we expect to perform. There's also more we can do with models. The models that we've seen so far have a fixed set of properties. Datastore does allow us to define models that can add new properties as needed, but doing that comes with its own set of restrictions.

In the next chapter, we'll look at some more advanced features of the datastore. We'll focus mainly on query issues, such as indices, keys, cursors, and policy, and how queries work. We'll also look at how we can create more flexible and expandable models. And we'll do some more work on our filesystem implementation by adding a way of doing queries for resources with particular properties.

Chapter 14

Advanced Datastore: Queries and Indices

In the last chapter, we looked at the types of values that can be used as properties in the Google App Engine datastore. Although we focused on Python, the set of property types—that is, the set of values types that can be used as fields of storable objects—is the same for both Java and Python.

In this chapter, we'll build on that by learning how datastore queries really work. Behind the scenes, the process of making queries work quickly and scalably is done using indices. The App Engine SDK can automatically generate indices for your data based on your code, but sometimes defining the indices yourself can give you better performance and fewer indices. Again, like the last chapter, we'll be working mostly in terms of Python, but queries and indices work exactly the same in both Python and Java, and you'll need to create a custom index for the same kinds of query in exactly the same way, regardless of which language you use.

In this chapter, we'll look at how indices are generated by datastore and how we can define our own indices by declaring them in app.yaml. We'll use that to define an index on file attributes that allows us to search our filesystem, and we'll put together the complete app.yaml file that we need to deploy our filesystem application.

14.1 Indices and Queries in Datastore

Before we get into detail about how App Engine indices work, it's worth taking a moment to understand the technology under the covers. Datastore indices, and datastore queries that rely on those indices, are quite different from what most people are used to. They have a lot of restrictions, which can seem downright arbitrary unless you understand what's really going on.

Under the Covers in Datastore

Ultimately, all of the data that we put into datastore is actually stored using a Google storage system called Bigtable. You don't get to program Bigtable directly using datastore, but Bigtable defines the basic properties of datastore storage, queries, and performance.

Without going too far into detail, what Bigtable provides as a structure for storing data isn't quite what you probably think of when you hear "tables." Bigtable's storage model is a bit tricky. In some ways, it's providing tables, but they're *not* like relational database tables. The most common mistake people make when working with both Bigtable and datastore is to try to set up data as if they're working in a relational database. In an RDB, you do a lot of things like data normalization in order to improve performance and queryability. But if you do data normalization in datastore, you'll actually harm your performance and make it much harder to write your queries!

Data normalization is a trick relational database people love because it has an absolutely enormous benefit in relational systems. The idea is that you try to push things down so that each individual table is as flat as possible: complex structures are stored in terms of relationships between tables. For example, if you had an object, like our Resource object, which contains a collection of values like our Attribute values, then you'd separate them through an intermediate table. That is, you'd take the resource object, entirely remove the attributes from its storage model, and you would add a unique identifier to each attribute. Then you would store the Resource and Attribute values in separate tables and add a third table called a relation, which contains a list of pairs of resource identifiers and attribute identifiers. To retrieve the attributes of a resource, you would do a query, something like `select attr from attributes, rel where attr.id = rel.attr_id and rel.resid = ?`.

In datastore, you *really* don't want to do that.

Bigtable is more like a two-level hashtable or lookup table than it is like a relational database table.

The way that most people think of a table is a rectangular grid with labels on the rows and the columns. Every row has exactly the same columns. That's the model of a table used by a relational database.

Bigtable has rows and columns, but they're not like relational database rows and columns. In Bigtable, all data is stored in rows. Each row has just one key. The only way to access a row quickly is by knowing its key. Each row, in turn, consists of a set of columns. But the columns are completely arbitrary. Different rows can have entirely different columns. And each row can have thousands of different columns!

That brings us back to what I mean about Bigtable being a two-level hashtable. Given a key for a row in a table, you can retrieve the row extremely quickly. On the top-level, then, it functions sort of like a hashtable or dictionary that lets you quickly look up the row associated with a particular key. Once you've got a row, it's like you have a hashtable for the columns: you can very quickly retrieve a particular column from the hashtable once you've got its row and its column identifier.

In addition to the lookup stuff, transactions are row-focused—each transaction protects an operation on a single row. Each time you do a datastore update operation that involves multiple Bigtable rows, you're dramatically increasing the complexity of the transaction.

Of course, you don't really know exactly how your model is going to be mapped onto a bigtable. In fact, it's entirely possible that over time, the way in which your data is mapped onto Bigtable could change. But the broad outlines of the mapping are pretty straightforward.

In datastore, each model object is a Bigtable row. Even if you have list fields—like the attribute fields of our resource objects—keeping the fields in the object will keep the values in the same Bigtable row, which will have significant performance benefits. So for performance in datastore, you actually want to do the opposite of normalization. Instead of flattening your objects out as much as possible, what you really want to do is bundle them together!

Similarly, the way that queries work is related to Bigtable. Bigtable supports a method of searching over a sequence of values based on a comparison. But it's restricted because of the way that Bigtable is

implemented. All of the query restrictions derive from the underlying properties of Bigtable. So, for example, you can only use relational comparisons on one field in any query. That's because the relational comparisons can only be done efficiently through indices, and you can only use one index in a Bigtable query.

Automatically Generated Indices

As I said before, the App Engine SDK can automatically generate datastore indices automatically. Basically, the first time you execute a particular query using the local App Engine execution environment provided by dev_appserver.py, it checks your storage model to see if there's an index that's well suited to that query. If there is one, then it just uses it. If not, it creates one.

For the majority of applications, this is wonderful. You don't need to worry about indices; datastore will take care of it for you. You just define models based on what's natural for your application (keeping in mind, of course, that you're defining them for datastore, not for a relational database!), and datastore's infrastructure will do its best to make your queries run quickly.

But as usual, nothing automatic is ever perfect. App Engine and datastore will only try to automatically generate indices for certain cases where it's pretty certain that it knows the right index. If you want to work outside of that, you'll need to define your own index.

So what cases does datastore know how to do correctly? There are four common cases:

Equality Filters.
: Datastore has built-in equality indices for all fields. If you've got queries based on equality comparisons—that is, any query whose entire filter clause is WHERE x = y (or some series of equality comparisons joined by AND), you don't need to worry about making an index: datastore always has one.

No Filters, Sorted.
: If you're doing a query where there is no comparison, and you're just getting a set of values ordered by a single sort order (like, for example, a query to return all chat messages sorted by date), then App Engine can automatically generate an index.

Object Equality, Property Relational.
: For queries where all the comparisons between top-level persistent objects are equality tests and all comparisons between properties

are relational comparisons of the same property, then App Engine can automatically generate the filter.

Relational Filters.
: If you're doing query comparison where the only filter is a relational comparison, (i.e., less than or greater than), and all of the comparisons are of the same property, then App Engine can automatically generate an index.

This list it might seem pretty restrictive. But it really does cover most of the common cases that you'll encounter. For example, in our chat application, we used queries that compared the `chat` property of a message with a particular value to show the messages in a chat room. That was an equality comparison—so it was automatically indexed under case one.

In fact, every query that we've written so far falls under the default indices. The default indices are good enough that nearly every query that we'd want to write can be expressed in them. Coming up with a case to justify a custom index actually takes a bit of work!

Creating Custom Indices

As I said before, for the most part, you can get by with the restrictions of automatically generated queries. And when you can, you should. There's a reason for those restrictions. Most things that get around them introduce potential performance issues when you scale up to large quantities of data. For example, the query that we're going to work through requires App Engine to keep track of ordering of two different properties of the same model type. That adds a cost to every update of the model type in the datastore. So before you jump in to building custom indices, think carefully about whether you really need them. In many cases, it's better to use a simpler query that returns too much data, which you then need to filter in your servlet code, than to use a more complex query whose index will have a negative impact on every update. There's a subtle trade-off there: a custom index makes sense if it can save you a lot of communication. So if you use a particular query frequently and you can significantly reduce the quantity of data produced by the query, then it might be worth doing a custom index. But if it doesn't get used often or it doesn't make a large difference in the quantity of results, then you're probably better off keeping things simple and sticking with standard indices.

Just to explore a bit further, let's try and come up with a case where we really need a custom index.

What we need to do is come up with a reasonable query that doesn't fit any of the automatically indexed constraints. Basically, the places where you need a custom index are when you're going to use a compound query that tests equality on multiple fields and relations on a single field. In our filesystem, we don't have any objects where we can even write a query like that! In fact, in many App Engine programs, your data models will end up like this, where there's just no case where you need a custom index. But suppose that we had a slightly different model, like this one:

```
class ResourceAttribute(db.Model):
  name = db.StringProperty(required=True)
  type = db.StringProperty(required=True)
  value = db.TextProperty(required=True)
```

So we've added a new property to our ResourceAttribute model, which allows us to assign a type-name to a given resource. This type-name would be something that tells us how to interpret the value property. So, for example, we could imagine a system where some properties were part of a security system. Those properties would have the type security-level. So then we might want to find all of the security-level attributes named protection that specify a security level between 400 and 500.

In GQL, that would be a query like

```
select a from attributes
 where a.name = "protection"
   and a.type = "security-level"
   and a.value >= "400" and a.value < 500
```

That query won't have an automatically generated index because it's got equality tests on two different properties and it requires relational comparisons on a third property. The App Engine SDK won't generate that index automatically: it's an expensive index to create, so it leaves it up to you to decide whether it's really needed or not.

To create the custom index, we need to specify the index inside of our applications app.yaml file. That's the master configuration file for a Python application. It's written in an (ugly) markup language called YAML. Index definitions go under the indexes section of the file. Each index definition starts with the kind of the data being indexed. The kind is the name of the model class.

Then you need to specify a list of the properties to be indexed and whether they should be sorted in ascending order or descending order. For our query, here's an index definition:

```
indexes:
- kind: ResourceAttribute
  properties:
  - name: name
  - name: type
  - name: value
    direction: asc
```

What this says is that we want to index ResourceAttribute in order to be able to do comparisons based on equality for the type and name properties and for inequality comparisons on the value, with the index sorting the value property in ascending order. And that's it: once the index is declared in the app.yaml file and the application is deployed, it will generate that index for any data already in the datastore, and it will update the index as you add and/or update data.

You can also see what indices App Engine has created for a running application. If you go into the App Engine control panel for the application, there's an entry to check the datastore indices. For example, for the Java chat room application, there's one index on PChatMessage, which indexes in ascending order by date and by chat.

Indices in Java

Datastore indices work pretty much the same way in Java as they do in Python. The same rules and restrictions apply, and they are very similar. There is, however, one difference. In Python, when we needed to create a custom index, we did it by putting an entry into the app.yaml that defined the Python application. But Java applications running in App Engine don't have an app.yaml file. In Java, you need to use a different file: in the case of Java, it's a file named datastore-indices.xml. And as you can see, it's an XML file.

I've heard a lot of people complain about having to use something complex like XML instead of the lightweight YAML from Python. Personally, I prefer the XML file. YAML file syntax is rather eccentric. I always find it hard to get the syntax right. It can be difficult to know when to add another "-", and when to do a newline without a dash. I never manage to get it right. XML may be ugly, and it might take twice as many lines as necessary, but XML is clear, and it's easy to check exactly how things should look.

The index we created in Python can be created for Java using the following XML:

Download filesystem/datastore-indexes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes
  autoGenerate="true">
    <datastore-index kind="ResourceAttribute" ancestor="false">
        <property name="name" direction="asc" />
        <property name="value" direction="desc" />
    </datastore-index>
</datastore-indexes>
```

14.2 More Flexible Models

One other important thing that comes up in datastore is model restrictions. The models we've built so far have suffered from a lot of restrictions. Every instance of a type has to have exactly the same properties. There's absolutely no flexibility about it with standard models. But sometimes that's just too restrictive. When that happens, you've got two choices to work around it: expando models and polymodels.

An *expando model* is a model in which you can arbitrarily add new properties to a model instance anytime you want. It's basically the ultimate in model flexibility. You can add anything you want, at any time you want. All of the model instances—all of the objects that you store—need to be instances of the same Python class.

A *polymodel* is a model in which you define a superclass of model instances. Then you can add subclasses of that class. When you do a query, it will retrieve all of the instances of the polymodel base class as well as the polymodel subclasses.

Before you use expando or polymodels, stop and think. In my experience, most of the time, if you can't define your model cleanly without either polymodels or expandos, it's an indication that you haven't thought things through enough. There are cases where you might need them, but when you really need them, they're there for you. This is especially true of expando models: the extreme flexibility of the expando model is sort of like the extreme flexibility of gotos in flow control: there are places where they make sense, but there are also places where they're the easy solution, not the right one.

In the case of a polymodel, things are nice and clean. You define a model class exactly the way that you define a standard model except that the base class inherits from polymodel.PolyModel. Then you define subclasses of the polymodel in the usual way.

For example, we could take our filesystem and add typed attributes by making the ResourceAttribute a polymodel. Then we could add subclasses of ResourceAttribute for string attributes, integer attributes, and so on:

Download filesystem/poly.py

```
class ResourceAttribute(polymodel.PolyModel):
    name = db.StringProperty(required=True)

class StringAttribute(ResourceAttribute):
    strVal = db.StringProperty(required=True)

class IntegerAttribute(ResourceAttribute):
    intVal = db.IntegerProperty(required=True)
```

If you really must use an expando model, it's very simple—you make the class inherit from google.appengine.ext.db.Expando. In an expando model, you don't need to define the properties at all. You can define a set of fixed properties, which will be included in every instance of the model. But also, every field of the Python object will implicitly be treated as an untyped property. When you do this, you lose all of the help that datastore normally gives you. You don't get automatic references; you don't get any data validation. It's totally flexible and totally unstructured. You probably don't want to use it. I've yet to see an example of an expando model that really needed to be an expando. I'm not going to show you an example of one, because I can't think of any case where it really makes sense.

I could define a version of our filesystem that used an expando model. I could get rid of ResourceAttribute entirely and just use fields of the Python resource objects to store properties. But that would be a lousy solution. I'd lose the ability to define queries over attributes, and I'd lose a lot of structural consistency. What would I gain? Frankly, not much. I'd get a slightly better syntax for updating attributes, and that's really all.

14.3 Transactions, Keys, and Entity Groups

Back in Section 10.2, *Storing Persistent Objects in GWT*, on page 145, we talked about transactions, why they're so important, and how they work automatically in Java. It turns out that in Python, we've got transactions too, we just haven't bothered to look at them. Like lots of other things in App Engine with Python, the default behavior is set up so that most of the time, it's the right thing. But as usual, there will be

cases where you'll want something other than the default, so you need to understand how it really works.

Transactions in Python are driven by something called *entity groups*. Every persistent object that you create in Python is part of a group of objects called an entity group. Every time you store a change to the datastore from Python, all of the changes to objects within the same entity group are saved within a single transaction. So if you can arrange for everything that you need to store to be within a single entity group, you don't need to worry about transactions: it will just work.

Entity groups are managed using ancestor relationships: every object is automatically in the same entity group as its parent object. You create parent relationships at object creation time: when you create a persistent Python object, you can specify its parent object by using a named parent parameter. If you don't specify a parent, then the object is a root; it defines its own entity group.

If you don't need transactional behavior within a collection of objects, then it's best to just let them be roots. Transactions are incredibly useful, but they're not cheap. In terms of datastore and the App Engine distributed environment, an entity group can't be distributed. It needs to be located together on a single server. And that means when you scale up to large numbers of users and objects, putting too many things into the same entity group can make your application perform *really* badly.

For example, in our chat application, we could put all of the chat messages into a single entity group. There would be some advantages, such as providing a very strong guarantee of consistency. But over time, we could easily have tens or hundreds of thousands of chat messages in hundreds of chat rooms, and adding a message to any chat room could block updates from users of other chat rooms while its transaction was in progress.

On the other hand, we don't want to forget about transactions either. In our filesystem, if users update a couple of attributes of a single resource and then save it, they expect that the update will either succeed or fail. So it makes sense in the case of our filesystem resources to make attributes the children of their resources, and thus make attribute updates become part of a transaction with their parent objects. The only change we need to make to do that is in the SetAttribute method of Resource:

Download filesystem/filesystem_servlet.py

```
def SetAttribute(self, name, value):
  if name == "children":
    self.children = [ de.key() for de in value ]
    self.put()
  else:
    for attr_key in self.attributes:
      attr = ResourceAttributeModel.get(attr_key)
      if attr.name == name:
        attr.value = value
        attr.put()
        return
      newAttr = ResourceAttribute(parent=self, name=name, value=value)
      newAttr.put()
      self.attributes.append(newAttr.key())
      self.put()
```

The only difference between this and the setAttribute method we've seen before is the call to create a new ResourceAttribute object. In this new version, we provide a parent parameter for the ResourceAttribute, which puts the attribute into the same resource group as the Resource object itself.

Aside from transactions, the resource group has an effect on an object's Key. The key encodes information about the resource path—that is, the canonical path from a root object to the specific object. This can be used in queries: you can narrow a query to include only objects that have a specific object as their ancestor. That's done using a special query clause ANCESTOR IS in the query. For example, in our filesystem with attributes having resources as a parent, we can query for an attribute value of a specific resource. If we wanted the length attribute of a resource R, we could write a GQL query like the following:

```
query = db.GqlQuery("SELECT * FROM ResourceAttribute WHERE "
"name = :name AND ANCESTOR IS :parent",
name="length", parent=R.key())
```

14.4 Policy and Consistency Models

The last thing that we need to learn about datastore involves the idea of *policy*. Policy is a general term that describes high-level options that define how the system should behave. In datastore, the set of which indices are generated automatically is a matter of policy. There are two main policy issues that affect datastore queries that you can control:

- Consistency models: the way that the system handles consistency between different instances of an application running within the App Engine cloud. Consistency policy is, without a doubt, the most critical policy issue. We'll say a lot more about consistency models below.
- Deadlines: the maximum amount of time that a query should be allowed to take before it fails. Sometimes you simply can't afford to wait for something—if you can't generate a response within a particular period of time, there's no point in generating it at all. For example, in our chat application, if responding to an update query takes longer than a second, there's no point in responding, because another update request will already have been queued—if responding to every query takes longer than the period of time between queries, we're going to develop an increasing backlog of queries. Better to have the queries fail and return than to develop a continually increasing backlog! A deadline lets you say, "If this is going to take too long, then just let it fail instead of waiting."

As I said above, the most important policy issue involves consistency models. By default, datastore uses something called strong consistency. Strong consistency means that any two clients that make the same query are always guaranteed to get the same results. That seems like an obvious requirement: if a given query doesn't always generate the same results, something must be wrong, right?

Not necessarily. If the data were static, then any query would have one precise, well-defined result, and every one would always return that result. But updates complicate that. A result is well defined with respect to a particular time. But, as we discussed before, time in a distributed environment is tricky.

Imagine a scenario like the following:

1. Client 1 fetches resource R1 and sets attribute A to the string abc.
2. Client 2 fetches resource R1.
3. Client 1 updates attribute A of R1 to the string def.
4. Client 2 fetches attribute A of R1.

What value should client 2 get for attribute A of R1?

According to strong consistency, it should get def. The attribute was updated, and client 1 sees the update, so every other client must also

see the update. For many applications, that's really what you want. For example, if you're running a web store, and you tell a user that there's one item left, you don't want to sell that item twice: as soon as anyone buys it, the fact that your inventory now has zero of that item should be visible to everyone immediately. Strong consistency guarantees that. In the lingo, strong consistency is conservative. It makes sure that it always provides the right results to everyone no matter what the cost.

But strong consistency is expensive. In our example, client 2 did a full retrieve of the resource before client 1 did the update. For client 2 to see the updated value of the attribute, it needs to send a new query to the server to retrieve the attribute through its reference. It can't have just fetched everything at once. That round trip is not free; it comes at a significant cost. Sometimes, that cost just isn't worth paying.

There's another consistency policy, called eventual consistency, which is less expensive. In eventual consistency, what happens is that eventually, all parties will agree about the values of data. But there might be a short period of time where they disagree. In other words, it's sort of consistency with fuzz: whenever you do an update, there's a fuzzy period where someone might see the old, un-updated value.

For the database folks, this is called BASE consistency (Basically Available, Soft state, with Eventual consistency) as opposed to ACID (Atomic, Consistent, Isolated, Durable state).

Eventual consistency is good for a lot of things. For example, in our chat room example, eventual consistency could mean that two different clients disagree about when the last message was posted. But since they're doing queries to check twice a second, they'll never disagree for longer than half a second! For a chat room, that's perfectly acceptable.

Similarly, for a filesystem, people generally expect that if two programs try to change the same file at the same time, the results are likely to be screwed up. You can get some race conditions in things like attribute values if you use eventual consistency, but there will inevitably be race conditions in attribute values with multiple clients making concurrent updates. Strong consistency slows things down and eliminates some but not all races.

So, suppose that we want to use eventual consistency for a query. The query will, maybe, return results that are slightly out-of-date, but we understand that, and we're OK with it. How do we go about actually using it?

Basically, we have to push our way down into the depths of the datastore interface. Deep down, datastore uses a Google asynchronous RPC system like the one that we saw in GWT in Chapter 12, *Building the Server Side of a Java Application*, on page 171, and we need to change an internal parameter of the RPC used by the query. Policy issues are determined by the RPC object, so to change the policy, we need to change the RPC object.

The fetch method of Query actually takes a parameter named rpc. It's defined with a default parameter value, which makes it create a new RPC object with standard parameters. When you want to change policy features, you need to create your own RPC object with the correct policy parameters.

You create RPC objects by calling db.create_rpc. It takes two named parameters: read_policy and deadline. To use eventual consistency, set read_policy=db.EVENTUAL_CONSISTENCY; you don't need to specify strong consistency, but if you really want to, you can explicitly set it to its default value using read_policy=db.STRONG_CONSISTENCY. For deadline, the value of the parameter is the minimum number of seconds you want to wait before the query fails. There's no guarantee that it will fail immediately after the deadline passes—the only guarantee is that it won't fail before the deadline.

Once you have an RPC object, you can supply it as a parameter to a query. So, for example, if q were your query, you could make it use eventual consistency with a timeout of two seconds using this:

```
q = ...
my_rpc = db.create_rpc(deadline=2, read_policy=db.EVENTUAL_CONSISTENCY)
result = q.fetch(rpc=my_rpc)
```

Similarly, if you're accessing the query result using a Python iterator, you can supply the RPC object in the run method instead of the fetch method:

```
q = ...
my_rpc = db.create_rpc(deadline=2, read_policy=db.EVENTUAL_CONSISTENCY)
for r in q.run(rpc=my_rpc):
    ...
```

Of course, as in so much of the advanced features: just because you can doesn't mean that you should. Most of the time, you're much better off doing the fetch. fetch is a lot more efficient.

The story in Java is similar, except that you don't use the RPC object explicitly. Instead, the JDO API that's used by App Engine provides methods for setting policy directly on the query object. Instead of creating an RPC object with the policy settings and passing that to a query method, you set the policy settings directly on the query object:

```
Query q = persister.newQuery(ChatMessage.class);
q.addExtension("datanucleus.appengine.datastoreReadConsistency", "EVENTUAL");
q.setTimeoutMillis(2000);
```

14.5 Incremental Retrieval

By default, when you do a query in datastore, it returns all of the results of that query at once. In terms of performance, that's usually the best choice: it's likely to provide the best runtime performance under most conditions. But sometimes, a query returns a lot of data—too much for you to hold in memory at once. So you need to change the query policy to one that accesses data in chunks.

Whether queries work by total retrieval or incremental (that is, chunk-by-chunk) retrieval is really a matter of policy. But it's a kind of policy that, frankly, is supported much better than what we saw with consistency. The different consistency models are supported rather sloppily. The App Engine designers correctly thought that strong consistency was almost always the right thing. And so they deliberately made it awkward to use eventual consistency. You need to really be sure that you want eventual consistency because getting it is a serious pain. But incremental queries? They're downright routine, so they're supported much more smoothly.

For incremental queries, use something called a *cursor*. You do a query, but when you do it, you provide it with a limit on the number of results it should return. Then you can iterate over those as normal. Because you did a bounded retrieval, you can now get a cursor that points at the location of the first query result that comes after the results that you've already retrieved.

To do a bounded query, you just provide a numeric parameter for the query. For example, to do a query over all resource objects but only retrieve them fifty at a time, you'd start by doing a bounded query:

```
q = Resource.all()
batch = q.fetch(50)
```

Then, once you were done iterating over the first batch, you could get a cursor for the query by calling q.cursor().

For the next batch, you'd then use that cursor by setting the cursor for the query as follows:

```
q = Resource.all()
batch = q.fetch(50)
c = q.cursor()
...
q.with_cursor(c)
q.fetch(50)
```

After each batch, you can retrieve a cursor from the query object. Then, when you do the next query, you can provide it with a cursor to use as a starting point. In fact, the cursor is capable of being used as a persistent object itself: you can save a cursor, and then use it later. (That will become particularly useful when we look at the Memcache service in the next chapter.)

In these last two chapters, we've taken a look at some of the more advanced features and the limitations of the App Engine datastore. This is still just scratching the surface: datastore is constantly evolving to provide more and better capabilities, so you'll definitely want to check the App Engine documentation. But the key features and the key design points will remain the same.

As we've seen, the real secret to datastore is understanding the basic model that it uses. It's a system for storing objects; everything else—from the transaction system to the query language to the key mechanics to references—derives from that basic principle. Datastore is all about storing persistent object—not tables, not collections, not lists, not trees.

The App Engine datastore is a flexible, lightweight, object-based persistence mechanism, *not* a relational database. If you keep that in mind, you'll find that it usually behaves in an intuitive way.

Chapter 15

Google App Engine Services

One of the major focuses of modern software engineering is reuse. We don't want to be constantly reinventing the same thing. For the most common kinds of software components, we create libraries that contain implementations, and then we just use those instead of writing our own. For example, in C++, we don't write a new hashtable implementation every time we build an application that needs one: we just use the map types from the standard template library. When we want to build a GUI in a Python application, we don't start writing code that draws buttons using individual pixels; we grab a library like wxWindows and use it.

In the cloud, we do pretty much the same thing, only instead of using libraries, we use services. The difference between a library and a service is subtle but important. A *library* is something static. It's a pile of code that we have on our computer that we can include in our application. When we use a library, the library's code becomes part of our application. In the cloud world, a *service* is something that is running somewhere in the cloud. It's essentially an application itself, but it's an application that's running to provide some kind of capability to other applications.

Services aren't entirely new to us. In the last couple of chapters, we've been looking in detail at the datastore, one of the services that's provided by App Engine. And in Chapter 5, *Google App Engine Services for Login Authentication*, on page 65, we saw how another App Engine service could be used for managing user logins.

Just as we have lots of libraries for standard desktop development that provide implementations of different things an application might need,

in the cloud world (and in App Engine in particular), we have lots of services that provide capabilities our cloud applications might need. In this chapter, I'm going to discuss a collection of the common services that App Engine provides. It's not going to be an exhaustive examination—there are a lot of services for App Engine development, and the App Engine team at Google is continually expanding the set of services that they provide. But we'll look at a sample of the services to get a sense of what kinds of capabilities services they provide and how interfaces to services work.

This chapter is a bit of a grab bag. App Engine provides lots of services, and for the most part they're very simple. They aren't worth dedicating a whole, focused chapter to individually, but they don't fit together terribly well. You may want to pick and choose services from this chapter as you need them.

15.1 Accessing Important Stuff Quickly: The Memcache Service

Memcache is one of the services that you'll probably use most in real applications—and it's so simple that it only takes a paragraph or two to explain. In every App Engine application you build, you'll store data in the datastore. The one problem is that retrieving things from the datastore takes time. When you first consider the time it takes to retrieve, it seems trivial—it's typically a tiny fraction of a second. But when you think about the fact that in a live service you could be serving thousands of requests a second, that time adds up.

You won't be surprised to find memcache provides exactly what its name suggests: an in-memory cache. You always keep the primary copy of your data in the datastore. But you can also put a copy into Memcache, and once it's there, retrieving it is essentially instantaneous.

Memcache is a great example of a simple service. It provides its capabilities through a simple library API. Behind the scenes, it's doing some very subtle stuff to provide a cache that's accessible from machines throughout the App Engine cloud, but all of that is invisible to you as a client of Memcache. You just have a couple of functions you can call, which look like a simple in-memory cache. It's a very elegant little service, and it's incredibly useful.

There are, of course, some catches to Memcache. First, is unreliable storage. There's no guarantee that something you've stored in Memcache will be there later. It's a shortcut for getting things quickly, not

a replacement for reliable persistent storage like the datastore. If you don't access a cached object for a while, or if you put too many things into the cache and it runs out of space, it will start to discard. Whenever you use Memcache, you need to include a fallback to datastore. Don't assume that just because you put something into Memcache you'll automatically be able to get it back!

Second, Memcache is intended for storing small things. Any request that you send to Memcache has a maximum size of 1 megabyte. So you can't store anything larger than 1 megabyte, and if you want to access multiple objects with a single request (whether a get or a put), the total size of all of them can't exceed 1 megabyte.

So, for example, if we wanted to use Memcache in our chat application, we couldn't use it to provide fast access to the list of all chat messages: that could easily grow to be more than 1 megabyte. But we could use it to store a list of all of the available chat rooms: the way we set things up, the list of chat rooms won't ever get bigger than a megabyte.

Using Memcache in Python

In Python, Memcache is basically a simple key/value dictionary. You put objects into it with a string-valued key. In Python, you can store an object in the cache with code like the following:

```
from google.appengine.api import memcache

memcache.set(key="aString" value=aValue)
```

And retrieving is just as easy:

```
v = memcache.get(key="aString")
```

get will return the value or return None if the key wasn't in the cache.

You can also do multiple updates/retrieves in one call. To set multiple values, use set_multi and give it a dictionary as a parameter:

```
memcache.set_multi({ "key1": "value1", "key2": "value2"})
```

To retrieve multiple values, use get_multi and pass it a list of keys. It returns a dictionary of the key/value pairs:

```
dict = memcache.get_multi(["key1", "key2"])
```

Finally, as I mentioned before, Memcache will throw things away if they're not accessed for a while or if it runs out of space. You can give Memcache a hint about how long it should wait before it dumps something by providing a time parameter. The time parameter basically says,

"Please keep this in memory for at most this long." There's no guarantee it will stay stored for that long; if the cache runs out of space before then, it will still discard the value. But it won't dump it because it's been unused for too long until that time period expires. The time parameter is the number of seconds that you'd like the value to be kept. So, for example, to store something for at least two hours, you could use memcache.set(key="foo", value="bar", time=7200).

Using Memcache in Java

In the usual App Engine style, when it comes to providing access to Memcache for Java, App Engine builds on a standard Java API. There is a proposed standard mechanism for working with caches in Java, described in Java Standards Request document JSR107. App Engine provides access to the Memcache from Java using the JSR107 API. And again, as usual, an API designed as part of a standards process is more cumbersome than a custom-designed one. So using Memcache from Java takes a bit more boilerplate. But conceptually, it's still the same as what we saw in Python. In fact, it has to be, because they share the underlying implementation. The surface API is different, but the service is exactly the same. A cache is just a map from keys to values. In Java, the values need to be serializable—exactly the way that they need to be if you want to put them into the datastore. Previously, we saw that to put something into Memcache in Python, we could just do a memcache.set. In Java, with the boilerplate, the code might be:

```
import java.util.Map;
import net.sf.jsr107.Cache;
import net.sf.jsr107.CacheException;
import net.sf.jsr107.CacheFactory;
import net.sf.jsr107.CacheManager;
import com.google.appengine.api.memcache.stdimpl.GCacheFactory;

class MyApplicationClass {
   Cache cache;
   Map cacheConfig = new HashMap();
   {
      try {
        CacheFactory factory = CacheManager.getInstance().getCacheFactory();
        cache = factory.createCache(cacheConfig);
      } catch (CacheException e) {
        // don't worry about it
      }
   }
   cache.put(aKey, aValue);
  // ...
}
```

Yes, it's a bit verbose compared to the Python version. And it will generate a warning from the compiler because you're using an untyped map. Also, you've got that worrisome empty catch clause. Don't sweat it. The Map parameter takes a set of implementation specific key/value pairs, but it deliberately leaves the types unspecified so that an implementation can use whatever keys and values it wants. And in App Engine, you'll always get a cache instance back: the CacheException will never be thrown by the CacheFactory, so you can leave the catch block empty.

Once you've got the boilerplate out of the way, it becomes pretty much as simple as in Python. To store an object, use cache.put, and to retrieve an object, use cache.get:

```
cache.put("aKey", aSerializableValue);

value = (ValueType)cache.get(aSerializableValue);
```

Also, there are getAll and putAll methods that work just like the Python get_multi and put_multi. getAll takes a collection of keys and returns a map of key/value pairs; putAll takes a map of key/value pairs and updates all of them.

Unfortunately, the JSR107 standard that you use to access Memcache from Java doesn't allow you to set the cache expiration time on individual objects. In Java, it's a policy property of the cache. When you get the cache, you can provide configuration parameters, and one of those is the cache expiration time. So, for example, to request that all cached objects stay around for at least two hours, you'd add a parameter to the properties map when you create the cache:

```
class MyApplicationClass {
   Cache cache;
   Map cacheConfig = new HashMap();
   cacheConfig.put(GCacheFactory.EXPIRATION_DELTA, 3600);
   {
      try {
        CacheFactory factory = CacheManager.getInstance().getCacheFactory();
        cache = factory.createCache(cacheConfig);
      } catch (CacheException e) {
        // don't worry about it
      }
   }
  // ... rest of the class
}
```

What Should You Cache?

Memcache is a system for unreliable storage of a small number of frequently accessed things. You don't want to just automatically dump everything into the cache—you need to be careful and selective and only put things into the cache when they're really, genuinely something that you're going to constantly access.

For example, in our filesystem application, every request will need to fetch the root directory. Requests will always come in with the pathname of the resource we want to fetch or update, and to find that resource, we'll start with the root directory resource and use it to look up the specific resource that's the target of the request. We'll constantly need the root directory resource. So *that* should be in the Memcache. On the other hand, a typical resource that we fetch probably shouldn't be in the Memcache. In between is where things get tricky: directories low in the hierarchy will get accessed relatively frequently; the higher up the hierarchy they go, the less frequently they'll probably be retrieved. In the case of the filesystem, they probably don't need to be cached.

As I mentioned before, in the case of our chat application, it definitely doesn't make sense to put all of the chat messages into the cache. But it probably does make sense to keep the list of available chats. The chat objects are small, and they're going to be accessed constantly.

The Cache Access Pattern

We saw before how to retrieve something from the Memcache. But because Memcache is unreliable, the value we're looking for might not be there. So in real applications, when we use the Memcache, we'll always retrieve values using a retrieval pattern that first tries the cache and then falls back to a datastore query if it can't find the object.

In the filesystem, we always actually retrieve the filesystem object. It's got a reference property for the root directory resource, so it doesn't look like we use a query to get to the root directory. It looks like the real query is for the filesystem object, and then it accesses a property of the filesystem object to get the root directory. But remember that reference properties are actually automatic queries. So by cacheing the root directory, we can eliminate two queries!

In the original version of FilesystemResourceHandler.get we fetched the root object by calling getFilesystem, which in turn did a GQL query to

retrieve the filesystem object, and we then retrieved the root object from the filesystem. In the Memcache version, it changes to the following:

Download filesystem/cached_filesystem_servlet.py

```
root = memcache.get(key="root")
if root is None:
    query = Filesystem.gql("")
    filesystem = query.get()
    root = filesystem.getRoot()
    memcache.put("root", root)
```

This is exactly the pattern that you'll use again and again with Memcache: first, try to retrieve the object from the cache. If it isn't there, do a datastore query to retrieve the object and then put it into the cache. Then go ahead and use it. So if the object is in the cache, you get it instantly; if it's not in the cache, you do a query and retrieve it. Then you put it into the cache so that next time you need it, you'll be able to get it instantly.

15.2 Accessing Other Stuff: The URL Fetch Service

Web services and applications interact through HTTP. If you want to write an application that interacts with other web services, you'll need to write code that sends HTTP queries. Since this is so common, instead of making you write HTTP protocol interactions by hand, App Engine provides a service.

In fact, both Java and Python provide a standard set of libraries for doing HTTP interaction. In Python, there are three different libraries: urllib, urllib2, and httplib. Java is (as usual) a bit more rigorous: it's got one standard HTTP library, java.net.URLConnection. You can use any of these libraries in App Engine exactly the way that you would in standard Python or Java, but under the covers, App Engine replaces their implementation with a custom version that's optimized to be both efficient and scalable in the App Engine environment.

So, if our filesystem service were running at mcfile.appspot.com, another Python application could retrieve a resource markcc/book.html using code like this:

```
import urllib

result = urlfetch.fetch("http://mcfile.appspot.com/markcc/book.html")
if result.status_code == 200:
  # fetch successful: file contents in result.content
```

Be aware that, like everything else in a cloud application, you pay for what you use. In App Engine, you're going to pay for the bandwidth that you use, so you don't want to do this too much. Writing something like your own web crawler using App Engine isn't going to be cheap.

15.3 Communicating with People: Mail and Chat Services

In addition to HTTP, there are other protocols an App Engine application or service could use to talk to other things. For example, you might want an application to be able to send an alert to a user using instant messaging. It's not exactly common, but there are places where having your application interact with users via instant messaging makes sense.

App Engine provides a service for doing that. And it's remarkably easy: you don't need to worry about exactly what IM service you want to work with. Over time, the various chat services have agreed on a standard protocol for describing instant messages, called XMPP (eXtensible Messaging and Presence Protocol). App Engine provides a service that lets you use XMPP to send and receive chat messages. App Engine provides a standard interface to all XMPP services: by using App Engine's XMPP service, you can talk to pretty much any user on any of the standard chat services—AIM, Google Talk, MSN Chat, whatever. (Under the covers, Yahoo and MSN chat use proprietary protocols, but there are XMPP gateways, which means that you can talk to them if you're willing to set up a connection to the gateway.)

The way that the XMPP service is set up is very typical of how App Engine handles communication services. The same pattern is used by services that allow you to send and receive email, send and receive large chunks of stored data, receive notifications about queued tasks, and more.

For sending messages, it provides an object that allows you to perform sends using method calls. This is pretty much the kind of interface that we've seen for all of the services we've used so far. For receiving messages, it provides an HTTP facade. That is, it takes the incoming message and presents it to you as an HTTP request at a particular URL. In order to handle incoming messages, you just implement a standard HTTP message handler/servlet for the service's incoming URL. This can seem a bit odd—why take an IM and turn it into an HTTP POST request? But the point of it is that you only need to know how to write one kind

of handler. Everything that you write that handles incoming data in App Engine is always an HTTP PUT or POST handler. There's only one interface you need to know, only one kind of handler that you need to worry about. App Engine calls this facility *webhooks*: the ability to interact with all sorts of protocols and services are provided in App Engine by using webhooks that provide a bridge to present requests in other protocols as if they were HTTP requests. Webhooks are one of the best features of App Engine: you can interact with any service without needing to know the details of its protocols—you just need to write HTTP servlets.

Sending Chat Messages

There are really just two methods that you need to be aware of: one for checking if a user is logged in and one for sending a message. To check, use xmpp.get_presence. So, for example, to check if I'm logged in and then send me a message, you could do the following:

```
from google.appengine.api import xmpp

...

if xmpp.get_presence("markcc@gmail.com"):
  status_code = xmpp.send_message("markcc@gmail.com", "Hi Mark!")
  if status_code != xmpp.NO_ERROR:
    self.response.setStatus(500, "Sorry, couldn't send a message to Mark")
```

Or, in Java:

```
import com.google.appengine.api.xmpp.JID;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.MessageBuilder;
import com.google.appengine.api.xmpp.SendResponse;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;

class JavaXMPPExample {

  public sendMessage() {
    // Get an identifier object for the user.
    JID jid = new JID("example@gmail.com");
    // Build the message.
    Message msg = new MessageBuilder()
            .withRecipientJids(jid)
            .withBody("Hi Mark!)
            .build();
    boolean messageSent = false;
    // Get the XMPP service.
    XMPPService xmpp = XMPPServiceFactory.getXMPPService();
```

```
    // Check if I'm logged in.
    if (xmpp.getPresence(jid).isAvailable()) {
      // send the message.
      SendResponse status = xmpp.sendMessage(msg);
      if ((status.getStatusMap().get(jid) != SendResponse.Status.SUCCESS)) {
      }
    }
  }
}
```

As usual, the Java is a bit more complicated: you need to create an identifier object for the user instead of just directly passing a string. But once again it's just boilerplate: write it once, and then reuse it.

Receiving Instant Messages

Receiving messages from XMPP is a bit harder than sending, but it's still pretty straightforward. As far as your App Engine code is concerned, XMPP is nothing but a stylized way of using HTTP. With the App Engine webhooks facade, it appears as if it's all implemented on top of HTTP in terms of POST to send a message. You don't need to worry about how XMPP really works: App Engine takes care of that for you. All you need to worry about is how to handle the HTTP request sent to you by App Engine. So to set up your application to receive XMPP messages from the XMPP service, you need to do three things:

1. Tell App Engine that you want to receive XMPP messages. You do that by putting an entry in your application configuration file—that's web.xml in Python or appengine-web.xml in Java.
2. Write an XMPP handler, which is really just a standard HTTP request handler that implements POST.
3. Register your XMPP handler. App Engine will always map XMPP messages to a specific URL: /_ah/xmpp/message/chat/, so you need to register the XMPP handler as the handler for that URL.

With that set up, people can send chat messages to your application by sending a message toanything@your-app-id.appspot.com.

Handling Chat Messages in Python

In order to receive an XMPP message in Python, you first need to put the following entry into app.yaml:

```
inbound_services:
- xmpp_message
```

Then write an XMPP handler:

```
class XMPPHandler(webapp.RequestHandler):
  def post(self):
    message = xmpp.Message(self.request.POST)
    message.reply("Received message: %s\nHello to you too" % message.body)
```

You just take the POST request, have the XMPP service convert it to an XMPP message object, and then get the message (message.body), the sender (message.sender), or the intended recipient (message.to).

Finally, you need to register that handler:

```
application = webapp.WSGIApplication([('/_ah/xmpp/message/chat/', XMPPHandler)])
```

Receiving Chat Messages in Java

The basic steps in Java are pretty much the same as Python, but as usual, it's slightly different stylistically to match the differences between the languages. First, you tell App Engine that you want to receive chat messages by adding an entry to the application configuration in appengine-web.xml.

```
<inbound-services>
   <service>xmpp_message</service>
</inbound-services>
```

Then, write a message handler:

```
import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.xmpp.JID;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;

public class XMPPReceiverServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse res)
          throws IOException {
        XMPPService xmpp = XMPPServiceFactory.getXMPPService();
        Message message = xmpp.parseMessage(req);

        JID fromJid = message.getFromJid();
        String body = message.getBody();
        // ...
    }
}
```

And finally, register this handler by adding servlet and servlet-mapping entries to your web.xml file, as shown in the following configuration:

```
<servlet>
  <servlet-name>xmppreceiver</servlet-name>
  <servlet-class>XMPPReceiverServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>xmppreceiver</servlet-name>
  <url-pattern>/_ah/xmpp/message/chat/</url-pattern>
</servlet-mapping>
```

15.4 Sending and Receiving Email

Along the same lines as XMPP, you can also send and receive email in App Engine. Although it might seem a bit silly at first, if you think about it for a moment, sending mail is actually much more common and useful than sending instant messages.

Email is commonly used for registration on sites, for notification, and for verification. When you first use a site that provides a chat service, you typically have to register—and you'll get registration by email. If you own a store, you'll typically send receipts and confirmation by email. If a user needs to reset a password, you'll typically do it through email. If users want to know when something is modified—whether it's someone adding something to a discussion in a chat room while they're not logged in or when an out-of-stock item becomes available in a store—they'll want it through email.

So sending email is pretty common. It's not as ubiquitous as something like persistent data, but it's not an unusual thing like sending instant messages. App Engine provides a very simple, flexible service that allows your application to both send and receive email.

Sending Mail

Sending mail couldn't possibly be easier. In Python, once you import the mail service, it's one line of code. To send a message with the subject "Hi Mark!" and with the message body "I'm still running" to me from chat-service@markcc-java-chat.appspot.com, all you'd need to do is this:

```
from google.appengine.api import mail

mail.send_mail("chat-service@markcc-java-chat.appspot.com", "markcc@gmail.com",
     "Hi Mark!", "I'm still running")
```

From Java it's a bit more complicated. Java provides a standard API for sending email in the javax.mail package, and App Engine just provides

an implementation of that. The Java code equivalent to the simple one-line Python script above is this:

```
import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

...
    Session mailSession = Session.getDefaultInstance(new Properties(), null);
    try {
      Message msg = new MimeMessage(session);
      msg.setFrom(new InternetAddress("chat-service@markcc-java-chat.appspot.com"));
      msg.addRecipient(Message.RecipientType.TO,
          new InternetAddress("markcc@gmail.com"));
      msg.setSubject("Hi Mark!");
      msg.setText("I'm still running!");
      Transport.send(msg);
   } catch (AddressException e) {
      // what to do if the address was invalid
   } catch (MessagingException e) {
      // what to do if there was an error sending it other than
      // a bad email address
   }
```

Receiving Mail

Receiving mail is a bit more complicated, but not by much. It follows the usual App Engine pattern for receiving data via a service. You tell App Engine that you want to receive data from the service by putting an entry into the application configuration. Then App Engine will route incoming messages from that service to a virtual HTTP URL, where it can be handled by your message handler/servlet. In the interests of simplicity, I'll show you a quick example of how a Python application can receive email; Java is, as usual, pretty much the same but a bit more verbose.

Step one of the pattern is telling App Engine that your application wants to receive incoming email using the email service. To do that, you just add an inbound_services entry to your app.yaml file:

```
inbound_services:
- mail
```

Once you've set that up, your application is ready to receive email. You'll get any email addressed to a username at your-app-id.appspotmail.com. You'll need to set up a handler for incoming email. Email will be received at the virtual URL /_ah/mail/receiver-address. So, for example, if someone sends mail to admin@markcc-chatroom-one.appspotmail.com, my application would receive that email on the URL /_ah/mail/admin@markcc-chatroom-one.appspotmail.com.

In order to handle the email, you need to set up a handler for the email URLs. Suppose that I wanted to add an email capability to the chat service so that people could send mail to the admin address, and that mail would be automatically forwarded to me. What I would need to do is add a script for handling the incoming email. Since it's being disguised as an HTTP request, if I were to implement it as just a simple RequestHandler, I would need to parse the incoming request into a mail message. But instead of making me do that manually, App Engine provides a InboundMailHandler class that I can inherit from. It provides an implementation of post that takes the incoming message, parses it into a InboundEmailMessage, and then invokes receive on the message. So all I need to do is provide an implementation of receive, such as the following:

Download multichat/chatmail.py

```
import email
from google.appengine.ext import webapp
from google.appengine.ext.webapp.mail_handlers import InboundMailHandler
from google.appengine.ext.webapp.util import run_wsgi_app

class ChatMailHandler(InboundMailHandler):
    def receive(self, mail_message):
        mail.send_mail(sender="admin@markcc-chatroom-one.appspot.com",
                       to="markcc@gmail.com",
                       subject="CHAT ADMIN MAIL: %s" % mail_message.subject,
                       body="Original message from: %s\n%s" %
                       (mail_message.sender,
                        mail_message.body)

chatmail = webapp.WSGIApplication([InboundMailHandler.mapping()])

def main():
    run_wsgi_app(chatmail)

if __name__ == "__main__":
    main()
```

Once I have the handler, I tell App Engine that it should route messages received at the email URL to the script containing that handler by adding an entry to app.yaml:

```
- url: /_ah/mail/.+
  script: chatmail.py
```

The InboundEmailMessage contains fields for all of the headers and body elements of an email message:

sender
: The email address of the message sender.

to
: The email addresses to in the To: header.

cc
: The email addresses in the cc: header.

reply_to
: The email address that replies should be sent to, as specified by the Reply-To: header.

subject
: The value of the message subject.

body
: The content of the message body.

attachments
: A list of attachments from the message. This is a list of pairs, where the first element is the attachment name and the second is the contents of the attachment.

15.5 Wrapping Up Services

In this chapter, we took a quick look at the idea of services in App Engine. We've seen a collection of the most common App Engine services, including services for providing high-speed access to commonly used data objects and for doing HTTP requests to communicate with other web services, as well as some basic communication services for sending and receiving email and chat messages. In the process, we saw what services look like and how we can interact with them. We saw how App Engine allows us to implement handlers for incoming data, even when it isn't really HTTP by using pseudo-URLs and HTTP facades. And we used a couple of services to enhance the applications we built before. We used Memcache to provide caching for the filesystem we've been working on, and we used the email service to allow our chat application to receive email from its users.

In the next chapter, we'll look at how we can use and build App Engine services that perform serious computation on App Engine servers. For the first time, instead of our applications being completely responsive—only performing an action when a user specifically asks them to—we'll set up our application to run tasks on the server automatically, without any user intervention. In the process of doing that, we'll use more App Engine services: for scheduling things to happen at a certain time, queuing tasks to run somewhere in the App Engine cloud, and sending results from those computations to other applications.

Chapter 16

Server Computing in the Cloud

So far, all of our App Engine programs have been completely passive and request-driven. They sit on the App Engine servers, doing absolutely nothing until a user initiates some kind of request; they immediately respond to that request, and then go back to doing absolutely nothing. For a lot of applications, that's exactly what we want. A chat room doesn't really need to do anything except send messages to people when they ask for them. A filesystem doesn't read or write files on its own; it just does things when someone specifically asks it to.

But sometimes the passive approach doesn't work. For example, if you were building an e-commerce site, you might want a daily sales summary. Or if you were building a collaborative calendar application, you might want to be able to notify people of upcoming events. Those things aren't passive. They require the server to run some code without being triggered by a request, based on timing or the occurrence of some data-based event.

App Engine provides two mechanisms for non-passive server-based computation: one for performing actions according to a fixed schedule and one for performing them according to an event-based JavaScript mechanism. In this chapter, we'll look at these two mechanisms and see how we can use them to run tasks on the server.

This is one of the areas where App Engine really shines. As we've seen throughout this book, App Engine is structured around HTTP request handlers. That's consistent in server computing as well: you still write all of your code as simple HTTP request handlers. Because of the way App Engine lets you use those request handlers, you can build arbitrarily complicated workflows for the things you need to compute on

the server. It's an extremely elegant way of doing things that gives you the ability to do whatever you need without adding a lot of complexity.

We'll start by looking at the simplest form of server computing—doing things according to a regular schedule. After that, we'll move on to the more powerful and flexible realm of task queues, where processing user requests can dynamically fire off complex sequences of computations on the server.

16.1 Scheduling Jobs with App Engine Cron

As an administrator of a cloud-based application, chances are good you're going to want to find out how your system is being used. You can get a lot of that information from the App Engine dashboard. But frequently, you'll find there is some application-specific data you need to examine.

For example, if you're running a web store, you may want a daily summary of how many purchases customers made and what your profits on those purchases were. The App Engine dashboard can't tell you that. It can tell you how many visitors came to your store, how long they spent there, how much CPU they used, how many datastore objects were modified, and how much bandwidth was used working with them. But it can't tell you how much of your inventory was depleted or how much money you made.

You could just provide a URL to check these details; you could go in and take a look at your receipts for a given day using the right URL for a request. But if you're running a business, you may want that report every day. Instead of remembering to go to the site, request the report, and then save it, how much better would it be to have App Engine automatically generate the report and send it to you every day?

We haven't actually implemented anything like a web store; that example would be a bit too complicated for the scope of this book. However, the process of scheduling tasks like report generation isn't all that different for a web store than it is for a chat service. In fact, it's a pretty common need: the point of scheduled tasks is to look at all of the data your application generated during some segment of time and analyze it. The purpose of the analysis varies—a web store might want to generate reports for the store owner, or a chat system might want to decide whether to terminate a chat room that hasn't had any activity for several days. But the basic goals are the same: set up a schedule, analyze

data at times dictated by that schedule, and do something useful with the results of that analysis.

We'll use report generation as a canonical example and set one up for our chat service: we'll generate a report of how many messages per day were sent in each chat room.

The Cron Scheduler

Our application has a file for a schedule. In Java, that file is named WEB-INF/cron.xml; in Python, it's cron.yaml. Since we're using the Java version of our chat app, we'll put an entry into the WEB-INF/cron.xml file. That file contains a list of entries wrapped in an <*cronentries*> tag. Each entry is a <*cron*> element.

The cron entry contains three fields, represented as XML sub-elements:

<*url*>
: First, it has a URL. As we've seen over and over in App Engine, it basically wraps everything in HTTP requests—pretty much everything is implemented using HTTP request handlers. Cron is no exception. App Engine will generate a **GET** request for this URL according to the schedule.

<*description*>
: Next, there's a description. This is just documentation; it has no execution-time effect. It's there to let someone reading the cron file understand what the task is about.

<*schedule*>
: A textual description of when the task should be invoked.

The schedule is written in what looks like an English description:

every *number units*
: Run at a specified interval. For example, every 2 hours, or every 3 minutes.

every *time*
: Run at a specified time. For example, every monday 12:00 will run at noon on Mondays; or every day 00:00 will run every day at midnight.

nth *time* of *month*
: To run your report less often, you can specify relative to months: 2nd tuesday (leaving out the month to mean *every month*), or 3rd friday of march,june,september,december to run the report four times a year.

For example, to make the chat room generate a report every day, put the following into the cron file:

Download ws2/ReportingChat/war/WEB-INF/cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
        <cron>
           <url>/report?to=markcc@gmail.com</url>
           <description>Generate a daily chat usage report.</description>
           <schedule>every day 00:00</schedule>
        </cron>
</cronentries>
```

The report will be generated by sending a **GET** to the /reportURL, and it will be done every day at midnight.

Implementing a Cron Request Handler

The basic code for a request handler is straightforward. We've been writing datastore queries over chat messages for a while—for this one, we just need to do a query that returns all chat messages added within the last twenty-four hours. Then we'll check the length, and that's the result.

Strictly speaking, this is overkill. We don't need to retrieve all of the messages. But in most analytic/reporting scheduled tasks, we'll want to skim all the data from the latest period, so this is the basic structure that you'll use in these sorts of tasks: retrieve everything and then skim over it, generating a result.

Download ws2/ReportingChat/src/com/pragprog/aebook/chat/server/Reporter.java

```
@SuppressWarnings("serial")
public class Reporter extends HttpServlet {

        Logger logger = Logger.getLogger(Reporter.class.getName());

        @Override
        @SuppressWarnings("unchecked")
        protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
                String toAddress = req.getParameter("to");
❶               PersistenceManager persister = Persister.getPersistenceManager();
                Query query = persister.newQuery(PChatMessage.class);
                query.setFilter("date >= yesterday");
                query.declareParameters("long yesterday");
                query.setOrdering("date");
                long yesterday = System.currentTimeMillis() - (24 * 60 * 60 * 1000);
                List<PChatMessage> messages =
```

```
                    (List<PChatMessage>)query.execute(yesterday);
                resp.setContentType("text/html");

❷               PrintWriter out = new PrintWriter(new CharArrayWriter());
                out.println("<html>");
                out.println("  <head>");
                out.println("    <title>Chat Usage Report</title>");
                out.println("  </head>");
                out.println("  <body>");
                out.println("  <h1>Chat Usage Report</h1>");
                out.println("  <p> Messages in the last 24 hours: " +
                messages.size());
                out.println("</body></html>");
                out.close();
                String report = out.toString();

                Session mailSession =
❸                   Session.getDefaultInstance(new Properties(), null);
             try {
               Message msg = new MimeMessage(mailSession);
               msg.setFrom(new InternetAddress(toAddress));
               msg.addRecipient(Message.RecipientType.TO,
                   new InternetAddress(req.getParameter("to")));
               msg.setSubject("Chat Status Report");
               msg.setText(report);
               Transport.send(msg);
            } catch (AddressException e) {
                   // Email address is constant and valid, so this
                   // can't happen.
            } catch (MessagingException e) {
                   logger.log(Level.INFO, "Error sending report: " + e);
            }
        }
}
```

This code is a straightforward servlet—by now, you should be able to read it without any help. It's just a combination of stuff we've seen before:

❶ First, we do the usual persistence to assemble a query and retrieve the chat messages from the datastore, just like the code was in Section 10.1, *Data Persistence in Java*, on page 141.

❷ Then we use that information to generate a string containing the report.

❸ Finally, using the App Engine mail service that we saw in Section 15.4, *Sending Mail*, on page 243, we send the message.

There's just one more thing we need to do. We told the scheduler to generate a request on the /reportURL for our application. We need to tell App Engine to connect this servlet to that URL. We do that as usual—by adding an entry to the <*servlet*> element of war/WEB-INF/web.xml to register the servlet and then adding a <*servlet-mapping*> entry to bind the servlet to the URL.

```
<servlet>
   <servlet-name>ChatServletReporter</servlet-name>
    <servlet-class>com.pragprog.aebook.chat.server.Reporter</servlet-class>
    <auth-constraint>
          <role-name>admin</role-name>
        </auth-constraint>
  </servlet>
   ...
  <servlet-mapping>
        <servlet-name>ChatServletReporter</servlet-name>
    <url-pattern>/report</url-pattern>
  </servlet-mapping>
```

16.2 Running Jobs Dynamically Using the Task Queue

The second method of performing computing on a server is more flexible and interesting than the cron scheduler. However, the basic elements are handled in a similar way: each piece of work that you want to do outside of a user request is set up as a handler for HTTP requests on a particular virtual URL.

With the cron scheduler, we specified that a particular request was supposed to be invoked according to a specific schedule. But there are a lot of tasks that we don't necessary want to do on a schedule. For example, on my blog, I regularly check statistics about how many hits it received and where the hits were referred from; if there's anything interesting in that data, I send a copy of the report to my email address. There's no specific schedule for it—I ask for a report whenever I notice something that I'd like to keep a reference to.

App Engine provides a service called *task queues* for handling things like this. Task queues are a very general mechanism for doing any kind of work that is ultimately initiated by a user request—it doesn't have to be work that needs to be done immediately. For example, if I wanted to build an online calendar, I could have the calendar application add something to the task queue to email a reminder; the task would be put into the queue with a specification of the time when it was supposed to be run. That task would be enqueued when the user created a calendar

entry; the actual task wouldn't run until its activation conditions were met. Task queues are an amazingly powerful mechanism because tasks run from the task queue can, themselves, add other tasks to the queue! In this way, task queues become a fully general workflow system for building arbitrary task sequences and flows.

The basic idea of a task queue is very simple. Your application can specify a set of named pools of work that need to be done, called a task queue. Each unit of work is described by a request. The App Engine servers will periodically check the task queues, and if there are any requests waiting in a queue, the server triggers them and executes the handlers. In general, when you put something into a task queue, App Engine runs it as soon as possible, but there are no guarantees about how long it will take.

Tasks

So what's a task? Conceptually, it's a small unit of work. It's basically a function in a language, such as Python: a chunk of code that takes some parameters and input and performs a computation. The main difference between invoking a task from an App Engine program versus invoking a function/method is that tasks are *asynchronous*: like the GWT methods we used for building Java UIs, the caller essentially says, "I want this task to be done," and then it goes off to do something else. It doesn't wait for a result. If the caller needs something to be done with the result, it must provide a callback. In a regular programming language like Java, we described a callback using an object. In the world of tasks, a callback isn't an object; it's another task. If that sound confusing, don't worry—let's take a look at some concrete examples.

Maybe we're building a collaborative calendar application. Any user can schedule a meeting, sending invitations to other people. Those people can respond, accepting or declining the invitation. When the meeting happens, the application sends reminders to the expected attendees.

When we create a calendar meeting, we want to email invitations to people, and we do that by creating a task for each invitation. We don't want to send our invitations while the user is waiting for a response to her request: sending email is slow! If we have a dozen invitees, sending a dozen emails leaves the user hanging for a long time. So we just create a bunch of tasks and throw them onto the queue: we know they'll get done, but we don't need to wait for them.

That's the basic idea: your App Engine code is always written as a bunch of HTTP request handlers/servlets. If you want to do computation on the server, you create something that tells the server how to generate the request, and you're in business. There's nothing special about how you write task implementations—they're just event handlers. In fact, you can create events that send requests to the same URLs used by human users. There is really no difference between a task implementation and any other handler. The reporting handler that we wrote for cron tasks works perfectly as a task implementation. We could also write tasks that submit chat messages, exactly the way a user would, simply by enqueuing a task that performs a request on the submit-chat URL with the right parameters.

Creating Tasks

What do the tasks look like from your code? As we said, they're simple objects that tell the server how to create an HTTP request.

So if we want to send an email (as we did in the previous section in order to send a daily report on chat room usage), we define a task that has one parameter—the email address that the report should be sent to. That's it. Since the URL for our report generation was /report, that will be the URL of the request. We implement the report generator to handle a GET request: so when we build the task, we make it a GET. We don't need to implement the handler, because we already did. The cron job generates a completely normal GET, so our task can use the same handler—it just needs to generate the same basic GET.

If we want users to be able to request a usage report in our chat service, we add a button to our user interface that generates a GWT remote procedure call. The call invokes the generateReport method:

Download ws2/ReportingChat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@Override
public void generateReport(String address) {
❶       TaskOptions task = method(Method.GET).param("to", address);
❷       QueueFactory.getDefaultQueue().add(task);
}
```

❶ There's something odd about this code. In App Engine, we submit a task by creating a TaskOptions object describing the task. But we don't create one explicitly. Instead, we call a static method on the class com.google.appengine.api.labs.taskqueue.TaskOptions.Builder. To create a GET task, we call the method called method in the pack-

age com.google.appengine.api.labs.taskqueue.TaskOptions.Builder. But that's a lot to type. In general, in task queue code, we'll just do a static import: by putting

```
import static com.google.appengine.api.labs.taskqueue.TaskOptions.Builder.*;
```

in the imports section of our code, the static methods become available. We just say method(Method.GET) to create a GET task. That creates the TaskOptions object. Then we set its parameters using methods on the options object.

❷ Submitting the task for execution is also easy: we just get the queue using a method on QueueFactory—either getDefaultQueue() for the default or getQueue(name) for other task queues—and then add the task options object to it.

We can configure the following task options:

Task name.
When we create a task for a task queue, we can assign it a name. That name will appear in errors, logs, and task monitors in the GAE control panel for your application.

Request type.
An HTTP request can, as usual, be a GET, a PUT, or a POST request. We set the HTTP request type using the method() method.

Request URL.
Instead of directly specifying the code that the task should run, we just specify a URL. App Engine sends the request to that URL, and whatever handler was set up to process requests for that URL is invoked. We set the request URL using the url() method.

CGI parameters.
CGI parameters are parameters that will be encoded into the URL of the request. In general, use CGI parameters for short, simple values that don't include punctuation or spaces. In our example, we used it for the event identifier, which should be something simple like an integer. We can specify as many CGI parameters as we want; for each one, invoke the param() method to add it to the task.

Header parameters.
These are parameters that are specified on HTTP request header lines. Header parameters are single-line string parameters, and they can have spaces, punctuation, and whatever else we want. Like the CGI parameters, we can specify as many headers as we want. Add them to a task by calling the header() method.

Message body.
: Since we're dealing with an HTTP request, if the request is a PUT or POST, there's a request body into which we can put any content we want. We can set the body using the payload() method.

Estimated execution time.
: We can tell App Engine that we don't want the task to execute until a particular time by setting the task's eta. The App Engine server keeps the task in the queue and waits until after the eta arrives. The server executes the task as soon as possible *after* the eta.

From the handler side, tasks are even easier—as I've said throughout this section, they're just standard event handlers. But sometimes, you might need to be able to tell whether a given request was generated by a human or an App Engine event. For example, if we had a Feedback box in our chat application, when the user submitted feedback, it would make sense to process it using the same handler as the send invitation shown above. When a human presses the Submit button, he's going to expect some sort of feedback—a message like, "Your feedback has been mailed to the system administrator." But if the event was invoked as a server computation task, anything included in the reply is going to be discarded, so why go to the trouble of generating it?

You can tell when a request was generated by a task by looking at the message headers. Every task queue–generated request has three headers, describing the task that invoked it:

X-App Engine-QueueName
: The name of the queue where the task was created. As we'll see in the next section, we can have multiple task queues, each with its own configuration.

X-App Engine-TaskName
: A name that identifies the task.

X-Appengine-TaskRetryCount
: When a task queue invokes a task, it waits ten minutes for the task to complete. If it doesn't complete in that time, or if the request returns an error code, the system will retry. This field is normally 0, but if the task has failed before and the current execution is a retry, this field specifies how many times the task failed. (The timeout period has varied pretty dramatically over the course of writing this book, ranging from thirty seconds to the current ten minutes, so if it's really important to you, you should check the current GAE task queue documentation to check what the current timeout period is.)

Using Multiple Task Queues

As I mentioned in the previous section, you can have more than one task queue. Since a single queue handles as many tasks as you want and each task specifies its own target URL and parameters, you may be wondering why you'd need more than one queue.

Most of the time, you probably won't. For many applications, you'll be able to do all of your server computing with a single task queue. But there are some things that you can only configure in terms of the queue, not in terms of the individual tasks. For example, keep in mind that you have to pay to execute tasks. If you've got a task that's relatively expensive—sending email messages, for example—you might want to limit the number of tasks the queue executes. You could, for instance, configure a queue to process only one event per second. That way, you limit how much you'll end up paying for email service by limiting how many emails your system can possibly send.

Dealing with concurrency is another situation in which you might want more than one queue. In general, App Engine runs tasks from a queue in order—that is, the first task inserted into a queue is the first task executed. If you have a hundred things in your queue and you add another task, that task won't be executed until the hundred tasks ahead of it have been executed. If you've got a high priority task that needs to be executed right away, you simply can't do it with one queue. Instead, you'll want a separate, high-priority queue. Normal tasks are sent to the default queue; tasks sent to the higher priority queue will execute without waiting for all of the tasks in the default queue.

Every App Engine application has a default queue. If you just want to use the default queue, you don't need to do anything special. But if you want more than one queue, create them by adding a queue.xml file to your application. For example, we can use queue.xml to make a default queue and a high priority queue:

Download ws2/ReportingChat/war/WEB-INF/queue.xml

```
<queue-entries>
  <queue>
    <name>default</name>
    <rate>5/s</rate>
  </queue>
  <queue>
    <name>priority-queue</name>
    <rate>10/s</rate>
  </queue>
</queue-entries>
```

Every queue has the following properties that you can define:

name
: The name of the task queue.

rate
: The maximum rate at which the queue will process tasks. Specified using a number per second (10/s for ten per second; 5/m for five per minute; and 1000/d for one thousand per day).

total-storage-limit
: The total amount of storage that can be used by tasks waiting in the queue. For example, 10m for 10 megabytes. This isn't the amount of space that the handlers for the tasks can consume; that's determined by your App Engine settings. This is just the amount of information that can be in the queue as part of the task descriptions.

16.3 Wrapping Up Server Computing

In this chapter, we looked at how to run tasks on the App Engine server outside of a user request. It turns out it's simple: you just provide a set of HTTP request handlers/servlets—exactly like you'd provide for any other HTTP request. Then you can run new jobs simply by putting an entry into a task queue that tells the server how to generate a request.

One thing that we haven't discussed is how to manage who can send a request on which URL. With task queues, you've got URLs that are only there to provide an entry point for a server computing task. You don't want users to be able to send requests to those URLs: it's a major security hole, both in the sense of providing people with a mechanism to get access to data that they really shouldn't have access to and in the sense of denial-of-service style attacks. This is particularly problematic in App Engine, because you're paying for the resources that you use. If someone is able to engage in an attack that uses unauthorized access and causes your application to use lots of CPU time, you're going to be stuck with a really big bill!

In the next chapter, we'll look at security issues in detail. We'll look at what security really means and how to build a secure system using App Engine.

Chapter 17

Security in App Engine Services

In this chapter, we're going to take a look at security, something we haven't really touched on yet. Security is a big topic—it would take more than this entire book to cover the subject completely. But let's take a quick look at the basics: what security means and how to implement basic security facilities inside of App Engine.

17.1 What Is Security?

The fundamental concept of security is simple: every application works with some set of data objects. Security is a policy that creates the set of rules that define who is allowed to view or alter those data objects and in what ways. A secure system is a system in which no data object can be viewed or altered by a user without that permission being specifically granted by the security policy.

For example, in our chat room application, we allow anyone to access any chat that they want—but only after they've logged in. So we've got a very weak security policy. There are three operations in chat: view the chat list, read a chat, and post a chat message. Any user can perform these operations as long as they've logged in.

For many applications, that's not even close to adequate. For example, imagine a shopping application. Users should only be allowed to see and edit their own shopping cart. Anyone should be able to view the catalog, but only store employees should be able to edit the catalog. Most employees should be able to look at carts to help shoppers, but they shouldn't be able to change them. And only an owner or manager should be able to look at the overall financial state of the business.

But as you'll notice from the shopping application example, we can still express the necessary security policy in terms of the sets of data that are manipulated by the object and by the operations that can be performed on that data.

17.2 Basic Security

At its most basic, security consists of two parts: defining a security policy and using authentication and checks to implement code that enforces that policy. The policy defines exactly who is allowed to do what; authentication gives us a way of identifying users so we can grant them access to the things they're allowed to do, and checks allow us to refuse them access to the things they're not allowed to do.

In this section, we'll take a look at how to define a basic security policy and how to implement that policy using App Engine facilities by working through an example: we'll add an administration facility to our chat application and define a security policy controlling who is allowed to use it.

Adding Administration to Chat

For a simple example of basic security, let's go back to our chat application. In our original chat application, the set of available chats was fixed. When the application ran for the first time, it automatically initialized a small list of chats, and that was all the chats that were ever allowed. In order to change that list, we would need to change the source code of the chat application.

Instead, we would like to be able to create new chats on the fly. But we don't want any user to be able to just create chats willy-nilly; we'd like some control over who can and who can't.

So what we'll do is define a policy, which describes the set of basic resources and operations that make sense for our application. Then we'll look at how to implement those policies in App Engine.

In our security policy, there are three kinds of objects: chat rooms, chat messages, and users. We'll define our security policy around just two of them: users and chat rooms. Individual chat messages will be controlled by the chat rooms they belong to. The actual security will be provided by controlling the rooms.

We do need to be careful here. One of the biggest mistakes developers frequently make is creating a security policy in which the policy is defined in terms of a high-level compound object, like our chat room, but then they provide some other way of accessing the objects through an alternative interface, which isn't defined in terms of the security policy objects. This alternate interface then ends up providing mechanism that can be used—either accidentally or as a part of a deliberate attack—to violate the security policy.

For example, in our chat interface, we sketched out a time-based view of chats. We could have extended that into a complete time-based interface built around a view that showed all messages posted in the last thirty minutes. So suppose that we did that—suppose we provided a view that just retrieved all of the messages by timestamp. If we had a room-based security policy that only checked users' privileges when they tried to access a chat room, then the timed view could completely invalidate the security policy; users could view the contents of any chat room—even one that they weren't allowed to see according to the security policy—by looking at the timed view.

That's what makes security so tricky: you really do need to consider everything very carefully. It's all too easy to make a tiny, trivial mistake in some obscure corner of your system, and boom, everything collapses; your system becomes completely insecure.

There are tons of examples of this in the real world. Most software viruses are, really, exploits that take advantage of exactly this kind of hole. There's a security system in place to prevent programs from doing certain kinds of things, but there's some little corner of the system where they didn't consider how it would interact with the security policy.

As you can see, it's important to make sure that you've designed your security policy carefully and completely. In fact, the approach that you're seeing in this book isn't actually the process you'd follow in a real application. What we're doing is retrofitting security onto an existing application; in practice, you must consider security from the start: security policy should be one of the very first things that you design.

For our chat app, we'll do our best to design a policy as if it were being designed for a new application. We'll look our Java application, since it's the simpler of our chat programs. We need to think about just what views exist in our application, design a policy that considers every pos-

Buffer Overflows

One interesting example of a security gap that has been widely exploiting is something called a buffer overrun. You don't need to worry about this kind of error in App Engine, but it's interesting as a case study.

In a lot of C code, data is read into a buffer—a chunk of pre-allocated memory. For example, if you had a form with an entry box, a lot of C frameworks would allow you to pass a pointer to a buffer, and the content of the entry box would be written into that buffer.

The size of the buffer is fixed; it's allocated before the buffer is passed. For example, if you had an entry box that could hold an 80-character string, you'd pass a buffer with 80 bytes of memory.

But what happens if someone returns 81 characters instead of 80? Or 800?

That's exactly what happens in a buffer overrun. An attacker stuffs much more data into a buffer than there is space in the buffer. If your code doesn't check that length and just copies the data, you'll end up copying data that overwrites your program's memory, changes data structures, and even possibly inserts new code into memory that will get executed!

How is this a security hole in the sense we've been talking about? There's an important piece of security policy that underlies it. On a very low level of security policy—the level of security provided by the basic programming language and libraries—the policy is *supposed* to be that no one can write into memory that he or she doesn't own. But sloppy implementations aren't verifying that the policy is being followed. The policy requires that every write into memory checks that the write is permissible. But the actual check done in the code is, "Is the address of the start of this buffer a valid location for this call to use?" The policy is broken because the check only looks at half of the actual policy condition; the correct check is, "Can I copy this entire string of data into this buffer?" That check consists of two parts: "Is this a valid buffer address?" and "Can the data that I'm copying fit into this buffer?"

sible way users can access data in our application, and make absolutely certain that every view is implemented to respect and enforce the security policy. We'll try to keep it simple and create three roles:

- Normal users, who can read and post to chats;
- Privileged users, who can do anything normal users can and who can also create new chat rooms; and
- Administrators, who do anything privileged users can, and who can also create new privileged users.

Implementing the Chat Roles

App Engine has a notion of a role associated with an application. By default, every application has two roles: admin and normal, logged-in user. For these built-in roles, it's super easy to manage authentication. In the application config file, you can mark certain URLs as administrator only. Back in Section 5.2, *The Users Service*, on page 66, we saw how to use the users service to configure things so that a page requires an authenticated logged-in user by putting a `login: admin` into the `app.yaml` file instead of `login: required`.

For more kinds of roles, you'll need to build it yourself. Basically, you'll need to create a persistent object containing the role data and then have your request handler retrieve role data and verify permissions before generating the page. That's what we'll do here for our chat application.

Normal users are basically anyone who can log in to Google. So all we need to do for normal user pages is the required login that we already set up. Privileged users and administrators need extra work. Both privileged users and administrators have access to operations that regular users shouldn't. So we'll add two new pages to our application to represent the two new views: one to allow privileged users to add new chats and one for administrators to manage users. I'm not going to go through the implementation of the full UIs for those pages: it's just more of the same web UI building that we've been doing before. We'll just put in placeholders for those. The important thing here is to understand just how to protect those pages to make sure that they can only be accessed in ways permitted by the security policy.

According to our security policy, regular chat view pages in our application are accessible to *any* logged-in user. So that part of our policy is already implemented: we've used the App Engine users service to require logins. No one is going to be able to access the views without being logged in.

To be able to implement the security policy, we need to be able to have a mechanism for recognizing which users have roles other than the default logged-in user role. To do that, we add a new type to the datastore: a UserRole type. The role type is just a username (the name of the logged-in user as returned by the login service), and a string containing the user's role. A basic implementation of the type and a function to retrieve a user's role is shown below:

Download secure-chat/tchat.py

```
class UserRole(db.Model):
    name = db.StringProperty(required=True)
    role = db.StringProperty(choices=["User", "admin", "privileged"],
                             default="User")

    @staticmethod
    def GetUserRole(name):
       user_record = db.GqlQuery("SELECT * from UserRole WHERE " +
                                 "name = :1",
                                 name)).get()
       if user_record != None:
         return user.role
       else:
            return "User"
```

This is very straightforward code: the class is a standard datastore type with two string fields. By annotating the role field with a specific list of possible values, only those roles will be permitted. It will be impossible for anyone, either through a mistake or through a deliberate attack to create a user with an invalid role. And any user who isn't in the datastore with a UserRole record will fall through the check and will therefore end up with the default role User.

That defaulting rule is enforced by GetUserRole, which returns the role for a specific user. If a user wasn't granted a specific role, then they won't have an entry in the UserRole table, and so the GQL query will end up raising an IndexError. When that happens, GetUserRole will end up running the exception-handling code, which returns the default role, User.

The one interesting thing about this code is the use of limited values for the role property. Instead of making role be a plain unrestricted string property, we specifically limited it to make sure that the user has a valid role; there is no possible way that GetUserRole will return anything other than the three valid roles recognized by the application. It might not seem important: but when you're doing security, it is absolutely

crucial that you be supremely paranoid about everything. If there are only three possible roles, then you make absolutely sure that there are only three possible role values that will ever show up in your code—and even so, you still check.

Next, when a role-restricted page is requested, we'll need to be able to check if the user has a correct role for the page:

Download secure-chat/tchat.py

```
def ValidateUserRole(actual, required):
    if required == "admin":                                    ❶
        return actual == "admin"
    elif required == "privileged":                             ❷
        return (actual == "admin" || actual == "privileged")
    elif required == "User":
            return True
        else:                                                  ❸
        return False
```

Again, the code is pretty straightforward. But as usual for security code, it's written to be absolutely pedantic. For an Admin page, we will only display the page if the user is admin; for a Privileged page, we'll only display the page if the user is either privileged or admin. And even though the third role, User, doesn't really need a check, we put it in anyway! Why be so pedantic? Suppose that in the future we added a superadministrator role and forgot to update this function. If we just fell through and returned True without testing for User, then a request to validate for a page that requires the new superadministrator role would end up falling into the same default case as User—allowing everyone to access it! We check everything specifically, so that we can be sure there isn't someone trying to cheat by injecting a nonstandard role.

For the rest of the application, we need to consider what kinds of UI elements you need to provide the capabilities required by the security policy. The policy says that privileged users are allowed to create new chats. Our application currently has no way of doing that, so we need to create a new page that provides that capability. An implementation of that capability is shown in code/secure-chat/new-chat.html.

The basic request handler is more of the typical App Engine code that we've gotten used to. It's a standard handler, except for the role checks at the beginning. Other than that, it's just a handler that generates a standard form, which submits a POST request.

The one thing that's different is that we need to check that the user has the necessary privileges before we show them that page. The way we do that is by using the code shown below:

Download secure-chat/tchat.py

```
class NewChatRoomHandler(webapp.RequestHandler):
❶   @login_required
    def get(self):
      user = users.get_current_user()
❷     role = GetUserRole(user)
      if not ValidateRole(role, "privileged"):
          self.response.headers["Context-Type"] = "text/html"
          self.response.out.write(
              "<html><head>\n" +
              "<title>Insufficient Privileges</title>\n" +
              "</head>\n" +
              "<body><h1>Insufficient Privileges</h1>\n" +
              "<p> I'm sorry but you aren't allowed to " +
              "access this page</p>\n" +
              "</body></html>\n")
      else:
        self.response.headers["Content-Type"] = "text/html"
        template_values = {
          'title': "MarkCC's AppEngine Chat Room",
          }
        path = os.path.join(os.path.dirname(__file__), 'new-chat.html')
        page = template.render(path, template_values)
        self.response.out.write(page)
```

The role check is shown at ❷. Since we built the role code using a couple of infrastructure methods, it's not terribly difficult to check roles. We just call GetUserRole to retrieve the role of the user, and then we call ValidateRole to verify that the user has the correct privileges to access the page. We also used a really nice shorthand. Instead of explicitly checking that the user really is logged in, we used a *decorator* at ❶. Decorators are a standard feature of Python (and Java, where they're called annotations) that allow you to attach metadata to code. In this case, this decorator (provided by App Engine) tells the webapp framework that this method should only be invoked by the handler if the user is already logged in. If they're not, it automatically redirects them to the login page.

Naively, that might seem like enough. But in fact, to build a correct system, it's not. In terms of the UI, you can't get to the administration page without privileges, and you can't create a new chat room without having the necessary role, so you should be secure.

But someone who wants to harm your system isn't necessarily going to respect your UI. To make changes, your application will submit a POST request to the server. A clever attacker could observe a privileged user, see the URL used to POST an administrator's privilege change, and then directly submit a POST request to that URL. So both the GET handler for the page and the POST handler for the update need to be protected by a verification process that ensures the request is being submitted by a user with the appropriate role.

So the POST handler needs to check privileges in exactly the same way as the GET handler.

Download secure-chat/tchat.py

```
class NewChatRoomPostHandler(webapp.RequestHandler):
  @login_required
  def post(self):
    user = users.get_current_user()
    role = GetUserRole(user)
    if not ValidateRole(role, "privileged"):
        self.response.headers["Context-Type"] = "text/html"
        self.response.out.write(
            "<html><head><title>Insufficient Privileges</title></head>\n" +
            "<body><h1>Insufficient Privileges</h1>\n" +
            "<p> I'm sorry but you aren't allowed to access this page</p>\n" +
            "</body></html>\n")
    else:
        newchat = cgi.escape(self.request.get("newchat"))
        CreateChat(user, newchat)
        self.response.out.write(
            "<html><head><title>Chat Room Created</title></head>\n" +
            "<body><h1>Chat Room Created</h1>\n" +
            "<p> New chat room %s created.</p>\n"
            "</body></html>\n" % newchat)
```

Similarly to how we create chat rooms, we'll create new privileged users by using a form with a privilege check. To become a privileged user, someone with the administrator role must grant you the privileged role. So just like new-chat, we create a GET handler which generates a form; and a POST handler which actually does the work of creating the privileged user.

So that's the basics of security: define a policy and then build the system to enforce it. The basic idea really isn't complicated at all: you define what objects are accessible in your system, who is allowed to view/modify them, and in what ways. That's your policy. Then you implement checks to ensure that the policy is followed. Alas, the reality is that implementing a secure system is incredibly difficult. You need

to be aware of the implications of every operation you make available to your users, either directly or indirectly, and you need to make sure that your implementation strictly enforces the policy in every case.

So far, what we've mostly been talking about is simple security for a cooperative system. What I mean by that is that we've mainly focused on building a system where we're mostly assuming that users are cooperative and aren't actively trying to destroy or disable our system. It's hard enough to do that, but that's not really enough, because there are people out there who will try to destroy your system for fun. So you need to consider something more complicated: you need to protect against attacks. In the rest of this chapter, we'll take a look at the basic kinds of attacks that malicious users can use to try to disable your system and at the basic mechanisms that you can use to protect your system from those attacks.

17.3 Advanced Security

A more complicated aspect of security is defending against *attacks*. An attack is any attempt by someone to access or alter data without explicit permission from the security policy or an attempt to prevent someone else from doing something that the security policy says that they should be permitted to do.

Attacks can be remarkably subtle and imaginative. There's no way that we can cover all of the ways your service could be attacked. We'll just walk through a few examples that are typical of basic, general techniques that attackers could use.

In the broadest sense, you can characterize attacks in terms of what they're trying to do:

Direct attacks.
: These attacks trick the system into doing things that shouldn't be allowed. Basically, instead of trying to do some kind of subtle trickery based on the network infrastructure, this kind of attack is based on finding a flaw in your security policy and exploiting it. We'll look at this kind of attack a bit more in Section 17.3, *Direct Attacks*, on page 271.

Cross-site scripting attacks.
: The cross-site scripting attack (XSS) is both common and very dangerous. In an XSS, the attacker includes some HTML and/or

JavaScript in their request. If you allow a user to submit things that will appear directly in a reply page, without any attempt at validation, attackers can trick your application into doing almost anything they want. Our chat application is *very* susceptible to this: we don't check the contents of a chat message. An attacker could insert HTML script tags into a chat message, which, in turn, could allow the attacker to trick another user's browser into performing requests. The attacker could thereby pose as any other user! We'll look at how to protect against XSS attacks in Section 17.3, *Cross-Site Scripting*, on the following page.

Eavesdropping attacks.

These are attacks, including some common varieties like "man in the middle" attacks, which try to interpose themselves between an authenticated user and the system in order to see data that the user has permission to access but the attacker doesn't. This kind of attack can also be used to steal credentials—that is, to find out information the attacker can use to pretend to be a valid user. This kind of attack is very difficult to protect against on your own. Cryptographic protocols have been designed for helping with this: in one of the coming sections, we'll look at how to use SSL, the secure sockets layer, to implement a defense against eavesdropping. SSL isn't a complete defense—but when used carefully as part of a well-designed, thorough security policy, it's a good start.

Denial-of-service attacks.

A denial-of-service attack isn't really an attack in quite the same sense as the others we've discussed. It doesn't try to access or alter data without permission. What it tries to do is prevent authorized users from accessing the data that they should have access to. The most common form of denial-of-service attack is to have thousands of virus-infected machines send a huge volume of spurious requests to an application. The application will be forced to spend so much time processing those spurious requests that it will be effectively impossible for a valid user to actually access anything.

Be particularly careful to defend against DoS attacks against your App Engine code: you pay for the CPU time you use processing requests. Every millisecond of time that you spend on an invalid request can be expensive: an extra millisecond of CPU time per request can easily consume an extra hour of CPU time in a typical DoS attack. Fortunately, App Engine's servers are managed by

administrators who are on the watch for traffic patterns that could indicate a DoS attack, and they'll usually shut down the attack before it gets to your application. But you should still be vigilant: always check your incoming requests for the most basic kinds of validity as soon as possible—before you do anything else—and reject invalid ones without wasting any unnecessary resources. We'll look briefly at some tools provided by App Engine to help you cope with DoS attacks in Section 17.3, *Denial-of-Service Attacks*, on page 274.

Direct Attacks

Direct attacks are the simplest things to defend against. A direct attack is a fairly straightforward attempt to trick your system into letting a user perform some action for which they don't have an appropriate role. For instance, in our example above, we talked about how an attacker could potentially try to use the create chat dialog without permission by using the POST server directly.

This is the kind of attack that we protect against by providing a well-defined security policy and by implementing it carefully. This is the easiest kind of attack to protect against—even so, it can be remarkably difficult. As we discussed in the previous section, we need to make sure that every handler always checks the authentication of the user, checks whether the user has permission to perform a specific operation, and, most importantly, that the operation being requested is valid. The most common and successful kind of attack against most systems is a direct attack using invalid requests. If you want your system to be secure, *always* check your requests. Allowing invalid requests provides a hole that a sufficiently clever attacker can figure out how to drive a truck through. For example, in our chat room example, we must check the name of the users sending messages, make sure that they're really logged in, verify that the chat rooms that they're trying to post to really exist, and confirm that their message is properly formatted as an XML request. If any of those things aren't true, the application should not continue processing the request.

Cross-Site Scripting

Cross-site scripting is a kind of attack that's based on sloppiness on the part of a cloud application developer. Since an HTML page is made up of simple text with easily typeable markup, it's possible for the body

of a request to contain HTML markup, and then, using that markup, to provide JavaScript code.

For example, what would happen if a user were to send a chat message containing the following?

```
Hi there <script language="javascript"> print("JavaScript!");</script>
```

A user's display would show the chat message "Hi there, JavaScript!"—but the JavaScript code that prints the word "JavaScript" would execute on the user's computer.

There's also a closely related kind of cross-site scripting attack that tries to run code on a server. The most common form of this is called SQL injection. SQL injection, specifically, isn't a problem for App Engine (since you don't use SQL), but it's a good example of a general type of attack.

In a SQL injection attack, an attacker sends a string to a form that will be used as a field in a query. For example, in many systems, when users log in, they'll be asked for their usernames and passwords in a form, and then the system will perform a SQL query to retrieve the correct password. It'll do a query like: `SELECT password FROM Users WHERE name=$1`. Then a user can provide a username like `me;DROP TABLE *`. If the user's username field value is used directly in the query, that will execute: `SELECT password FROM User WHERE name = me; DROP TABLE *`. By knowing that the string was going to be passed directly into a SQL query, an attacker can create a string that will execute other, damaging SQL commands.

The way to protect against both of these attacks is clear: always clean your inputs. That is, when you get an input from a user, always take it, and put it through a cleaning process that eliminates any metacharacters. To prevent someone from injecting JavaScript into your HTML, you need to make sure that any < characters are replaced with XML literal characters like `<`. To prevent someone from injecting SQL, you need to make sure that your input doesn't contain things like quotes, query-metacharacters, or statement terminators; if it does, they need to be escaped.

App Engine provides tools that you can use to protect against this kind of XSS attack. If you're using Python, there's a function in the `cgi` module of the webapp framework. Just invoke `cgi.escape(input_string)`, and you don't need to worry about this kind of XSS. In your templates, you can also do escaping on the output side by attaching a filter; if you

append |escape to any variable in a Django template, the value of that variable will be properly escaped for HTML to prevent XSS attacks. For Java, it's a bit more complicated. If you're doing everything with GWT, you're covered. GWT takes care of all of the XSS and escaping issues in everything that it does with its RPCs. If you're not using GWT, then how to handle it depends on what libraries you're using, but pretty much every framework provides some built-in method. For example, if you use Java Server Pages (which is sort of like a Java version of the Django template system we used in Chapter 6, *Organizing Code: Separating UI and Logic*, on page 70, then there's a standard function c:out that is used to print values; if you add an attribute escapeXml="true" to the c:out call, it will automatically take care of processing the string.

Eavesdropping Attacks

Eavesdropping attacks are very subtle and difficult. They're attacks where instead of trying to do anything to your system, they hide in the background and watch. The main things you can do to protect against eavesdropping are to have a careful, thorough security policy and to use an encrypted channel for any privileged communications.

There are several ways of providing an encrypted channel. The easiest is to just let App Engine take care of it for you by telling it to force all requests for a particular URL to use SSL (the secure sockets layer, a standard Internet encryption system). To tell App Engine to use SSL for a particular URL, just add the line secure: always to your app.yaml entry for a URL.

When it comes to encryption, there's one absolutely crucial piece of advice I can give you: don't do it yourself. If you can get by with just using the secure entry in app.yaml, then do that. If, for some reason, you need something more complex, then find a widely used, well-supported library that provides encryption services. Both Python and Java have extensive encryption libraries that you can use to do encryption in App Engine. Use them! Getting encryption right is incredibly tricky. If you implement your own encryption system, you're virtually assured of getting it wrong. All it takes is one mistake: as an encryption implementer you have to get absolutely everything perfectly right, and there are so many subtleties, it's almost impossible to get everything perfectly right yourself. Even when teams of experts build encryption systems, they usually make mistakes. As someone trying to build a secure encrypted system, you need to find every possible hole in your scheme and close it; as an attacker trying to break an encrypted system, you only need

to find one hole. The attacker has a much easier job than you do! So take advantage of the tested systems that have been written by other people, carefully analyzed, and stress-tested by real-world exposure.

Denial-of-Service Attacks

As I explained earlier, denial-of-service attacks are when an attacker hammers your site with huge numbers of requests. The requests could be perfectly legitimate, or they could be invalid. In either case, the purpose of the attack is to force your web application to spend so much time processing the flood of attacker requests that it will become unusable for legitimate users.

Aside from the fact that it can make your site inaccessible through load, it can also cost you money. On App Engine, you're paying for the resources you use, up to your quota. Once your quota is exceeded, you won't run up any more bills, but your application will also be completely shut down. So you really want to prevent an attacker from being able to do this to you.

One part of defending against DoS attacks is simply to validate requests and reject invalid ones as quickly as possible. That's not going to help a lot if you're getting hit with 10,000 requests per second, but for smaller attacks, it can make a huge difference. When you receive a request, the first thing you should do is check that it's actually valid. (If you're using GWT, this is taken care of for you by the framework; GWT RPCs do automatic validation of requests.) And you should try to make sure that you do it as quickly as you can: if every request is supposed to have two header fields, then first check that both fields are there before you spend time validating the contents of those fields. If one is left out or if an extra one is appended, you can reject the request without spending any more time.

If you're faced with a real DoS attack, what you'll usually see is floods of messages coming from a very small number of IP addresses. It's usually a couple of computers that have been commandeered and forced to send requests to you. App Engine provides a really convenient way of blocking that. In your application configuration, you can specify a *blacklist*: that is, a list of individual IP addresses or ranges of IP addresses that will be blocked by the App Engine server. With DoS blacklists, App Engine blocks the requests before they ever reach your application, so they protect you from wasting your valuable resources on attackers' requests.

The DoS service is very new, and it's likely to evolve rapidly. In the current release of App Engine, you configure the DoS service by putting a file into your application configuration, which specifies the contents of the blacklist. In a Python application, you create a file named dos.yaml containing a list of blacklist entries. The entries are, unfortunately, currently completely unreadable. They're based on a standard routing notation for the IP called CIDR (classless inter-domain routing). You'll need to look up CIDR notation to be able to write it. For example, to say "all addresses starting with 192.168.0", you'd write "192.168.0.1/24".

So, for example, in Python, if you were under a DoS attack coming from the the IP network block containing 192.168.72.12, you'd put the following into dos.yaml:

```
blacklist:
- subnet: 192.168.72.12/22
```

As I said, the DoS protection service is a new addition to App Engine, so it's likely to have evolved by the time you read this. But the basic concept will be the same: you define who needs to be blocked and put that into a file in your application configuration. For the details, you'll definitely want to look in the up-to-date documentation on the App Engine website.

Now you know what you're up against. In this section, we've looked at the threats that you need to consider—the ways that malicious people on the net can attack your service and how you can protect yourself against them.

In this chapter, we've taken a very shallow look a security. Security is an important, complicated concern that cuts across all of the other aspects of building a web service. As we've seen, conceptually building a secure application or service isn't difficult, but practically it's close to impossible.

The starting point of any security system is the security policy—the rules that describe the objects manipulated by your application/service and who is allowed to access and/or manipulate each object. We've seen an example of a security policy for our chat application, defined in terms of chat rooms and users. Care in designing this policy is critical; we saw an example of how providing a view that wasn't covered by policy can completely negate the policy.

Once you have a security policy designed, you need to implement it. It's critical to do this carefully. You need to consider every aspect of

how users can interact with your system and ensure that your system carefully protects its data, always verifying that the user is allowed to perform a given action by the security policy. You need to be paranoid in doing this: make your application check and recheck the credentials/authentication of its users.

And you need to design your system with the idea that it *will* come under attack. You need to be aware of the different kinds of attacks that can be made against it, implement safeguards against those attacks, monitor your application to detect active attacks like DoS, and respond with the tools at your disposal when an attack is detected.

References and Resources

CIDR Notation ..... http://en.wikipedia.org/wiki/CIDR_notation
A Wikipedia article on the standard IP address notation used for managing IP address blocks, as used by App Engine's Denial-of-Service protection system.

OpenSSL ..... http://www.openssl.org/
A standard, widely used implementation of the secure sockets layer. The site includes both a fantastic implementation and extensive documentation on the library, the underlying SSL protocol, and how to use it correctly.

Common Gateway Interface (CGI) http://www.w3.org/CGI/
The official standard and documentation for CGI.

Understanding Denial-of-Service Attacks. . .
. . . http://www.us-cert.gov/cas/tips/ST04-015.html
An excellent article from the US Computer Emergency Readiness Team explaining the mechanics of denial-of-service attacks and how to recognize, protect against, and respond to them.

Chapter 18

Administering Your App Engine Deployment

So far, we've focused almost exclusively on how to write Google App Engine applications. That's because the hardest part of working with App Engine is actually implementing the applications. Once you've got your application working, it's really easy to administer it.

But you *do* need to administer your App Engine services and applications. It's easy, but it is a job that needs doing on an ongoing basis. In this chapter, we'll take a quick look at the tools that are available to you via the App Engine control panel. We'll see how you can do things like monitor the amount of resources you're using, how you can scan the data being stored by your application to identify problems, and how to see who is using your application from where.

18.1 Monitoring

Monitoring your application just means seeing how it's being used and what kinds of resources it's using. App Engine makes it really easy for you to see exactly what's happening with your application. If you go to http://appengine.google.com and sign in, you'll get a list of the applications that you've deployed. You can look at any of your applications by clicking on their names.

When you click on one of the applications, you'll be brought to the main App Engine control panel for that application, with a view showing you the *dashboard* for your application. The dashboard give you a concise overview of the data you probably care about. You can see an example

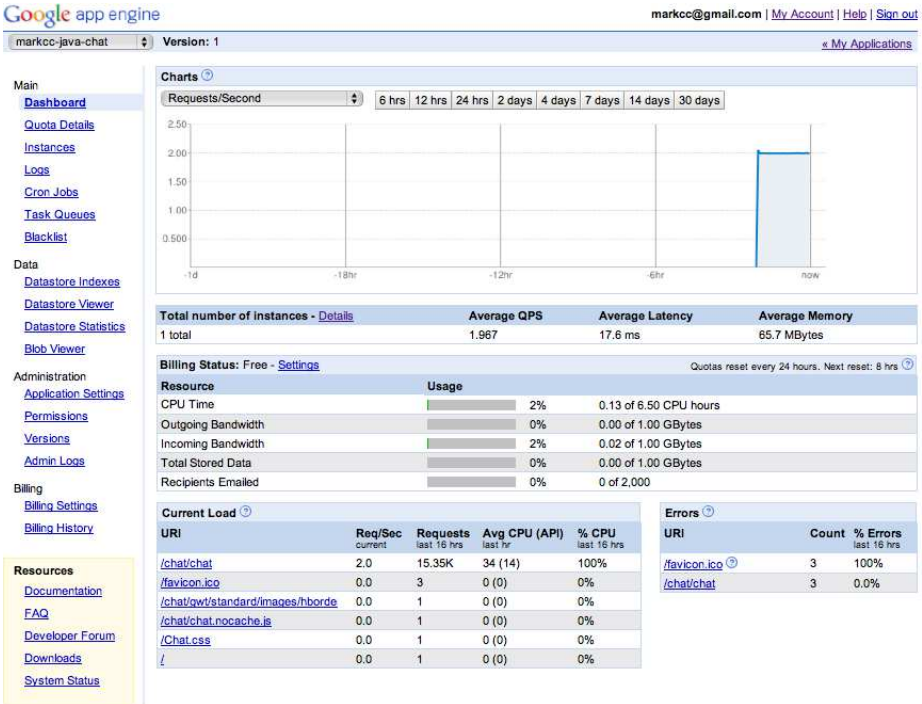


Figure 18.1: The control panel dashboard for the Java chat app

of the App Engine control panel for my Java chat application in Figure 18.1. As you can see, it's not exactly seeing a lot of use, but the information is all there.

At the top, you've got a graph showing you the average number of requests per second being handled by your application. For my Java chat, it's been seeing so little use that its average usage is zero! By really pounding on it for a while, I was able to get the usage load up a bit in the last time segment, up to about two queries per second.

The top graph can show you summaries of lots of different things. In its top left, there's a drop-down menu that lets you select which view you want. In addition to the number of requests per second, you can get a graph of the time spent to process each request, the number of errors, the bandwidth per request, the amount of CPU use per second, and the number of quota denials (that is, requests that were denied because you used up all of your App Engine resources). For example,

Figure 18.2: The CPU-time per request view

in Figure 18.2 you can see a graph of how much time it's spending per request. According to that graph, my application has been averaging around twenty-four milliseconds per request when I was using it.

Below the graph, there's a collection of slightly more detailed data:

Instances.

Each process that's running your handlers is called an instance. There's a line under the graph that tells you how many instances your application is using, how quickly it's processing queries, and how much memory it's using. My Java chat generally runs either one or two instances, determined mostly by random factors in the App Engine cloud—it's not related to load. (In general, the number of instances that you get is proportional to the load on your application. In this case, the load is so low that the threshold between one and two instances is effectively random.) My application averages about two requests per second, so when I've got one instance, it shows an average queries-per-second of around two; when I've got two instances, they each average about one query per second.

Their average latency—that is, the amount of time it really takes from when they receive a message until they can start processing it—is just seventeen milliseconds. But they're each using 65 megabytes of memory! That sounds crazy on the face of it, but remember that we're looking at a Java application, so the figure includes the whole JVM and all of the class libraries that it loaded. Java isn't exactly memory efficient. My Python chat room averages about one-fifth of that. But given the amount of resources available to me with App Engine, that's not really a big deal—it shouldn't affect your choice of which language to use.

Billing Status.

The billing status tells you whether or not you're paying for App Engine. If you are, it will show you a list of the resources you've paid for and how much of those resources you've used. By default, it just says "free," because if you're using the default resources that come with an account, they're free.

If you find that you're running out of resources in the free status, there's a link in the billing status section that allows you to purchase more resources. We'll talk about that in Section 18.5, *Paying for What You Use*, on page 285.

Resources.

This is a list of the available resources that you're allowed to use and some small graphs showing how much of each resource you've used. It lists CPU time, outgoing bandwidth, incoming bandwidth, stored data, and recipients emailed. For example, in my chat room application, which isn't getting used much, it says that today I've used 0.14 CPU hours out of my available 6.5, and I've used about 3 percent of my available bandwidth. That's mostly caused by the fact that I've left the client open, running two AJAX calls per second for the last three hours!

From this information, I can already draw some interesting conclusions. When I look at my bandwidth usage, I can see a problem. I've got one client accessing my chat room: I'm not actually chatting with anyone. But it's running a steady two queries per second and steadily consuming bandwidth. If I actually had a dozen people hanging around and chatting for a couple of hours, I could actually end up running out of bandwidth quota. So if I really wanted to use this application, I'd need to either reduce the frequency of AJAX updates, reduce the size of the update requests, or purchase more bandwidth. For this application, the best choice would probably be a combination; I don't really need updates more than once per second, so I could cut the bandwidth use in half by just reducing the AJAX update frequency. Beyond that, the request is really small, and there's not a lot that I could do to make it smaller. So I'd need to buy more bandwidth.

From the dashboard, if the default view doesn't give you enough information, you can drill down to get more details. Nearly everything contains links which allow you to get more information. For example, you can get more information about your instances by clicking on Details, next to the number of instances.

In addition, if you look on the left side of the panel, there's a list of different categories that you can use to drill down and get more precise information. You can check how you're doing on your quotas; you can get very precise information about your instances; you can check your task queues; and so on.

18.2 Peeking at the Datastore

As we've seen throughout this book, most of App Engine isn't really anything new. Wherever possible, App Engine tries to reuse standard technologies to allow you to write cloud applications. The one place where App Engine is really different, the one really novel thing in App Engine, is the datastore. Most cloud application frameworks provide you with access to some kind of lightweight relational database like MySQL. But App Engine goes a very different route, storing things in Google's Bigtables using datastore.

So one really important thing as you administer your App Engine application is to understand how your data is stored and how that storage model affects your resource usage.

In the App Engine control panel, there are a set of views for letting you peek and probe at how your application is using the datastore.

The easiest thing to see is how your data is indexed. If you just click on Datastore Indexes, it will show you a list of the data types that you're storing and which fields of them are indexed. This is most valuable early in the development process. Datastore automatically indexes some fields based on your queries. By looking at this view, you can see what indices you have and what indices you don't have.

You can also browse your way through all of the objects that you've stored. If you click on Datastore Viewer, you can see a list of all of the objects in your datastore, categorized by type. You can also find specific objects: if you click on the Options link in the datastore viewer, it shows you a GQL query that was used to generate the current view, and you can edit the query. For example, in my chat application's datastore, there are PChatMessage objects, and PChatRoom objects. I can look at all of the chat messages that were sent by me by entering a query like SELECT * FROM PChatMessage where senderName='markcc'.

Finally, you can see how much storage objects are taking— including the metadata for their indices—by checking under Datastore Statistics.

18.3 Logs and Debugging

When you're having trouble with your application, the most useful thing on the entire control panel is logs. Logs give you a way of looking into your running application in almost the way that you could with a local debugger. Every time anything goes wrong in your application, App Engine will generate a log entry for it; by viewing the logs, you can see what happened.

For example, when I open the Logs view, I can see that I made a mistake in the chat application. The template that I used for generating chat views says that there should be a favicon; but there is no favicon available. So every pageview request ends up generating an error due to the missing favicon. That doesn't stop my application from working. In fact, I didn't even know that I'd made that mistake before the first time I checked my logs! But that's exactly why logs are so valuable: in App Engine, you can't directly watch your application run the way you could if it were on your own computer. It's running somewhere else on a machine you can't access directly. But the logs provide you with a valuable tool that allows you to get as much information about your running application as you want.

Of course, App Engine isn't going to automatically tell you everything. It can't know exactly what bits of information you're going to want. So all that it does automatically is record errors that it's aware of. If your application tries to access data from the datastore that it can't find, App Engine knows about it, and it can generate an entry in your log. But if your application successfully retrieves information from the datastore, but due to an error in the query, it's not the correct information, App Engine doesn't know, and so it doesn't automatically log anything.

You can insert logging commands into your application using your language's standard logging support (like java.util.logging in Java); all of the log entries are viewable by clicking on the Logs entry in the control panel. This is your most valuable tool for debugging: you can see exactly what your application is doing by generating log entries.

For example, the version of my chat application that's deployed right now contains an error. If I give it a long chat message, it will fail to process it correctly. Why?

When I look in the default logs, I see a bunch of entries like Figure 18.3, on the following page, all complaining about not being able to process chat messages that are longer than 500 characters, because the data-

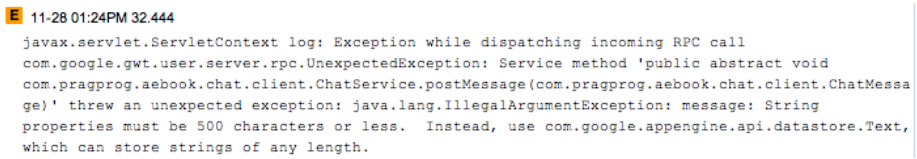


Figure 18.3: A log entry from the control panel log viewer

store model specifies the message content as a string property, and strings can't be longer than 500 bytes. But my chat messages aren't 500 characters long!

For this problem, the default log message is enough to clue me in, but suppose it wasn't. Maybe I thought that the chat messages weren't that long. I could add logging code to my application. First, I'd need to declare the logger in my source file. So I'd add the following to com.pragprog.aebook.persistchat.server.ChatSubmissionServiceImpl.java, at the very end, right before the closing brace:

```
private static java.util.logging.Logger logger =
    java.util.logging.Logger.getLogger(ChatSubmissionServiceImpl.class.getName());
```

And then, inside of the postMessage method, I could add a logging statement:

```
logger.info("Chat message size = " + message.getMessage().length() +
            " with body \"" + message.getMessage() + "\"");
```

With that statement added, every single post message will generate a log entry containing the length and content of a posted message. With that, I could confirm that the messages really were more than 500 characters.

You should, however, be a bit careful about adding logging to your application. It's an incredible tool. But logging takes CPU time, which does count against your quota. (You don't need to worry about space; App Engine will only keep a certain number of your most recent log entries, automatically discarding older ones.) So doing too much logging can, quite literally, cost you money. Not much, but it is a real consideration.

More importantly, the logs are your primary tool for understanding what's going on in your application. If you generate a ton of log entries for every trivial thing, then you're crowding your logs with irrelevant

data. That data can make it hard to find the important entries—or if there are really a lot of them, they can even crowd out the relevant ones entirely! The rule of thumb is don't log something that you don't actively want to read—don't write log entries that you know you're never going to look at. But when you're having a problem, add log statements so that you can see what's going on and then hit the logs in your control panel while you're debugging.

18.4 Managing Your Application

The control panel contains a collection of settings that help you manage your application. There are four views that you can use for managing things: Application Settings; Permissions; Versions; and Admin Logs.

The Application Settings view gives you access to a set of basic settings for your application. It includes the following:

Application Title.
: This produces the name of your application in an editable text box. To change the application name, you can just change the contents of the text box, and click Save Settings.

Configured Services.
: If your application uses any services, like email or XMPP (chat), the services will be listed here, with any options that you can configure.

Cookie Expiration.
: App Engine uses cookies for authentication of users. You can choose between setting the cookies so that users stay logged in for one day or for one week.

Disable or Delete Application.
: If you want to either temporarily or permanently shut down your application, this is the place to do it. If you click Disable Application, you'll first be taken to a confirmation dialog to ensure that you really want to shut the application down. If you confirm it, the application will be disabled, and the Application Settings view will be changed so that it contains two items—one to relaunch the application and one to permanently delete it.

Domain Setup.
: If you want to run your application on your own domain, you can tell App Engine about the domain here. Currently, you can buy

and set up a domain using Google Apps and then have your App Engine programs run at that domain. At the moment, if you own a domain that isn't run by Google Apps, you can't use it for your App Engine services.

The Permissions view allows you to give administration privileges to other users. If you grant administration privileges to another user, then they will be able to access the full control panel for your application, including the ability to change any settings, to modify the code, and to disable or even delete the application.

The Versions view shows you a list of the versions of your application that have been deployed and gives you the ability to choose any of them as the current, active version. The main use of this is in case of errors. If you deploy a new version of your application and it contains an error, you can instantly revert to any prior version by going to the Versions view and selecting the error-free prior version.

Finally, Admin Logs let you see what administrative actions have been performed on your application. This is mainly useful for letting you monitor the actions of other administrators: any action taken by any administrator will generate an entry in the admin logs.

18.5 Paying for What You Use

As I said at the beginning of this book, the basic point of cloud computing is to let you buy exactly the resources that you want to use. You don't buy an entire computer and then let its CPU sit idle for most of the time in order to ensure that you'll have enough CPU if you get slashdotted; you just buy what you need. The billing panel is where App Engine lets you tell Google how much you want to pay.

There's a Billing Settings link on your control panel. If you follow it, it gives you the option of enabling billing. By default, you're working with free resources—Google gives you your first hit for free—-but if you want more than the minimum resources, you'll need to pay for it.

With the billing controls, you can set up a daily budget. You can set a daily maximum—the most money you're willing to pay for additional resources in a given day. You can also set per-resource maximums to say things like, "I'm willing to spend up to $4 per day on CPU but only $1.50 for bandwidth."

Once you've set up your budget, you can set up a payment on your credit card using Google Checkout.

You can see exactly what resources you've been using and how much you've paid every day by looking in the billing history. There's a daily entry in the billing history in the same format as the log entries that tells you exactly how much of each resource you've consumed every day.

In this chapter, we've taken a quick look at the administration controls provided by the App Engine control panel for your application. Nothing here should be at all surprising: it's all very straightforward and easy to use.

Chapter 19

Wrapping Up

We've covered a lot of ground in the last 300 pages. At this point, you've become familiar with the basics of programming for Google App Engine. Of course, there's no way that we could cover everything in this amount of space, but now you are ready to build your first applications and work with the Google App Engine online documentation. In this chapter, we'll recap the basic concepts we've covered and give a quick overview of what else you might want to look at and where to go from here.

19.1 Cloud Concepts

As we've seen, cloud programming is very different from programming in more traditional environments. The key characteristics that make the cloud so interesting and powerful include these concepts:

- You don't know—or care—where your program is running. This is one of the most fundamental facts of the cloud. You don't run your program on a computer: your cloud service provider gives you a platform—that is, a basic software system that can run programs written in a specific way—and all of the low-level details are their problem, not yours. In general, you don't know what kind of computer it's running on or how many computers it's running on, where those computers are, or what operating systems they're running. None of that matters.
- Software is a service. In a traditional programming world, you implement applications that are executed on your customers' computers. In the cloud, you still write an application, but from the

viewpoint of your customers, it's not quite the same thing as a program because they don't run it. In the cloud, your software is a service—an application that is running somewhere else, which they access through their web browser when they want to use it. The key difference is that the users don't run the program: the program is always running. The users connect to it when they want to use it, and they don't need to worry about how to run it, whether it needs to be upgraded, whether it works on their computer, or anything like that. The service is always there, ready to be accessed by any computer from anywhere.

- There are no limits. You don't need to worry about limits and questions like, "How many page views can my computer handle?" In the cloud, everything is elastic. If you suddenly get more users, it's not a problem: your cloud platform just grabs more resources as it needs them and continues to provide services to your customers.
- You pay for what you use. In the cloud, you literally pay for what you use. You don't need to buy a big, powerful, expensive computer in case you eventually need it. If today you only need a single, rinky-dink processor and tomorrow you need something a thousand times more powerful, that's no problem. Today you'll pay a few cents for a single processor; tomorrow, you might pay a few dollars for some serious computing horsepower. Whatever you need, you pay for it as you need it, and when you're done using it, you're not stuck with a useless brick of overpowered hardware.

19.2 Google App Engine Concepts

As we've seen in this book, Google App Engine is built around a collection of basic concepts. To review, the fundamental concepts of programming for the cloud using Google App Engine include the following:

- Programming is HTTP request handling. In Google App Engine, your program's job is to handle HTTP requests. Through the magic of webhooks, every interaction that your program has with users or with other applications, is implemented as an HTTP request handler.
- You can bootstrap from standards. In Google App Engine, everything tries to build on existing standard technologies when possible. So, for example, in Python, templates are built using the

widely used Django template framework. In Java, the interface to the datastore is built in terms of the standard Java Data Objects framework. Throughout Google App Engine, we've seen that wherever possible, GAE uses existing standard technologies as their basis, rather than inventing new stuff.

- Everything is a service. All of the bits and pieces of GAE that you interact with in your program are services. Things that would be libraries to link against in a conventional program are services that you talk to through RPC requests in GAE cloud applications. And there's a wealth of services available to you, providing everything from security to data storage to instant messaging.
- The browser is the UI. Every bit of user interface, everything that your customers see, everything that your customers do, all happens inside of a web browser.
- There is no state. In a traditional program, you can count on variables to hold state—that is, if you put something into a variable, when you come back to look at it later, it will still be there. In a cloud environment, that's not necessarily true. You have to program under the assumption that the only way to preserve state is through external storage in datastore.
- Always think about security. In a cloud application, security is much trickier than in a conventional application. Since your code is running as a service that can be accessed online by malicious people, and your customers' data is all accessible by that online service, you must be extra careful that your system is built to be secure. From the very first step of designing your system, plan your security policy and make sure you enforce it in every facet of the design. All it takes is one mistake to compromise your users' data!

These basic ideas make up the heart of Google App Engine. There's not that much to it. But that's sort of the point. What makes App Engine such a beautiful, powerful system to work with is its basic simplicity. There's just this modest set of concepts and libraries built around them—and that's it. But this simple framework turns out to be a remarkable and elegant tool for building your own cloud applications.

19.3 Where to Go from Here

Google App Engine is a very new platform, and it's constantly evolving and growing. In the time it took me to write this book, many things changed enough that I had to rewrite whole sections. By the time you read this, there will probably be even more new stuff. A few exciting things that are up and coming include the following:

- Datastore options. As this book was being written, there was just one datastore implementation. Now, as I'm writing this conclusion, there is now a second datastore in testing. You'll have a choice between two different implementations with the same interface, which will give you the ability to choose an implementation that's best suited to your application. The current implementation is called the *master-slave datastore*. The master-slave datastore will still be the default version. But with master-slave, it's possible that during datacenter maintenance windows, access to your data may be interrupted. The alternative is the *high-replication datastore*. The high-replication datastore is slightly slower (by up to a factor of two) and has eventual consistency, but it can guarantee that your data is always available, even in the face of major disruptions.

- Developer storage. This is one of the really exciting developments that's currently in test deployments. Google is providing a storage service very much like Amazon S3. In fact, you can easily port programs that were written for S3 to use Google developer storage. It's also got some of its own APIs that make it very attractive as a platform. This is going to be a big deal.

- MySQL database support. If you've got an existing application that you want to migrate to the cloud or you really need/like relational databases, the GAE team has implemented a version of MySQL that scales to work inside of the App Engine cloud. With this, you'll be able to use MySQL relational databases for storage instead of the datastore.

- Go language support. A team at Google developed a beautiful new systems programming language called Go. I believe that Go will eventually be available as a language for developing programs for Google App Engine. Go is an excellent language, which combines the simplicity and conciseness of Python with the security and

type safety of Java. When this happens, in my opinion, it will be the clear language of choice for GAE development.

- Django nonrel. Currently, GAE uses the Django framework's template library for Python development. The rest of Django, and in particular its data management framework, is terrific but incompatible with how the datastore manages data. There's an effort to build a variant version of Django that will work correctly with nonrelational object stores like the datastore. Once this is ready, Django will be usable as a full-fledged alternative to webapp for GAE development.
- Channels. Currently in GAE, clients need to request data from the App Engine server. In our chat application, we saw that in the mechanism we used to keep our chat window up-to-date by sending update requests twice per second. Channels are a mechanism for push support, which provides a way that the GAE service running in the cloud servers can send updates to a client UI without any requests. With channels, we could implement our chat service without the polling for updates: the server would just push updates to the client UI using a channel.
- MapReduce. For complex computations on data in the datastore, you're currently limited to simple processing tasks using the task queue. The GAE team is working on support for Google's MapReduce framework for massively parallel computation. When this is ready, GAE will be capable of being used for implementing parallel computational workloads for things like bioinformatics processing.
- Better SSL support. Right now, GAE allows you to use SSL for secure communication only if you're building an application that runs within the appspot.com domain. Soon, you'll be able to provide GAE with SSL credentials to let you use your own SSL keys on your own domain.

And that's just a taste. There's plenty more. You should follow the App Engine blog for information about all of the changes—both the ones I mentioned above, and more.

As we've seen, Google App Engine is a tremendous system for building cloud- based applications. You now know how to use it, so go out and build some great applications!

19.4 References and Resources

The Google App Engine Blog . . . . . http://googleappengine.blogspot.com/
The official Google App Engine blog. Every App Engine developer should really follow this.

The Google App Engine Community Home. . .
. . . http://code.google.com/appengine/community.html
A Google Code clearing house for the App Engine community. This site includes forums where you can get questions answered, downloads of sample applications, discussions of upcoming features, App Engine system status information, FAQs, etc. It's a terrific resource for GAE developers.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

T

U

V

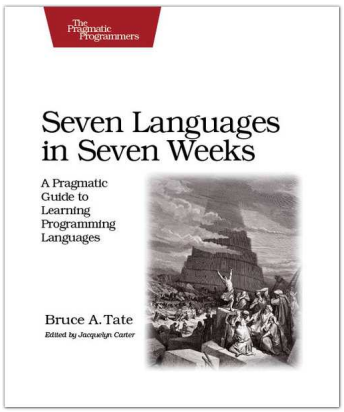
W

X

More from PragProg.com

Seven Languages in Seven Weeks

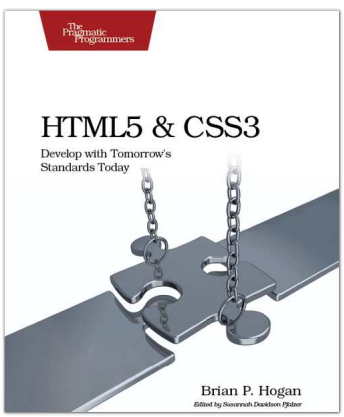
In this book you'll get a hands-on tour of Clojure, Haskell, Io, Prolog, Scala, Erlang, and Ruby. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages side-by-side. You'll learn something new from each, and best of all, you'll learn how to learn a language quickly.



Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages
Bruce A. Tate
(300 pages) ISBN: 978-1934356-59-3. $34.95
http://pragprog.com/titles/btlang

HTML5 and CSS3

HTML5 and CSS3 are the future of web development, but you don't have to wait to start using them. Even though the specification is still in development, many modern browsers and mobile devices already support HTML5 and CSS3. This book gets you up to speed on the new HTML5 elements and CSS3 features you can use right now, and backwards compatible solutions ensure that you don't leave users of older browsers behind.



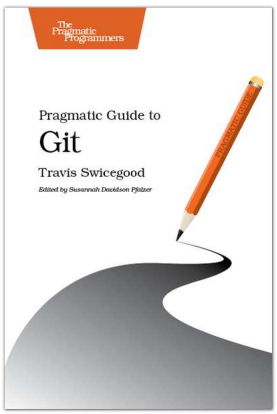
HTML5 and CSS3: Develop with Tomorrow's Standards Today
Brian P. Hogan
(280 pages) ISBN: 9781934356685. $33.00
http://pragprog.com/titles/bhh5

More from PragProg.com

Pragmatic Guide to Git

New Git users will learn the basic tasks needed to work with Git every day, including working with remote repositories, dealing with branches and tags, exploring the history, and fixing problems when things go wrong. If you're already familiar with Git, this book will be your go-to reference for Git commands and best practices.

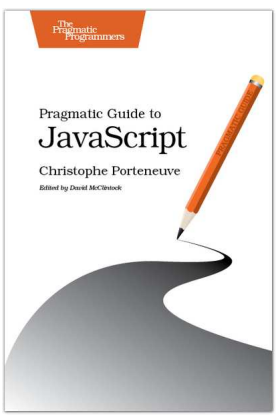
Pragmatic Guide to Git
Travis Swicegood
(168 pages) ISBN: 978-1-93435-672-2. $25.00
http://pragprog.com/titles/pg_git



Pragmatic Guide to JavaScript

JavaScript is now a powerful, dynamic language with a rich ecosystem of professional-grade development tools, infrastructures, frameworks, and toolkits. You can't afford to ignore it–this book will get you up to speed quickly and painlessly. Presented as two-page tasks, these JavaScript tips will get you started quickly and save you time.

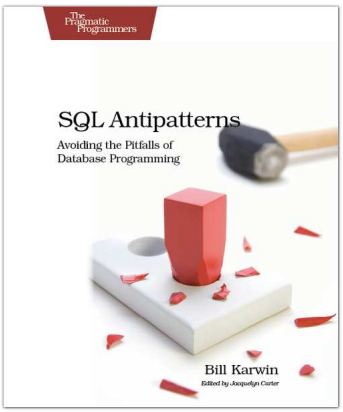
Pragmatic Guide to JavaScript
Christophe Porteneuve
(150 pages) ISBN: 978-1934356-67-8. $25.00
http://pragprog.com/titles/pg_js



More from PragProg.com

SQL Antipatterns

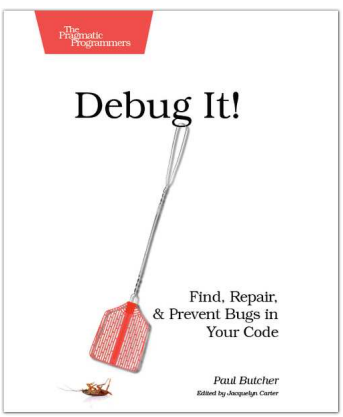
If you're programming applications that store data, then chances are you're using SQL, either directly or through a mapping layer. But most of the SQL that gets used is inefficient, hard to maintain, and sometimes just plain wrong. This book shows you all the common mistakes, and then leads you through the best fixes. What's more, it shows you what's *behind* these fixes, so you'll learn a lot about relational databases along the way.



SQL Antipatterns: Avoiding the Pitfalls of Database Programming
Bill Karwin
(300 pages) ISBN: 978-19343565-5-5. $34.95
http://pragprog.com/titles/bksqla

Debug It!

Debug It! will equip you with the tools, techniques, and approaches to help you tackle any bug with confidence. These secrets of professional debugging illuminate every stage of the bug life cycle, from constructing software that makes debugging easy; through bug detection, reproduction, and diagnosis; to rolling out your eventual fix. Learn better debugging whether you're writing Java or assembly language, targeting servers or embedded micro-controllers, or using agile or traditional approaches.



Debug It! Find, Repair, and Prevent Bugs in Your Code
Paul Butcher
(232 pages) ISBN: 978-1-9343562-8-9. $34.95
http://pragprog.com/titles/pbdp

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home Page for Code in the Cloud
http://pragprog.com/titles/mcappe
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mcappe.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)