The Pragmatic Programmers

Code in the Cloud

Programming Google AppEngine

Mark C. Chu-Carroll

Edited by Colleen Toporek

Code in the Cloud
Programming Google AppEngine

Mark C. Chu-Carroll

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

ISBN-10: 1-934356-63-8

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

Printed on acid-free paper.

B1.0 printing, March 17, 2010

Version: 2010-3-16

Contents

Part I

Getting Started with AppEngine

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 using AppEngine, 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, 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 is the server? 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. 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 point of view of a user—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 companies 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 1 terabyte 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 whomever 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 have 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, named 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 30 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 2 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 that 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 computer, and lets them sync that music with their iPod 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 huge 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 the options.

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 AppEngine, 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, AIX, 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), the costs vary from $0.10 per CPU hour and $0.10 per gigabyte of bandwidth to around $2,000 for a dedicated server for one year.

Amazon S3

Amazon provides another extremely interesting cloud service, which is very different from the other cloud offerings. S3, simple storage services, 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 it 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 15 cents per gigabyte per month, and about 10 cents per gigabyte uploaded, 17 cents per gigabyte downloaded.

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 AppEngine, 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 how to build 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 an AppEngine 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 AppEngine 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, and move into Java. For building complex applications, Java can be a lot more convenient. And AppEngine provides access to an absolutely brilliant framework called GWT, which abstracts away most of the boilerplate plumbing of a web-based cloud application, and allows 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 GWTs 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 AppEngine datastore service; how to implement server-side processing and computation using things like cron; and how to integrate security and authentication into you AppEngine application.

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

Chapter 2

Getting Started

In this chapter, we're going to take our first look at AppEngine, and get started using it. You'll learn how to:

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

This definitely 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 an AppEngine Account

The first thing you need to do in order to write cloud applications with AppEngine is open an AppEngine account. When you're developing for the cloud, you're renting computing and storage resources for your application; the AppEngine 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 AppEngine is free. A basic, no-charge AppEngine account gives you the ability to run up to 10 applications, along with:

1. 46 hours of CPU time per day

Counting CPU Time

The CPU time resources may look strange: there are only 24 hours in a day, but you get 46 hours of CPU time. In AppEngine, your application isn't running on one server; it's running in a Google datacenter. 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.

2. 10 gigabytes per day each of outgoing and incoming bandwidth
3. 1 gigabyte of data storage
4. Privileges to send email to 2,000 distinct recipients

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

To get an AppEngine 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, get started with AppEngine 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 cellphone. In order to prevent spammers from setting up AppEngine 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 AppEngine 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 10 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

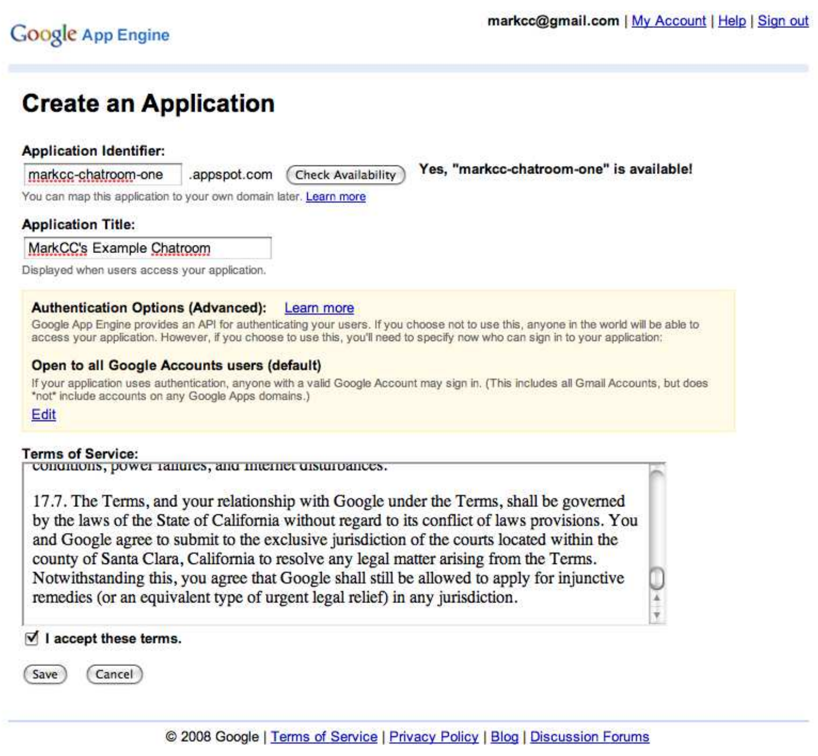
Now that you have an AppEngine account, the first thing you'll want to do is create an application. Already, the process of developing for the cloud is a bit different from normal application development. To write a new program to run on your own computer, you'd just open up an editor and start typing. But for a cloup 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 AppEngine tools, make sure that you have Python installed on your machine. If you're running either Linux or MacOS, Python should be installed automatically. If you're using Windows, you'll need to download and install a Python interpreter. 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 AppEngine Python SDK by logging into the AppEngine account you created in the previous section, and clicking "Create at 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. (AppEngine is updated frequently, so the exact form may appear slightly different.)

To create your application, you need to provide some information to the AppEngine service:

An application identifier. This is a unique name for your application, distinct from every other application being run by *any* other AppEngine 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; 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 AppEngine 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



Create an Application

Application Identifier: markcc-chatroom-one .appspot.com [Check Availability] **Yes, "markcc-chatroom-one" is available!**

You can map this application to your own domain later. Learn more

Application Title: MarkCC's Example Chatroom

Displayed when users access your application.

Authentication Options (Advanced): Learn more

Google App Engine provides an API for authenticating your users. If you choose not to use this, anyone in the world will be able to access your application. However, if you choose to use this, you'll need to specify now who can sign in to your application:

Open to all Google Accounts users (default)

If your application uses authentication, anyone with a valid Google Account may sign in. (This includes all Gmail Accounts, but does *not* include accounts on any Google Apps domains.)

Edit

Terms of Service:

conditions, power failures, and internet disturbances.

17.7. The Terms, and your relationship with Google under the Terms, shall be governed by the laws of the State of California without regard to its conflict of laws provisions. You and Google agree to submit to the exclusive jurisdiction of the courts located within the county of Santa Clara, California to resolve any legal matter arising from the Terms. Notwithstanding this, you agree that Google shall still be allowed to apply for injunctive remedies (or an equivalent type of urgent legal relief) in any jurisdiction.

☑ **I accept these terms.**

Save
Cancel
© 2008 Google | Terms of Service | Privacy Policy | Blog | Discussion Forums

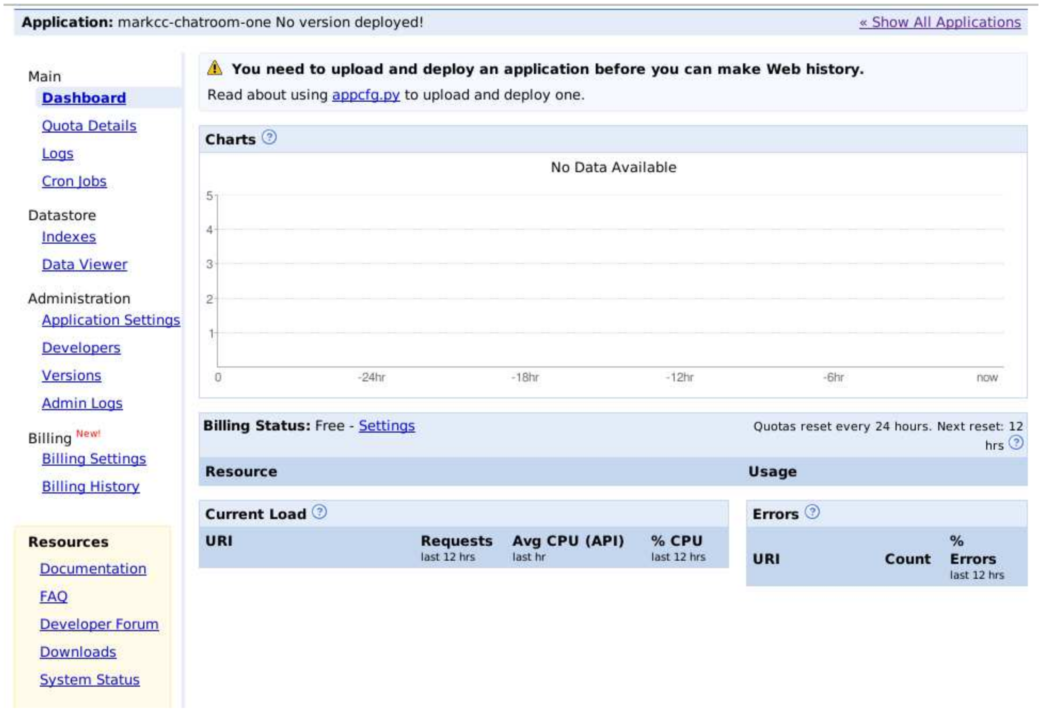
Figure 2.1: The "Create Application" form

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 will turn up in search results that include your application. For the example, I used "MarkCC's Example Chatroom". You can change the application title from the control panel anytime you want.

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 15, *Security in AppEngine Services*, on page 157.

Terms of service. Before you can create an application on AppEngine, you have to accept Google's terms of service. Take the time to read



Application: markcc-chatroom-one No version deployed!

You need to upload and deploy an application before you can make Web history.
Read about using appcfg.py to upload and deploy one.

No Data Available

Billing Status: Free - Settings

Quotas reset every 24 hours. Next reset: 12 hrs

Resource	**Usage**

Current Load

URI	**Requests** last 12 hrs	**Avg CPU (API)** last hr	**% CPU** last 12 hrs

Errors

URI	**Count**	**% Errors** last 12 hrs

Figure 2.2: The AppEngine control panel

through, so you understand the commitments you're making, and the guarantees Google is giving to you as an AppEngine developer. Then click the checkbox to indicate that you accept the terms.

When you're done filling the form out, click Save, and the framework of your brand new application will be created by AppEngine.

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 AppEngine server. You need to write them locally, and then use an administrative script to transfer them into the AppEngine 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 labelled "Resources". This box contains links to software, forums, and documentation that you'll want as you learn and use AppEngine. For right now, click on the download link, and download the appropriate version of the AppEngine 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 AppEngine environment that you can use to test your application on your local computer.
- appcfg.py uploads and configures application using AppEngine in the cloud.

2.3 Starting to Program in Python with AppEngine

Now we're ready to do some programming!

Python-AppEngine 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 AppEngine 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 AppEngine. Any framework and any Python library that was written for CGI scripting can be used in AppEngine—just include the framework/library files when you upload your application code.

The easiest way to work with AppEngine is to use its own framework, called "webapp". Webapp is a very simple but very powerful framework, which makes it easy to use AppEngine and 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

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 two reasons. First, Python is a nice, simple language. We can start writing AppEngine 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, 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 nderstand 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.

AppEngine, you'll be able to write AppEngine applications using other frameworks, as well.

In most of this book, we'll be working on a chatroom 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 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 AppEngine 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 AppEngine 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 AppEngine, we do that by writing a file named app.yaml:

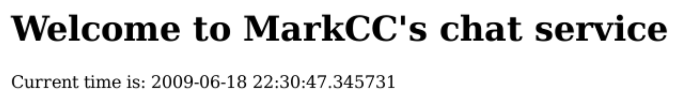
Download chatone/app.yaml

```
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:

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 number 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.


Welcome to MarkCC's chat service

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

Figure 2.3: Our welcome program in a web browser

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 AppEngine 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: 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. If you 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 what's in Figure 2.3, on the previous page.

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 developers interface to AppEngine, so it supports a number of different applications. To send code to the server, use its update command:

```
$ appcfg.py update chat
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 simple example, we can start to see the basic flavor of programming for AppEngine. 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 the work manually that way 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 webserver 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 simple enough that we don't need to use the templating capabilities, but we'll look at them in detail in Chapter 6, *Organizing AppEngine Code: Separating UI and Logic*, on page 67.

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 that 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 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() in a subclass of RequestHandler.

❹ Instead of generating things like 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.

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 on to particular application scripts; 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_wsgiapplication. 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 chat. That will upload the changes. Immediately after running the update, going to the application URL will run the new version of the code. It will still appear in AppEngine 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 AppEngine is like. In later chapters, we'll look at all of the aspects of developing interesting web applications in AppEngine 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 AppEngine control panel.

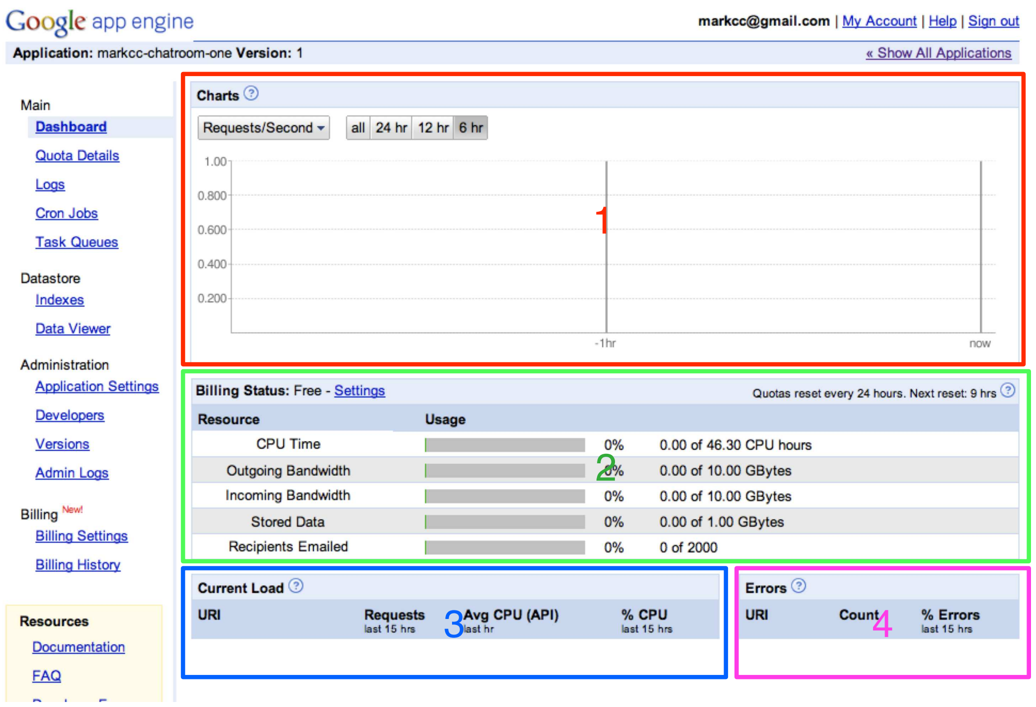
The monitoring features of the AppEngine control panel is going to be *really* important for you. Your cloud application is running on a computer that you don't control. You can't just run a profiler or a debugging to try to see why things aren't working the way you expect. So when you find that something 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, on my day-job, I build cloud-based data analysis applications for Google. I've had cases where an analysis that usually runs for 1 hour suddenly started taking 8 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 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 in the control panel, from information logged by your code while it was running, to information recorded by the AppEngine servers, to detailed information about the resources that your application has consumed.

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:

- 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.



Billing Status: Free - Settings

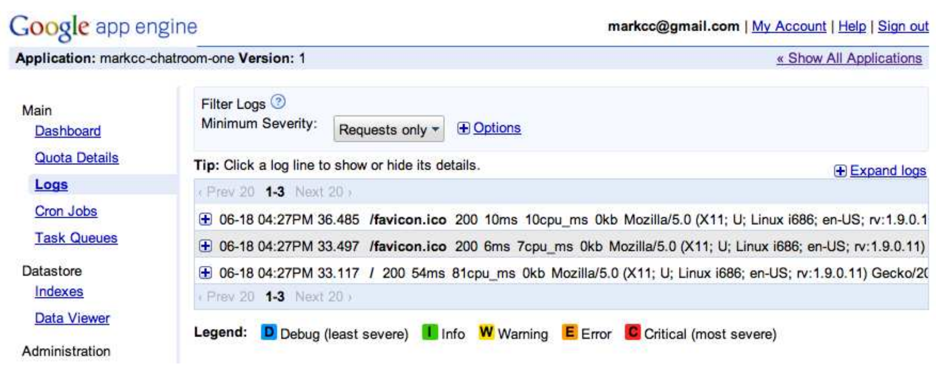
Quotas reset every 24 hours. Next reset: 9 hrs

Resource	Usage		
CPU Time		0%	0.00 of 46.30 CPU hours
Outgoing Bandwidth		0%	0.00 of 10.00 GBytes
Incoming Bandwidth		0%	0.00 of 10.00 GBytes
Stored Data		0%	0.00 of 1.00 GBytes
Recipients Emailed		0%	0 of 2000

Figure 2.4: The sections of the control panel

- 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;
- 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.
- An error view, which shows detailed information about any errors that occured in your appplication.

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 AppEngine developers and Google's AppEngine team; the official, up-to-the-minute AppEngine 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


Google app engine
markcc@gmail.com | My Account | Help | Sign out
Application: markcc-chatroom-one Version: 1
« Show All Applications
Main
Dashboard
Quota Details
Logs
Cron Jobs
Task Queues
Datastore
Indexes
Data Viewer
Administration
Filter Logs
Minimum Severity: Requests only
Options
Tip: Click a log line to show or hide its details.
Expand logs
‹ Prev 20 1-3 Next 20 ›
Legend: D Debug (least severe) I Info W Warning E Error C Critical (most severe)

```
06-18 04:27PM 36.485 /favicon.ico 200 10ms 10cpu_ms 0kb Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.1
06-18 04:27PM 33.497 /favicon.ico 200 6ms 7cpu_ms 0kb Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.11)
06-18 04:27PM 33.117 / 200 54ms 81cpu_ms 0kb Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.11) Gecko/20
```

Figure 2.5: The request log view

encounter developing AppEngine programs will be similar to things that other AppEngine 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 up the left-hand 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 AppEngine server generates a data record describing the request. In addition, errors are logged, and you can add logging statements to your program. For now, we'll just look at request logging.

Open the application's control panel. The left-hand section of the screen contains a collection of links to various views and tools for monitoring, managing, and administrating your application. In the topmost section, labelled "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 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 2.5, on the previous page 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 AppEngine account, and the AppEngine tools are installed. You're ready to build an AppEngine application. In the next chapter, we'll start building a real chat application that runs in the cloud, using the AppEngine tools that we just finished setting up.

References and Resources

AppEngine Developers Guide . . . . . http://code.google.com/appengine/docs/
The official AppEngine 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 AppEngine with Python

Chapter 3

A First Real Cloud Application

In this chapter, we're going to build our first non-trivial cloud application—a basic Python chatroom. Along the way, we'll look at:

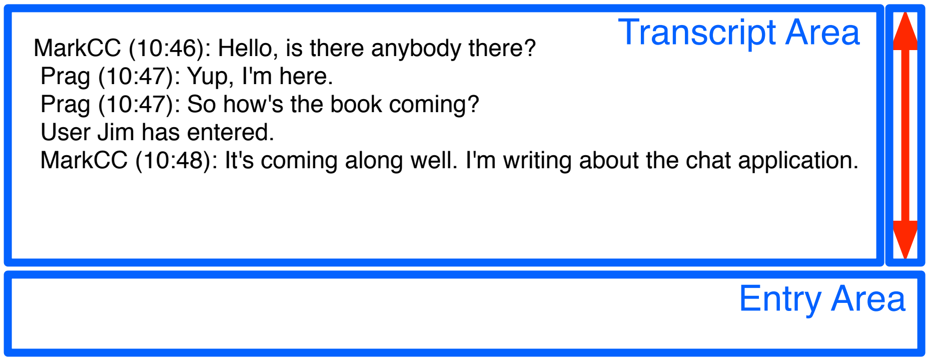
- The HTTP protocol used by cloud applications, and see how cloud applications communicate
- How to take a simple program written in Python, and wrap it in an HTTP framework so that it works in the cloud
- How managing data and variables is different in a cloud application

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 multi-user. 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 AppEngine.



MarkCC (10:46): Hello, is there anybody there?
Prag (10:47): Yup, I'm here.
Prag (10:47): So how's the book coming?
User Jim has entered.
MarkCC (10:48): It's coming along well. I'm writing about the chat application.

Figure 3.1: A mockup of the chat interface

For now, we'll ignore the user interface, and focus our attention on the backend. (We'll deal with the user interface in Chapter 7, *Making the UI Pretty: Templates and CSS*, on page 84.) For now, we're just going to write a basic backend for a traditional local chat application, with the calls to hook a user interface on to it. We're not going to implement the entire thing in this chapter; instead, getting to the point where we can do everything that's in this code will be our goal for most of this book.

A basic chat application is very simple. 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 backend 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 AppEngine 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's 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.

In our space, we want multiple conversations to be available, 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, just 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, and so they need to be implemented differently. In the rest of this book, we'll build up to the point where we have an AppEngine cloud app that does everything that this one does—but in the cloud way. You'll see just 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 chatroom 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: the timestamps are distractingly verbose, and there's no formatting to the text to make it 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 chatroom 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 backend of your application, and you 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 backend 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 backend and the frontend 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* AppEngine applications, are built using a basic protocol called HTTP (Hypertext Transport 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 AppEngine application works. Before we examine how to put together an application protocol for our chatroom, we need to review the basics of HTTP.

HTTP is a simple request/response client/server protocol. In other words, 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 URL (Universal Resource Location); 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 don't need to 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 once in a while, it's useful.

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 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 AppEngine services will process has three parts:

1. The request line, consisting of the HTTP method for the request, followed 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.

2. A series of *header* lines, containing metadata about the request (like the "Content-Type" specification we used in the welcome application in Section 2.3, *Starting to Program in Python with AppEngine*, on page 23). Most of your 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.

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

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. One means "Informational response".

2. Two means success.

HTTP Status Codes

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

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 they 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 authenticated, that user is not permitted to access it.

404 Not found No resource at the specified location.

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 mispell the URL in a **POST** request.

3. Three means redirect, which tells the client that the client should 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. Four indicates a client error (for example, 401 means that the client requested a resource which doesn't exist on the server).
5. Five 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** for submitting chat messages. A request could look like the following:

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

```
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 AppEngine webapp, 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 chatroom. What are the things that you want to be able to do?

First of all, you'll 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 chatroom, and then show it to you.

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 chatroom. 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 chatroom. 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 in AppEngine, we need to build a RequestHander that implements **GET** for the chat room content, and another RequestHandler that receives the **POST**s, 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 18. 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="/talk" method="post">
           <div><b>Name:</b>
           <textarea name="name" rows="1" cols="20"/></div>
           <p><b>Message</b></p>
           <div><textarea name="message" rows="5" cols="60"/></div>
           <div><input type="submit" value="Send ChatMessage"/></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 RequestHandler, and inside of it, we include a 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

```
class ChatRoomPoster(webapp.RequestHandler):
    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 users 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.

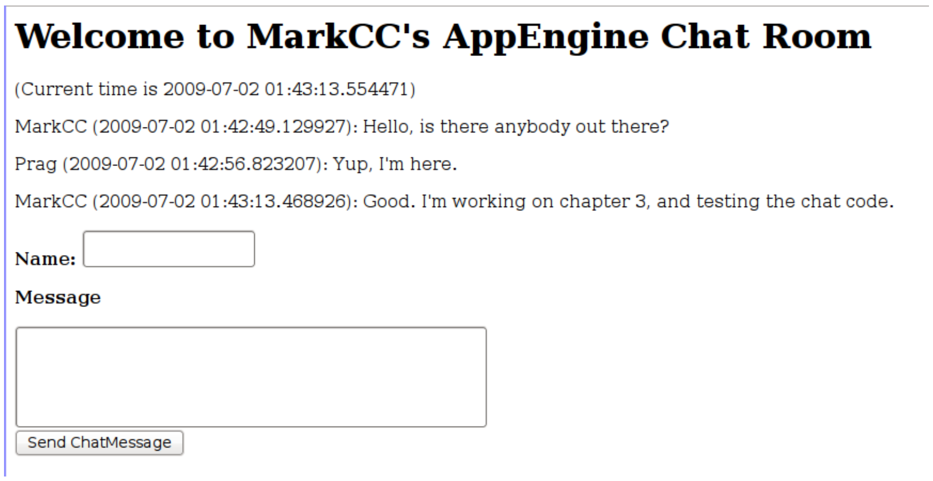
Download chattwo/app.yaml

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

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

The Python webapp glue code is:

Download chattwo/chattwo.py

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

def main():
```


Welcome to MarkCC's AppEngine Chat Room

(Current time is 2009-07-02 01:43:13.554471)

MarkCC (2009-07-02 01:42:49.129927): Hello, is there anybody out there?

Prag (2009-07-02 01:42:56.823207): Yup, I'm here.

MarkCC (2009-07-02 01:43:13.468926): Good. I'm working on chapter 3, and testing the chat code.

Figure 3.2: The Chatroom app in action

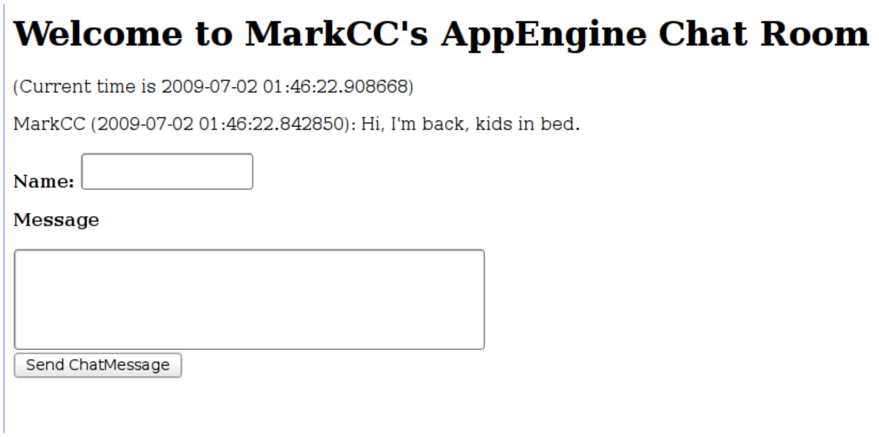
```
    run_wsgi_app(chatapp)

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

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

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 chatroom. So what happened to the transcript that we so before? Where did the old messages go?



Welcome to MarkCC's AppEngine Chat Room

(Current time is 2009-07-02 01:46:22.908668)

MarkCC (2009-07-02 01:46:22.842850): Hi, I'm back, kids in bed.

Name:

Message

Figure 3.3: The Chatroom app after taking a break

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 datacenter. There's no guarantee that any two requests will be routed to the *same* server, or even to servers on the 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 AppEngine server, it was running out in the cloud. When the first request was received by AppEngine, it started a Python interpreter, and used it to run the request. When I submitted a message, that sent a second request off to AppEngine. The main AppEngine 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 it wasn't busy—so it routed the request to that interpreter.

But then I took a break, and was away from my computer for 15 minutes; at some point, the AppEngine 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 chatroom 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 AppEngine. 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://www.w3.org/Protocols/rfc2616/rfc2616.html
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 AppEngine. Many AppEngine facilities are modeled on or borrowed from the Django framework.

Chapter 4

Managing Data in the Cloud

In this chapter, we're going to modify our chat application to use persistent storage with 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's AppEngine'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 chatroom application uploaded to AppEngine—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. AppEngine's servers definitely don't have it loaded on any computers right now. If I load the chat app's page in a browser, AppEngine 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 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

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 when I see code that isn't functional, I cringe a little. Almost everything that I write ends up being at least mostly functional—the places where I use non-functional code, it's because the language and compiler aren't up to the task of keeping the code efficient.

having to deal with fancy persistent storage. But if your application 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 will not be able to handle the number of requests being received. But in the cloud, as the number of requests increases, the number of machines running your application also increases, so no matter how many requests you're receiving, 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

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 AppEngine, provide a more flexible database-like storage. We'll only look at the AppEngine datastore, but there are plenty of others, some of which can be used by AppEngine programs.

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 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

AppEngine has a custom data persistence system called *datastore*. 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, it's got 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.

Creating and Storing Persistent Objects

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 simple, and easy to use. As you use datastore more, you can start to use more complex features as you need them. For now, we'll stick with the basics.

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 datastore, you have to give up some of that flexibility. With 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 datastore is by jumping right in and looking at some code. As I said, in datastore, you need to define a model to tell 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 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. 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 12, *Advanced Datastore*, on page 154

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 a simple 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 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 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 12, *Advanced Datastore*, on page 154, 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 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 auto-generate 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
❶ messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY time")
for msg in messages:
    self.response.out.write("<p>%s</p>" % msg)
# Output the set of chat messages
messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY time DESC LIMIT 20")
messages.reverse()
for msg in messages:
    self.response.out.write("<p>%s</p>" % msg)
```

You 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 time").

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 SQLCREATE TABLE statements.

That is, in fact, very much *not* the case. Datastore has a much richer range of data types and data structures than a relational database. In 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 datastore is *better than* a relational database; just *different*. For example, relational databases have much better performance on joins than datastore. But datastore lets you use familiar data structures that make sense in your application in a simple, scalable way.)

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. 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.

People using a chatroom don't want to have to constantly scroll through 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 20 messages in the chatroom, or they might want to only see messages posted within the last 5 minutes.

Using GQL, it's downright trivial to fix the verbosity issue by adding clauses to our GQL query. To see the 20 most recent messages, we can add a **LIMIT** clause, and to see the messages from the last 5 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. That's very easy: 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 20* values that match the query in the specified sort order. Since we want to get the 20 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 exactly the same as ChatRoomPage, except for two lines. I copied ChatRoom-Page, 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 time '
                               'DESC LIMIT 20')
❷       messages.reverse()
        for msg in 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 20 most recent posts to the chat will be the first ones in the query result (ORDER BY time DESC), and limited it to 20 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 sub-part 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 two of the biggest limitations of GQL:

1. 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 string.
2. You can't compare things in a query directly to literal values. You can only do comparisons between queried values and *parameters*.

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, we could have written the number-limited view as 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. Parameters can be either numbered or named. If they're named, specify their value using a named parameter to the Python call. Again, for example, we could use a named parameter in the number-limited view query like ChatMessage.gql("ORDER BY time DESC LIMIT :limit", limit=20).

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."

It's easy to say "now minus five minutes" in Python 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.

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 datastore documentation.

Chapter 5

Using AppEngine Services for Login Authentication

Most web applications that you build with AppEngine must to be able to keep track of users. You need to be able to let users login, 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 AppEngine *service*, which is a piece of the AppEngine 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 chatrooms. 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 AppEngine application that you write.

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

Using Other Domains

If you really want to have your own logins, you can set up and register a domain using Google sites, and then use Google's login service with accounts on that domain. To do that, write the code as described here, but in the app.yaml file, declare a secure domain using the secure: attribute. We'll cover this in more depth when we get to security issues in Chapter 15, *Security in AppEngine Services*, on page 157

that is accessible to every AppEngine 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 AppEngine services.

The easiest way to set up authentication is to use the AppEngine 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.

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 log in and log out 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 is frequently the full name of the user, but it's configurable by the user. In my case, it's "Mark C. Chu-Carroll".

email() The user's email address. In my case, "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'll check if the user is logged in, using 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 pretty simple: 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 AppEngine 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 login 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 let users login in Section 5.2, *Letting Users Log In*, on the previous page. Integrating that into our chat page request handler is simple. 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 time")
      for msg in messages:
        self.response.out.write("<p>%s</p>" % msg)
      self.response.out.write("""
❷        <form action="/talk" 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>
```

```
        """ % 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 also simple. In fact, 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

```
class ChatRoomPoster(webapp.RequestHandler):
  def post(self):
❶    user = users.get_current_user()
    msgtext = self.request.get("message")
    msg = ChatMessage(user=user.nickname(), message=msgtext)
    msg.put()
    # 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 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 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 AppEngine 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 error-prone. In this chapter, you'll learn about *templates*, which organize the code of an AppEngine 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 35, 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 AppEngine 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 lots of template languages. The AppEngine 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 AppEngine 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 laborious and error-prone. 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. The idea behind the use of templaces is that you can separate programs into two basic parts: the computational and the interface. The computational part is where you do the work, and produce the data that you want to render in the webpage. 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 part is all about generating HTML. What it does is take the data produced by the computational part, and render it into HTML. In pretty much every application, a large part of the HTML is fixed; it's boilerplate stuff. 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. When we're writing in Python, 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 embed any computation that's needed to insert dynamic content using special meta-syntax 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 do complicated population using 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 %} {#}
    <p> {{ m.user }} ( {{ m.timestamp }} ): {{ m.message }} </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>
```

In the Django template system, a 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 is 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 simple variable reference: in Django, variable references are written as double-braces surrounding an identifier—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—you can 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 simple 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.

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 time")
      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.

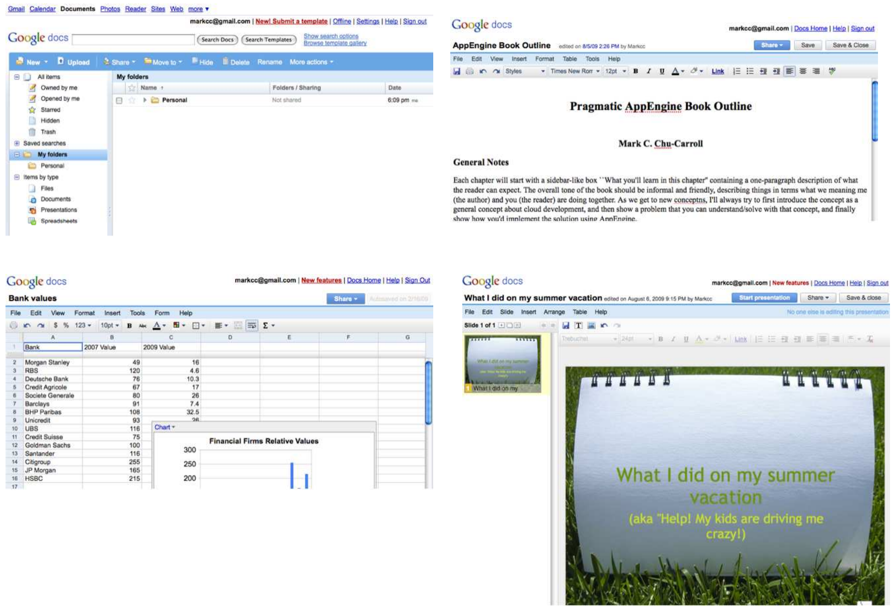
Getting the pathname requires a bit of a trick in AppEngine. As we keep seeing, in the cloud you have less control over your environment than you're used to in a traditional application. Your application and its data are *somewhere*, but you don't know where. The AppEngine server determines where it's 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 AppEngine to tell you where the directory containing your application files is located. AppEngine uses the Python metavariable __file__, which is a reference to the source file for the current module. AppEngine guarantees that any file uploaded in the same directory as a source file can always be referenced from the same directory as that source file. So 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 in to 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.

Almost every good web application has a unified look and feel to all of 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


Google docs
markcc@gmail.com | Docs Home | Help | Sign out
AppEngine Book Outline edited on 8/5/09 2:26 PM by Markcc
Share
Save
Save & Close
File Edit View Insert Format Table Tools Help
My folders
All items
Owned by me
Opened by me
Starred
Hidden
Trash
Saved searches
Personal
Items by type
Files
Documents
Presentations
Spreadsheets
Bank values
markcc@gmail.com | New features | Docs Home | Help | Sign Out
Chart
Financial Firms Relative Values
300
250
200
What I did on my summer vacation edited on August 6, 2009 9:15 PM by Markcc
Start presentation
Share
Save & close
What I did on my summer vacation
(aka "Help! My kids are driving me crazy!)

Pragmatic AppEngine Book Outline

Mark C. Chu-Carroll

General Notes

Each chapter will start with a sidebar-like box ``What you'll learn in this chapter" containing a one-paragraph description of what the reader can expect. The overall tone of the book should be informal and friendly, describing things in terms what we meaning me (the author) and you (the reader) are doing together. As we get to new conceptns, I'll always try to first introduce the concept as a general concept about cloud development, and then show a problem that you can understand/solve with that concept, and finally show how you'd implement the solution using AppEngine.

Bank	2007 Value	2009 Value
Morgan Stanley	49	16
RBS	120	4.6
Deutsche Bank	76	10.3
Credit Agricole	67	17
Societe Generale	80	26
Barclays	91	7.4
BHP Paribas	108	32.5
Unicredit	93	26
UBS	116	
Credit Suisse	75	
Goldman Sachs	100	
Santander	116	
Citigroup	255	
JP Morgan	165	
HSBC	215	

Figure 6.1: Common Styling in Google Docs

application, you provide the user with the feeling that they're using an 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 always tell at a glance that a window contains something from Google Docs, because it *looks* like Google Docs.

The look and feel that defines your application is created by a combination of common page layouts and shared styles 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, while maintaining the

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 both for 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!

defaults from the master template shared by everything in the application.

These views will have common features like a 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 %}
    <div id="navbar">
      <ul>
        {% for c in chats %}
        <li><a href="{{ c.url }}">c.name</a></li>       {% end for %}
      </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, which 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 </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 with its pagecontent block 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. So 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 the messages that we pass to the UI template; and 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 time "
                             "DESC LIMIT 20")
      messages.reverse()
      template_values = {
        'title': "MarkCC's AppEngine Chat Room",
        'msg_list': messages,
        }
      path = os.path.join(os.path.dirname(__file__), 'chat.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 ({{ 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 hardcode 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")
    chat = self.request.get("chat")
    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 20 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.

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 }} </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 69, 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 the 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 simple. It's 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! The template is shown below.

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 }} </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 add a parameter to the POST request generated by the entry form at the end of the page, 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 AppEngine 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 going to eventually be able to add and remove chats, there's no fixed list of chats that we can hardcode 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,
             }
         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 time", requested_chat)
         template_params = {
             'title': "MarkCC's AppEngine Chat Room",
             'msg_list': messages,
             'chat': requested_chat
             }
         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 name "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 AppEngine programs, so webapp supplies an extensive library of methods for examining and manipulating URLs. To get at query parameters from a request URL, it provides a method get, which takes the name of a designed 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 page-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 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 define and lay out a really nice-looking application in a web page, and how to use CSS with templates to turn sour chat application from an ugly-but-functional system to something much better.

References and Resource

The Django Template Language: for Template Authors. . .
. . . http://www.djangoproject.com/documentation/0.96/templates/
The official Django template documentation.

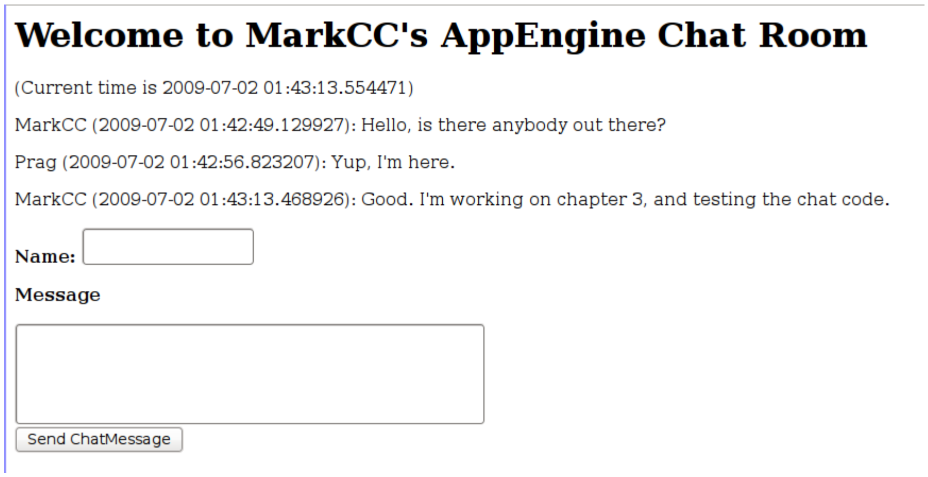
Googe AppEngine Template Documentation. . .
. . . http://www.djangoproject.com/documentation/0.96/templates/
Google's documentation on using Django templates in AppEngine applications.

The Django Book 2.0 . . . . http://www.djangobook.com/en/2.0/
An online version of a book on the full Django framework, included 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 towards filling in the missing functionality of our application. But, to be blunt, our application is still *ugly*. In this chapter, we're going to pull our application together, formatting and styling it using a combination of templates and CSS.



Welcome to MarkCC's AppEngine Chat Room

(Current time is 2009-07-02 01:43:13.554471)

MarkCC (2009-07-02 01:42:49.129927): Hello, is there anybody out there?

Prag (2009-07-02 01:42:56.823207): Yup, I'm here.

MarkCC (2009-07-02 01:43:13.468926): Good. I'm working on chapter 3, and testing the chat code.

Figure 7.1: The unstyled, ugly chat interface.

7.1 Introducing CSS

In Figure 7.1, on the previous page, there's a screenshot of a chat in our application. It's ugly, isn't it? Or if not ugly, at least extremely bland. It's just a standard web page. 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*.

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 AppEngine. 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 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, with 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, extremely difficult to maintain, and it 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, 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; 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 re-use 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 stylesheet 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 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 chatroom'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. It's selector is h1—so it only applies to top-level headers. The property it specifies is text-decoration, which is used to modify text by adding things like underlines, shadows, 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 users' 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 chatroom 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 background-color property for both the <*body*> element, and the <*h1*> element. That's an extremely simple use of *cascading*. Styles, both from multiple stylesheets and from multiple statements within the stylesheets, 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 stylesheet; a page-specific linked stylesheet will take preference over a site-default stylesheet; 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 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 chatroom names, *and* wrapped it in an annotated <*p*> tag. The tag includes an 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 chatroom—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. The CSS for styling our navbar is shown below:

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.

Now, 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 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 }} </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:

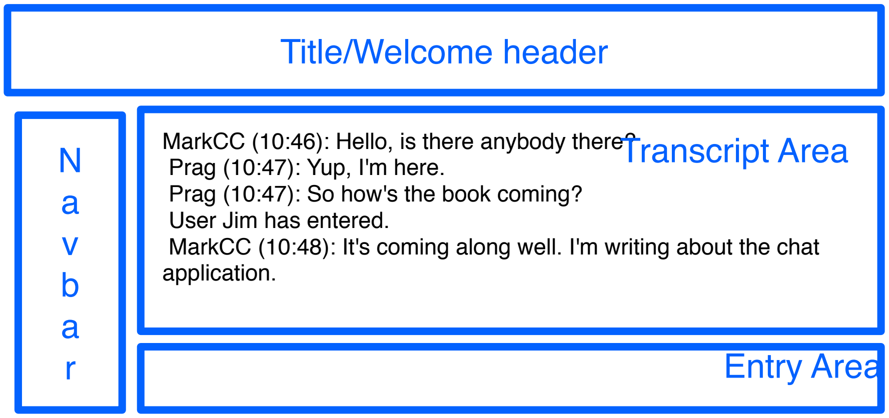
1. There are properties that we might want to use in many places in our document. For example, we might want to be able to make some of our text red; we could create a class red. Then anywhere we wanted to use red text—whether it was a header, a paragraph, a small section of boldfaced text—we could add class="red" 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 XML, 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 users 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 7.2, 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-hand side of the screen; an area showing the chat transcript; and beneath the transcript is the entry form. Each of those elements are, basically, a rectangular region, and we want them to be laid out in a specific way.



MarkCC (10:46): Hello, is there anybody there?
Prag (10:47): Yup, I'm here.
Prag (10:47): So how's the book coming?
User Jim has entered.
MarkCC (10:48): It's coming along well. I'm writing about the chat application.

Figure 7.2: The block structure of the chat user interface

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).

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 7.2, 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 browsers windows that could be any size, on computer displays with different resolutions, 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 in to 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 <*div*>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*.

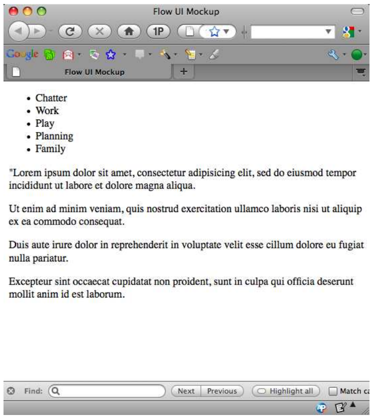
Floats

A float is a <*div*> (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 confusing enough that we'll 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. The HTML is shown below, and you can see what it looks like when it's rendered without any CSS in Figure 7.3, on the following page.

Download css-chat/flow-mockup.html

```
<html>
   <head>
        <title>Flow UI Mockup</title>
        <link rel="stylesheet" media="screen"
```



- Chatter
- Work
- Play
- Planning
- Family

"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

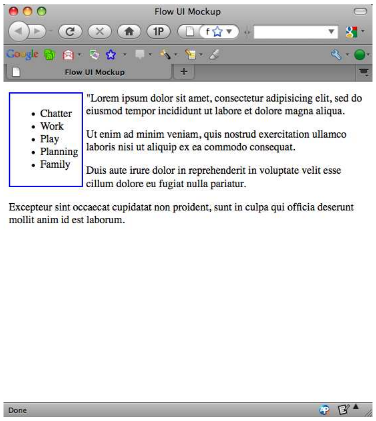
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 7.3: A mockup of the chat interface

```
          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>
```



- Chatter
- Work
- Play
- Planning
- Family

"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 7.4: A mockup of the chat interface

We can add make the sidebar float using CSS:

Download css-chat/flow-mockup.css

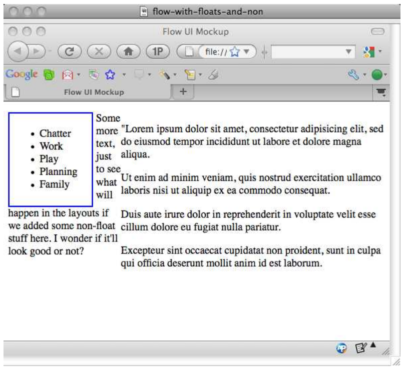
```
   #sidebar {
❶          float: left;
❷          border: 2px solid #0000FF;
❸          padding: 5px;
           margin-right: 5px;
   }
```

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, doublegrooved, ridge, inset, or outset. For this, we'll use a simple solid outline.

❸ To make things look good, we'll frequently want to add small amounts of space around things. CSS allows you to use two kinds of space: *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 shown in Figure 7.4.



- Chatter
- Work
- Play
- Planning
- Family

Some more text, just to see what will happen in the layouts if we added some non-float stuff here. I wonder if it'll look good or not?

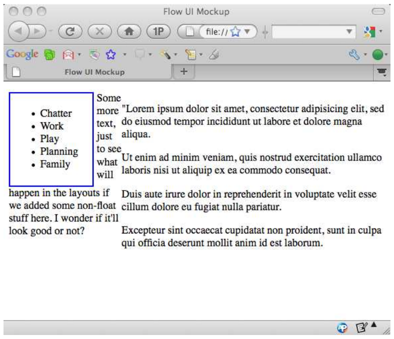
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 7.5: Chat mockup with two-column floats



- Chatter
- Work
- Play
- Planning
- Family

Some more text, just to see what will happen in the layouts if we added some non-float stuff here. I wonder if it'll look good or not?

"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 7.6: Chat mockup with floats and non-floats

Clears

As you can see in Figure 7.4, on the preceding page, 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 just like Figure 7.3, on page 95. 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. An updated stylesheet is below; the result is show in Figure 7.5, on the preceding page.

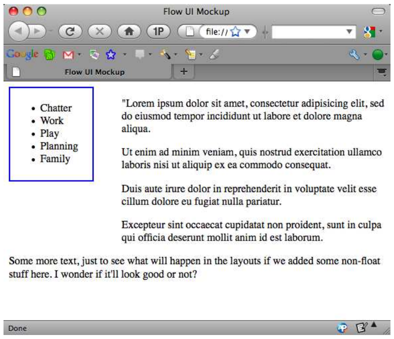
Download css-chat/flow-twocol.css

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

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

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

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—as in Figure 7.6, on the previous page.

What we need is a way of saying "Don't flow around this." And 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 stylesheet. The result is in shown in Figure 7.7, on the following page: exactly what we wanted.



- Chatter
- Work
- Play
- Planning
- Family

"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Some more text, just to see what will happen in the layouts if we added some non-float stuff here. I wonder if it'll look good or not?

Figure 7.7: Chat mockup with floats and clears

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 spend some time tweaking various options, shifting margins, changing colors, re-arranging 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 just 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. The CSS for rendering that is below.

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:

Download css-chat/fake-ui.html

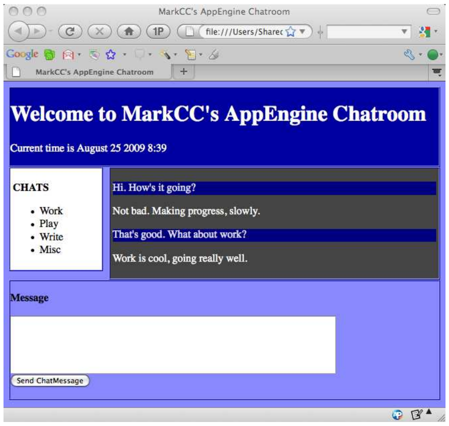
```
<html>
```

```
 <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 August 25 2009 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>

<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>
```

And the rendered result is shown in Figure 7.8, on the next page.

Once we're happy with how the faked one looks, we can hook it up to the live application: add the stylesheet link line to the master template, and upload it to AppEngine.

As we've seen in this chapter, styling an AppEngine 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 capa-



Welcome to MarkCC's AppEngine Chatroom

Current time is August 25 2009 8:39

CHATS

- Work
- Play
- Write
- Misc

Hi. How's it going?

Not bad. Making progress, slowly.

That's good. What about work?

Work is cool, going really well.

Message

Figure 7.8: Faked chat interface for testing CSS

bilities 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 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.

IronMyers Layout Examples . . . . http://layouts.ironmyers.com/
An excellent online resource containing HTML and CSS templates for 224 different web element layouts.

Chapter 8

Getting Interactive

In this chapter, we're going to make a huge leap forwards 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:

- JavaScript, the programming language embedded 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 content of a web page 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 for applications design, which is 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 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 pre-packaged 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 as on 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 a 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 To embed JavaScript code in an HTML page, put 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"—that is, 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: so 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 is simple: it's just 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:

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

All of this has been manageable so far because our code has always followed a basic structure. We've built everything by focusing the structure 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 users 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, 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 30 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 following 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

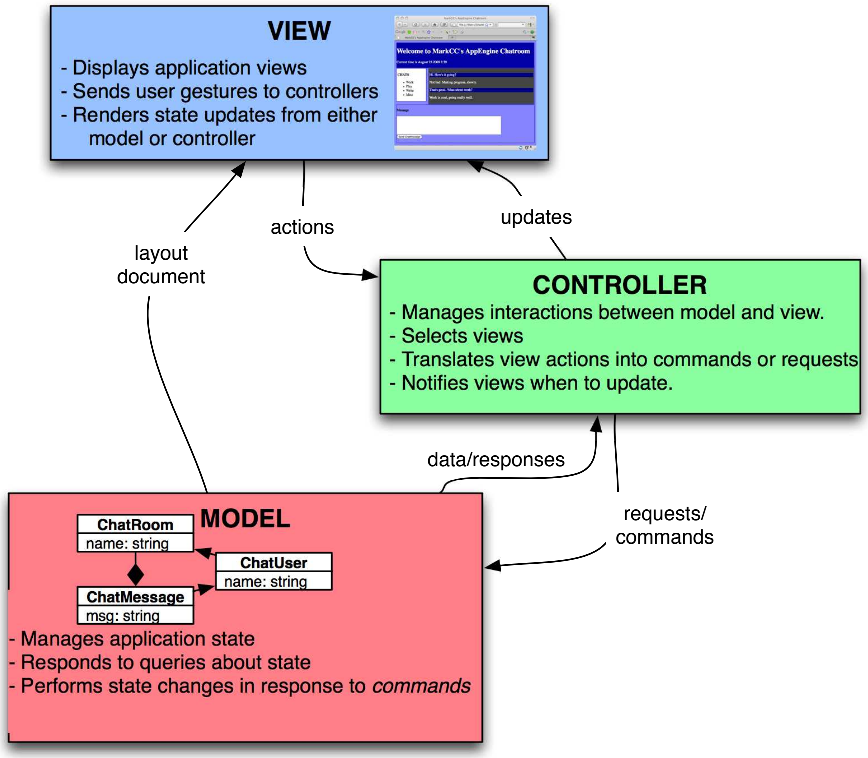

VIEW
- Displays application views
- Sends user gestures to controllers
- Renders state updates from either model or controller
Welcome to MarkCC's AppEngine Chatroom
actions
updates
layout document
CONTROLLER
- Manages interactions between model and view.
- Selects views
- Translates view actions into commands or requests
- Notifies views when to update.
data/responses
requests/ commands
MODEL
ChatRoom
name: string
ChatUser
name: string
ChatMessage
msg: string
- Manages application state
- Responds to queries about state
- Performs state changes in response to *commands*

Figure 8.1: The structure of a cloud MVC application

the application logic or data: it's capable of rendering things when it's told to.

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 translate those into operations that can be performed in the model, and to take content data produced by server, and translate it into a form that the view can render. In the cloud, the controller is the JavaScript code that's executed inside the users' 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 style: 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 managing 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 clients 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 really 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. Code that does that 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 132, 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 thing about an 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

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*. It's an object used to *create* requests, *send* requests that it was used to create, and *handle* the responses to requests that it sent.

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 that your program doesn't need to wait for a response.

the user gets a 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.

Described that way, it all sounds very abstract. But it's really pretty simple—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 recieves 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, is basically sending 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 116. 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):
    requested_chat = self.request.get("chat", default_value="none")   ❶
    if requested_chat == "none" or requested_chat not in CHATS:   ❷
      template_params = {
          'title': "Error! Requested chat not found!",
          'chatname': requested_chat,
          }
      error_template = os.path.join(os.path.dirname(__file__), 'error.html')
      page = template.render(error_template, template_params)
      self.response.out.write(page)
    else:
       template_params = {
          'title': "MarkCC's AppEngine Chat Room",
          'msg_list': messages,
          'chat': requested_chat
          }
       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 different. 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 sent a 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.html

```
<?xml version="1.0" encoding="UTF-8"?>
<ChatUpdate chat="{{ chat }}" time="{{ time }}">

{% for m in msg_list %}
  {% ifequal msg.sender m.user %}
  <p class="sentbyme">
  {% else %}
  <p class="sentbyother">
  {% endifequal %}
{% endfor %}

</ChatUpdate>
```

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

Download interactive/chat.py

```
def DataRequestHandler(webapp.RequestHandler):
        def get(self):
          requested_chat = self.request.get("chat", default_value="none")
❶     messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                             "ORDER BY time", requested_chat)
      template_params = {
          'title': "MarkCC's AppEngine Chat Room",
          'msg_list': messages,
          'chat': requested_chat
                  'time': self.request.get("time", default_value="0")
          }
      path = os.path.join(os.path.dirname(__file__), 'update.html')
      page = template.render(path, template_params)
      self.response.out.write(page)
```

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 to controller. Let's jump right in to the code:

Download interactive/asynch.js

```
❶  var request = new XMLHttpRequest();
   var BASE_URL = "http://mcc-chatroom-one.appspot.com";
❷  var transcript = document.getElementById("chat-transcript");

   function SetUpAndSendRequest(time) {
❸    request.open("GET", BASE_URL + "latest?time=" + time, true);
❹    request.onreadystatechange = function() {
       // Wait until the request is done. Done == ready state 4.
       if (request.readyState != 4) {
         return;
       }
❺      var xmlData = request.responseXML;
       messages = xmlData.getElementsByTagName("p");
❻      for (m in messages) {
         transcript.appendElement(m);

       }
❼      newtime = xmlData.getElementsByTagName("time").getAttribute("latest");
❽      SendUpAndSendRequest(newtime);
     }
     request.send("");
   }
```

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.

❸ 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.

❹ 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 XML-HttpRequest object invokes when it gets its response. In this case, we put the callback function in-line.

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.

❺ 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.

❻ Update the chat transcript by appending the new messages.

❼ At this point, the user interface has been updated with the latest chat messages. But we need to re-issue 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.

❽ 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 that issues the request and 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 %}

❶  <script src="asynch.js"/>

   {% block pagecontent %}
❷  <div id="transcript">
```

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

❶ Our new template extension needs to 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 continually handshakes with the server, getting 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, but that's easy: we just need 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, view, and the controller.

We're ready to move on to some more complicated things! AppEngine 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/?CMP=OTC-KW7501011010&ATT=9780596528386

An excellent textbook on AJAX.

Part III

Programming AppEngine with Java

Chapter 9

AppEngine and Java

We've been doing all our development of AppEngine 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. AppEngine supports two main programming languages: Python and Java. Python is the lightweight dynamic language. Java is the heavy artillery. And in AppEngine, you build Java applications using a toolkit call 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, including:

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

Static Versus Dynamic Languages

One of the great debates among programmers is 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 tradeoff: 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 *do* 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.

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 AppEngine can be much more comfortable than Python.

Tools. Google released a set of plugins for the free Eclipse IDE for building Java/GWT AppEngine services and applications. Eclipse is an absolutely *amazing* tool, and the AppEngine plugins make everything easier. (You can use Eclipse with Python, but there's no specific AppEngine 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 needs 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 JavasScript 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; 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 AppEngine 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 18. 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 AppEngine plugins. The ability to use Eclipse for AppEngine 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.

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 name; when users enter their names, it pops up a dialog box saying hello to them.

The Structure of a GWT Application

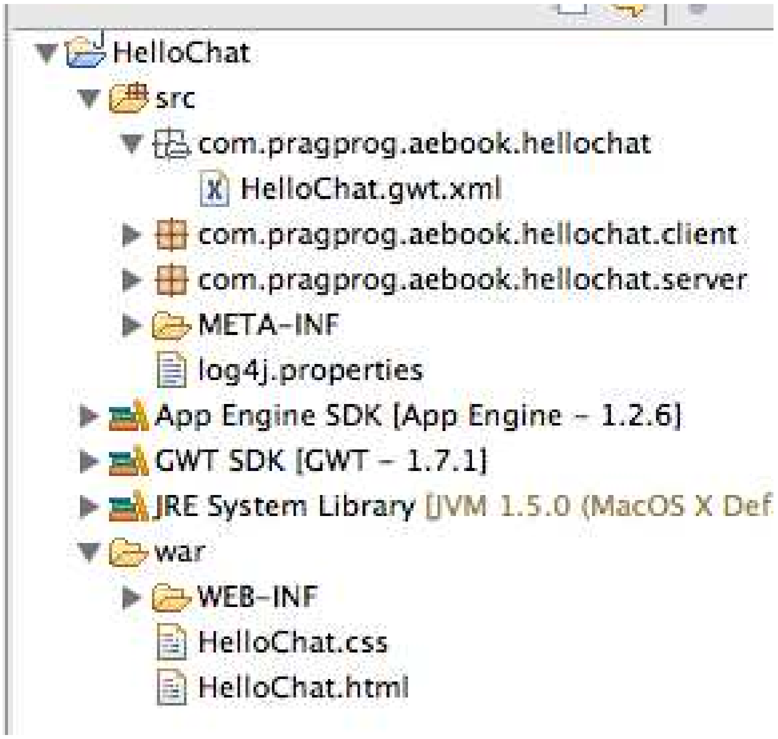
A GWT application consists of a set of *modules*. A module is a GWT package consisting of Java code, JavaScript, HTML files, images, data definitions, and whatever else you need in a web application. The directory structure that you get when you create a GWT/AppEngine project in Eclipse is based on the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



```
HelloChat
  src
    com.pragprog.aebook.hellochat
      HelloChat.gwt.xml
    com.pragprog.aebook.hellochat.client
    com.pragprog.aebook.hellochat.server
    META-INF
    log4j.properties
  App Engine SDK [App Engine - 1.2.6]
  GWT SDK [GWT - 1.7.1]
  JRE System Library [JVM 1.5.0 (MacOS X Def
  war
    WEB-INF
    HelloChat.css
    HelloChat.html
```

Figure 9.1: The GWT project directory structure in Eclipse

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 132.

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

❶ The fundamental unit of code in GWT is a *module*. A module 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 *sub-module* 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, 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 spans 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; 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 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 get a layout box on the page by calling get, and then inserting a GUI widget into it, using add(widget).

5. 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 place-holder text that we put into the box, so if the users start typing, their text will replace the placeholder.

That's it for the basic building of the GUI.

That leaves us 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
1   final DialogBox dialogBox = new DialogBox();
    dialogBox.setText("Remote Procedure Call");
    dialogBox.setAnimationEnabled(true);
2   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();
3   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);
4   dialogBox.setWidget(dialogVPanel);

5   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 free-standing widget. In GWT, that's easy: 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 non-editable 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 pretty simple: it's 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: standalone widgets like this don't actually appear on the users' 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.

5 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:

```
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.

The problem with this approach is that 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 recieved.

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

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

In continuation passing style, that would be something like:

```
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's 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 multi-threaded 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/aeb . . . ok/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 http://gwt.appspot.com/foo/greet. Service interfaces should always specify their relative path this way.

❷ The service declaration is a standard interface declaration, except that it must extend com.google.gwt.user.client.rpc.RemoteService. The super-interface specifies that this 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 uses the Java synchronization interface to mark classes that will be passed in an RPC does *not* mean that it uses Java serialization. It doesn't. 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 guys 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/aeb . . . llochat/client/GreetingServiceAsync.java

```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:

Download workspace/HelloChat/src/com/pragprog/aeb . . . ellochat/server/GreetingServiceImpl.java

```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 only unusual thing in this code is the use of the @SuppressWarning annotation, and it's there because in the later generations of the Java virtual machine, Java serialization uses a version identifier for each class file. If we have a class which 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 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 AppEngine: a local mode, where the application runs on your machine; and a deployed mode, where the application runs in the AppEngine 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, the 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 AppEngine, go to the package explorer view, and right-click on the project. In the project menu that comes up, there's a Google submenu. Just select "Deploy to AppEngine", and your program will be live in the AppEngine cloud. If there's any information about the project that it needs, it will prompt us to fill it in the first time we run the deploy command.

We finished our first basic GWT application. It's 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 AppEngine

In the previous chapters, we examined the pieces of a basic GWT application in AppEngine. 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 AppEngine. 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 AppEngine; specifically, AppEngine puts some restrictions on what code can do, and how it can run in the AppEngine cloud environment, and we'll examine what those restrictions are, and what effect they have on how we write the server side of our AppEngine applications.

10.1 Data Persistence in Java

If you look back at Chapter 4, *Managing Data in the Cloud*, on page 51, 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 backend datastore used by AppEngine 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 AppEngine 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. AppEngine uses that kernel.

Below, we'll see how JDO persistence works for describing persistent objects, and 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 the way that we add attributes in Java is 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;
        }

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

❸       @Persistent
        protected String senderName;

        @Persistent
        protected String message;

        @Persistent
        protected String chat;

        @Persistent
        protected Date 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 Date getDate() {
                return date;
        }
        public void setDate(Date date) {
                this.date = date;
        }

}
```

Let's take a closer look at this example:

❶ For a Java object to be stored in the AppEngine datastore, the class needs to be declared as persistent. In JDO, this is accomplished by attaching a @PersistenceCapableannotation. In full JDO, we need to declare an identity type. AppEngine only supports IdentityType.APPLICATION, but because Java's type system requires annotations to match their declaration, the field must always be declared.

❷ 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 which 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 tree 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.

❸ 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:

1. 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.

2. When a persistent object has a collection of other persistent objects, it owns all of the objects in the collection.

3. 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 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.

4. We can use Java serializable types in data objects, by marking them with the annotation @Persistent(serialized=true). But 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.

5. 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.

6. If we're *not* using Eclipse, we need to add an extra compilation step called *code augmentation*. During augmentation, the AppEngine JDO implementation adds code to classes based on the persistence annotations that allow them to be stored and queried by 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 into the datastore was incredibly simple. We created a persistent object, and then we called its put method—and 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 re-initialize 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 AppEngine 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

```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 you call its .close() method, everything we do is part of an *atomic unit*—that is, it either all succeeds, or it all fails. Every object that we store, every

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. The atomic unit in which everything gets stored successfully or nothing gets stored is called a transaction.

Java datastore makes it really easy to collect multiple storage operations into a single transaction.

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 implement 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/a . . . stchat/client/ChatSubmissionService.java

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

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);
}
```

We turn it into an asynchronous interface as usual:

Download workspace/PersistChat/src/com/pragprog/a . . . t/client/ChatSubmissionServiceAsync.java

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

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);

}
```

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; make the message object persistent, so that it will be saved as part of a transaction; and then 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/a . . . at/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 a 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 by:

```
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 12, *Advanced Datastore*, on page 154.) But for now, retrieving by key isn’t particularly useful.

There *is* one part of the fragment above that’s important: the use of try...finally. As I mentioned before, persistence objects aren’t exactly lightweight. There’s 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 be 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 this time, 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 chatroom, the JDOQL query would be:

```
select from ChatMessage
where chat=desiredRoom
parameters String desiredRoom
order by date
```

Our JDOQL query (and in fact, most JDOQL queries) consists of four parts:

select from ChatMessage The select clause specifies what set of objects should be searched for 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 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 SQL: it specifies what order the objects selected by the query should be returned it. 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/a . . . at/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. I personally think that It's better this way because it separates out the different elements of the query and makes it easier for someone reading the code to understand.

10.4 Gluing the Client and Server Together

Now we just need to tell AppEngine 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 AppEngine web.xml, which is located in the war/WEB-INF directory of our AppEngine project. This file declares the servlets that are part of our application, tells AppEngine 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.ChatSubmissionService
        </servlet-class>
  </servlet>

❷ <servlet-mapping>
    <servlet-name>chatServlet</servlet-name>
    <url-pattern>/persistchat/chat</url-pattern>
  </servlet-mapping>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>PersistChat.html</welcome-file>
  </welcome-file-list>

</web-app>
```

❶ Tell AppEngine 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 applet specified in the servlets clause, give AppEngine 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 AppEngine 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 AppEngine 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 AppEngine 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 AppEngine.

Chapter 11

Building User Interfaces in Java

Part IV

Advanced AppEngine

Chapter 12

Advanced Datastore

Chapter 13

AppEngine Services

Chapter 14
Computing in the Cloud

Chapter 15

Security in AppEngine Services

Chapter 16

Wrapping Up

Index

More Books go here...