Extracted from:

Code in the Cloud

Programming Google AppEngine

The Pragmatic Programmers

Code in the Cloud

Programming Google AppEngine

Mark C. Chu-Carroll

Edited by Colleen Toporek

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Printed in the United States of America.

ISBN-10: 1-934356-63-8

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

Printed on acid-free paper.

B1.0 printing, March 17, 2010

Version: 2010-3-15

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)
        msg.put() ❶
        # Now that we've added the message to the chat, we'll redirect
        # to the root page,
        self.redirect('/')
```

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

Retrieving Persistent Objects

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

Download persist-chat/pchat.py

```
# Output the set of chat messages
messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY time") ❶
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 ChatRoomPage, and renamed the copy to ChatRoomCountViewPage. The modified get method is shown below:

Download persist-chat/pchat.py

```
class ChatRoomCountViewPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
           <html>
           <head><title>MarkCC's AppEngine Chat Room (last 20)</title>
           </head>
           <body><h1>Welcome to MarkCC's AppEngine Chat Room</h1>
           <p>(Current time is %s; viewing the last 20 messages.)</p>
           """ % (datetime.datetime.now()))
        # Output the set of chat messages
❶       messages = db.GqlQuery('SELECT * From ChatMessage ORDER BY 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.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Code in the Cloud's Home Page

http://pragprog.com/titles/mcappe

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

http://pragprog.com/updates

Be notified when updates and new books become available.

Join the Community

http://pragprog.com/community

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

http://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mcappe.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	`support@pragprog.com`
Non-English Versions:	`translations@pragprog.com`
Pragmatic Teaching:	`academic@pragprog.com`
Author Proposals:	`proposals@pragprog.com`
Contact us:	1-800-699-PROG (+1 919 847 3884)