Google App Engine Data Store

ae-10-datastore

www.appenginelearn.com

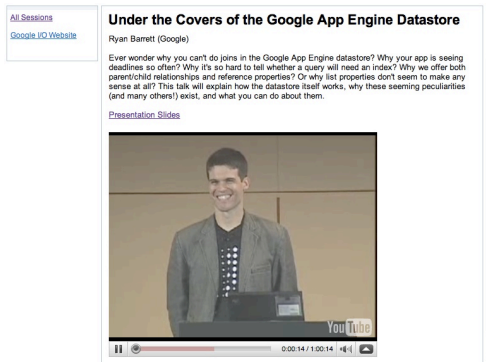
Data @ Google is BIG

- Google's main applications are their search and mail
- The entire Internet and everyone's mail is a lot of data
- Traditional data storage approaches such as a relational database just don't scale to the size at which Google applications operate

- Sharded, sorted, array with hierarchical keys

http://labs.google.com/papers/bigtable.html

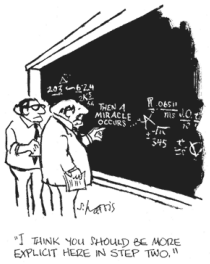
Advanced Stuff

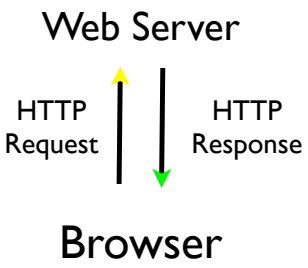


http://sites.google.com/site/io/under-the-covers-of-the-google-app-engine-datastore

Model-View-Controller

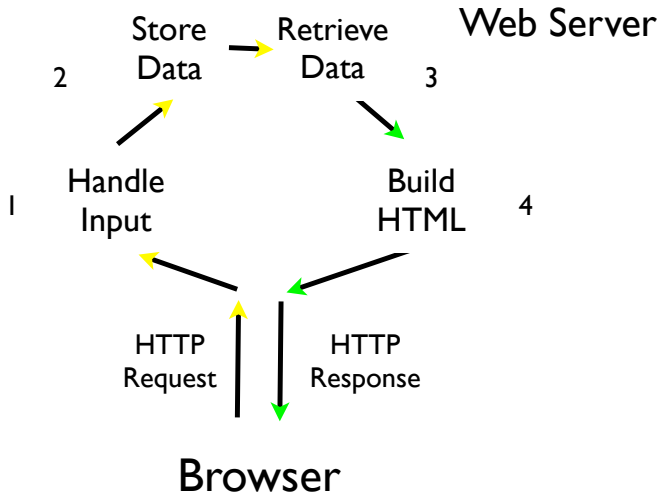
Design Pattern

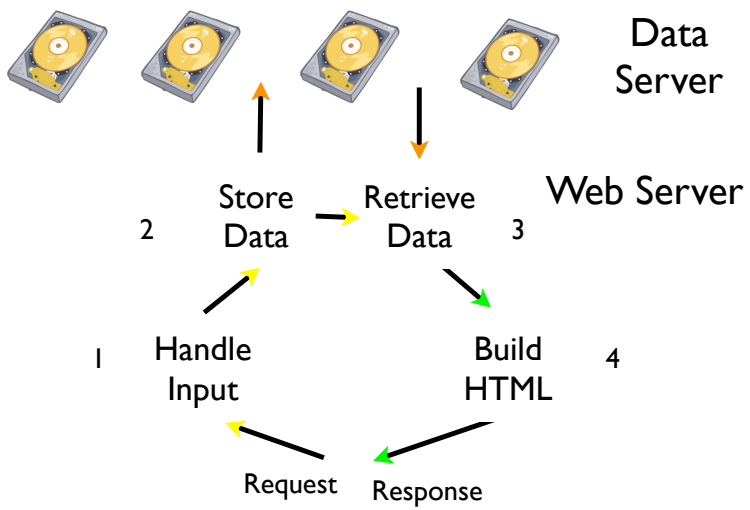




Tasks Inside the Server

- Process any form input - possibly storing it in a database or making some other change to the database such as a delete
- Decide which screen to send back to the user
- Retrieve any needed data
- Produce the HTML response and send it back to the browser





Terminology

- We call the Data bit - the "Model" or Data Model
- We call the "making the next HTML" bit the "View" or "Presentation Layer"
- We call the handling of input and the general orchestration of it all the "Controller"

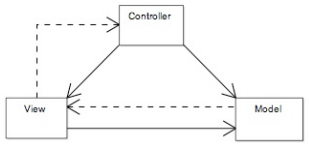
Model View Controller

- We name the three basic functions of an application as follows
  - Controller - The Python code that does the thinking and decision making
  - View - The HTML, CSS, etc. which makes up the look and feel of the application
  - Model - The persistent data that we keep in the data store

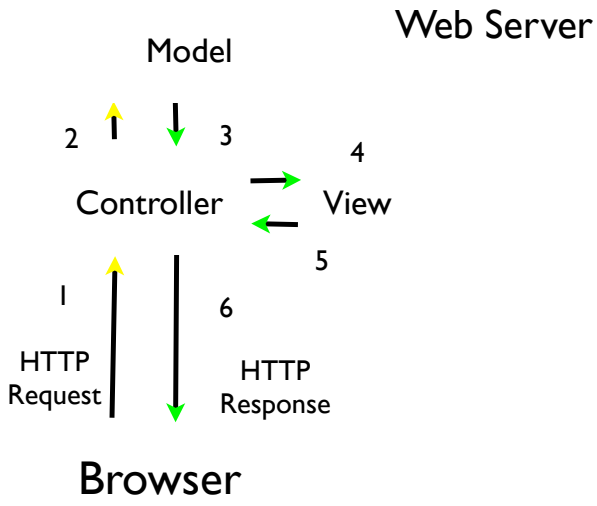
http://en.wikipedia.org/wiki/Model-view-controller

Model-View-Controller

"In MVC, the model represents the information (the data) of the application and the business rules used to manipulate the data; the view corresponds to elements of the user interface such as text, checkbox items, and so forth; and the controller manages details involving the communication to the model of user actions."



http://en.wikipedia.org/wiki/Model-View-Controller



Our Architecture: MVC

- Model - Holds the permanent data which stays long after the user has closed their web browsers
- View - Produces the HTML Response
- Controller - Receives each request and handles input and orchestrates the other elements

Controller "Orchestrates"

http://www.kapralova.org/MORAL.htm

Session

Cookies

Model

Logic

Ajax

Browser

View

The controller is the conductor of all of the other aspects of MVC.

Adding Models to our Application

ae-10-datastore

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

DJango Models

- Thankfully we use a very simple interface to define objects (a.k.a. Models) and store them in BigTable
- Google's BigTable is where the models are stored
- We don't need to know the details
- The pattern of these models is taken from the DJango project

http://docs.djangoproject.com/en/dev/ref/models/instances/?from=olddocs

A Simple Model

```
from google.appengine.ext import db

# A Model for a User
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```

Each model is a Python class which extends the db.Model class.

```
newuser = User(name="Chuck", acct="csev", pw="pw")
newuser.put()
```

Property Types

- StringProperty - Any string
- IntegerProperty - An Integer Number
- DateTimeProperty - A date + time
- BlobProperty - File data
- ReferenceProperty - A reference to another model instance

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

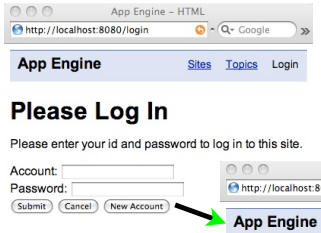
Property class	Value type	Sort order
StringProperty	str unicode	Unicode (`str` is treated as ASCII)
BooleanProperty	bool	`False` < `True`
IntegerProperty	int long	Numeric
FloatProperty	float	Numeric
DateTimeProperty DateProperty TimeProperty	datetime.datetime	Chronological
ListProperty StringListProperty	list of a supported type	If ascending, by least element; if descending, by greatest element
ReferenceProperty SelfReferenceProperty	db.Key	By path elements (kind, ID or name, kind, ID or name...)
UserProperty	users.User	By email address (Unicode)
BlobProperty	db.Blob	(not orderable)
TextProperty	db.Text	(not orderable)
CategoryProperty	db.Category	Unicode

Keep it simple for a while

```
from google.appengine.ext import db

# A Model for a User
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```

Each model is a Python class which extends the db.Model class.

```
newuser = User(name="Chuck", acct="csev", pw="pw");
newuser.put();
```

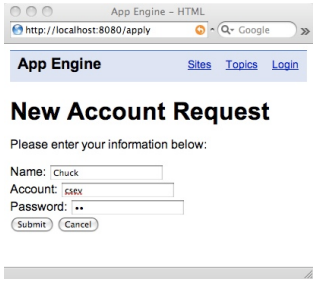


Inserting a User and listing Users

```
class ApplyHandler(webapp.RequestHandler):

    def post(self):
        self.session = Session()
        xname = self.request.get('name')
        xacct = self.request.get('account')
        xpw = self.request.get('password')

        # Check for a user already existing
        que = db.Query(User).filter("acct =",xacct)
        results = que.fetch(limit=1)

        if len(results) > 0 :
            doRender(self,"apply.htm",{'error' : 'Account Already Exists'} )
            return

        newuser = User(name=xname, acct=xacct, pw=xpw);
        newuser.put();
        self.session['username'] = xacct
        doRender(self,"index.htm",{ })
```

Get Session

Form Data



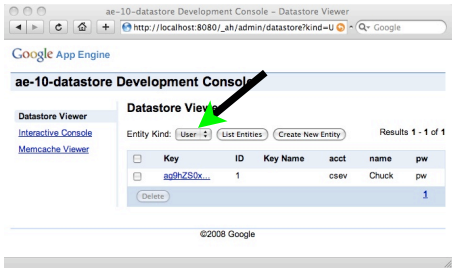
Check for existing user.

Insert User

Update Session

http://localhost:8080/_ah/admin/

Using the Developer console we can see the results of the put() operation as a new User object is now in the data store.




```
newuser = User(name=xname, acct=xacct, pw=xpw);
newuser.put();
```

```
class MembersHandler(webapp.RequestHandler):

    def get(self):
        que = db.Query(User)
        user_list = que.fetch(limit=100)
        doRender(self, 'memberscreen.htm',
                 {'user_list': user_list})
```

We simply construct a query for the User objects, and fetch the first 100 User Objects. Then we pass this list into the memberscreen.htm template as a context variable named 'user_list'.

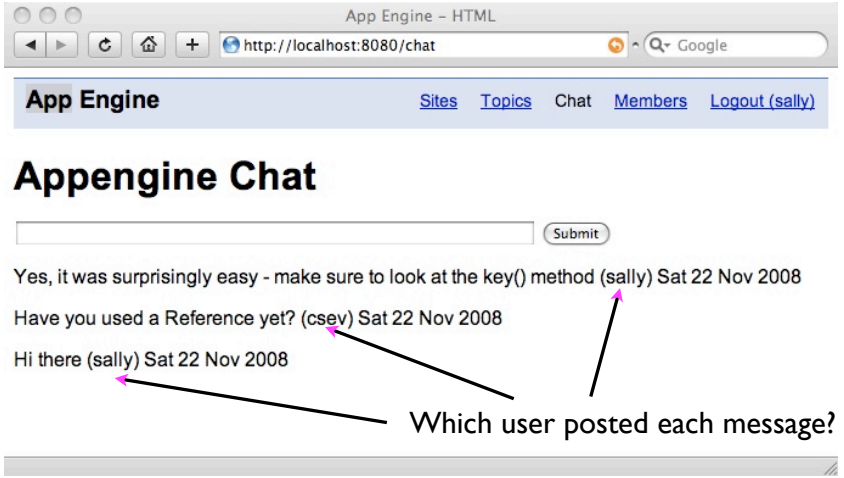
templates/members.htm

```
{% extends "_base.htm" %}
{% block bodycontent %}
    <h1>Members</h1>
    <p>
    <table>
    <tr><th>Name</th><th>Account</th><th>Password</th></tr>
    {% for user in user_list %}
       <tr>
       <td>{{ user.name }}</td>
       <td>{{ user.acct }}</td>
       <td>{{ user.pw }}</td>
       </tr>
    {% endfor %}
    </table>
{% endblock %}
```

In the template, we use the for directive to loop through each user in the user_list variable in the context. For each user we construct a table row with their name, account, and pw.

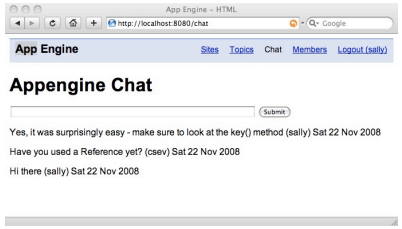
Google App Engine References

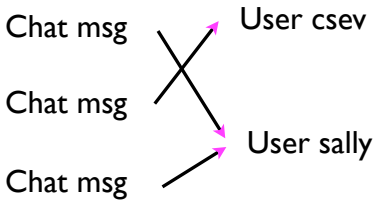
ae-11-chat



Relationships

- We need to create a new model for Chat messages and then relate Chat messages by marking them as belonging to a particular user





Three Kinds of Keys

- Logical Key - What we use to look something up from the outside world - usually unique for a model
- Primary Key - Some "random" number which tells the database where it put the data - also unique - and opaque
- Reference - When we have a field that points to the primary key of another model (a.k.a. Foreign Key)

```
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```

User

acct
pw
name

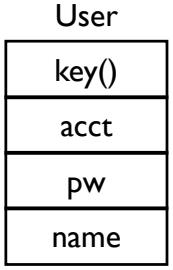
```
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```

User

key()
acct
pw
name

```
newuser = User(name=name, acct=acct, pw=pw)
newuser.put()
self.session['username'] = acct
self.session['userkey'] = newuser.key()
```

```
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```




```
newuser = User(name=name, acct=acct, pw=pw)
key = newuser.put();
self.session['username'] = acct
self.session['userkey'] = key
```

Fast Lookup By Primary Key

- Lookup by primary key is faster than by logical key - because the primary key is about "where" the object is placed in the data store and there is *only one*
- So we put it in session for later use...

```
newuser = User(name=name, acct=acct, pw=pw);
key = newuser.put();
self.session['username'] = acct
self.session['userkey'] = key
```

When we log in...

```
que = db.Query(User).filter("acct =",acct).filter("pw = ",pw)
results = que.fetch(limit=1)
if len(results) > 0 :
    user = results[0]
    self.session['username'] = acct
    self.session['userkey'] = user.key()
    doRender(self,"index.htm",{ } )
else:
    doRender(self,"loginscreen.htm",
                {'error' : 'Incorrect login data'} )
```

When we log Out...

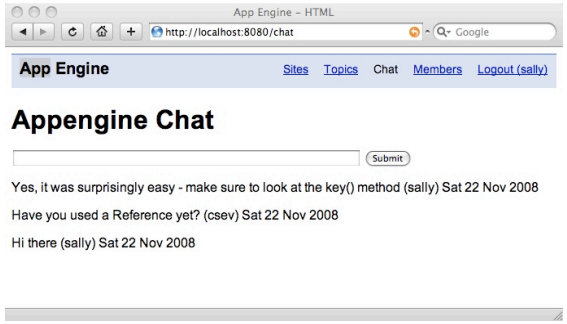
```
class LogoutHandler(webapp.RequestHandler):

    def get(self):
        self.session = Session()
        self.session.delete_item('username')
        self.session.delete_item('userkey')
        doRender(self, 'index.htm')
```

When we log out - we make sure to remove the key from the session as well as the account name.

Making References

References

- When we make a new object that needs to be associated with or related to another object - we call this a "Reference"
- Relational Databases call these "Foreign Keys"



Database Normalization

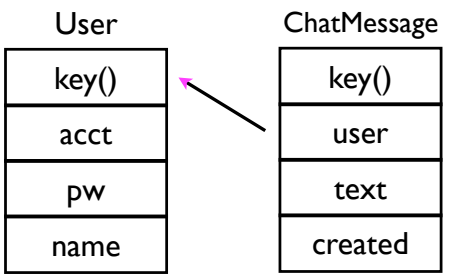
We could just store the account strings in each chat message. This is bad practice generally - particularly if we might want to know more detail about the User later. We don't like to make multiple copies of anything except primary keys.

http://en.wikipedia.org/wiki/Database_normalization

```
class ChatMessage(db.Model):
    user = db.ReferenceProperty()
    text = db.StringProperty()
    created = db.DateTimeProperty(auto_now=True)
```

So we make a reference property in our Chat message model. The property does *not* need to be named "user" - but it is a convienent pattern. Also note the created field that we let the data store auto-populate.

Relating Models



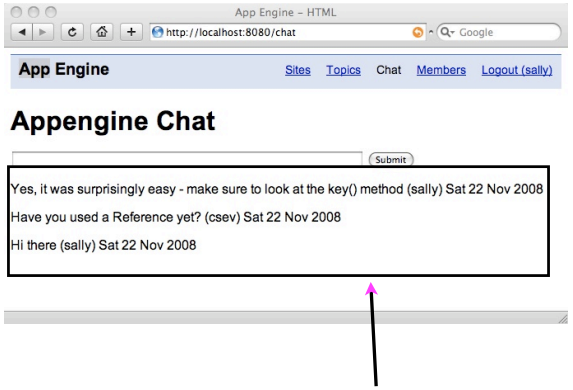

```
class User(db.Model):
    acct = db.StringProperty()
    pw = db.StringProperty()
    name = db.StringProperty()
```

```
class ChatMessage(db.Model):
    user = db.ReferenceProperty()
    text = db.StringProperty()
    created = db.DateTimeProperty(auto_now=True)
```

Populating References

```
class ChatMessage(db.Model):
    user = db.ReferenceProperty()
    text = db.StringProperty()
    created = db.DateTimeProperty(auto_now=True)

def post(self):
    self.session = Session()

    msg = self.request.get('message')
    newchat = ChatMessage(user = self.session['userkey'], text=msg)
    newchat.put();
```

When we create a ChatMessage, we get the message text from the chatscreen.htm form, and then user reference is the key of the current logged in user taken from the Session. Note: Some error checking removed from this example.



We need to display the list of the most recent ChatMessage objects on the page.

```
def post(self):
    self.session = Session()
    msg = self.request.get('message')
    newchat = ChatMessage(user = self.session['userkey'], text=msg)
    newchat.put();

    que = db.Query(ChatMessage).order("-created");
    chat_list = que.fetch(limit=10)
    doRender(self,"chatscreen.htm",
                  { 'chat_list': chat_list })
```

We retrieve the list of chat messages, and pass them into the template as context variable named "chat_list" and then render "chatscreen.htm".

ChatMessage

key()
user
text
created

chatscreen.htm

```
{% extends "_base.htm" %}
{% block bodycontent %}
    <h1>Appengine Chat</h1>
    <p>
    <form method="post" action="/chat">
    <input type="text" name="message" size="60"/>
    <input type="submit" name="Chat"/>
    </form>
    </p>
    {% ifnotequal error None %}
     <p>
     {{ error }}
     </p>
    {% endifnotequal %}
    {% for chat in chat_list %}
      <p>{{ chat.text }} ({{chat.user.acct}})
          {{chat.created|date:"D d M Y"}}</p>
    {% endfor %}
{% endblock %}
```

In the chatscreen.htm template, we loop through the context variable and process each chat message.

chatscreen.htm

```
{% extends "_base.htm" %}
{% block bodycontent %}
    <h1>Appengine Chat</h1>
    <p>
    <form method="post" action="/chat">
    <input type="text" name="message" size="60"/>
    <input type="submit" name="Chat"/>
    </form>
    </p>
    {% ifnotequal error None %}
     <p>
     {{ error }}
     </p>
    {% endifnotequal %}
    {% for chat in chat_list %}
      <p>{{ chat.text }} ({{chat.user.acct}})
          {{chat.created|date:"D d M Y"}}</p>
    {% endfor %}
{% endblock %}
```

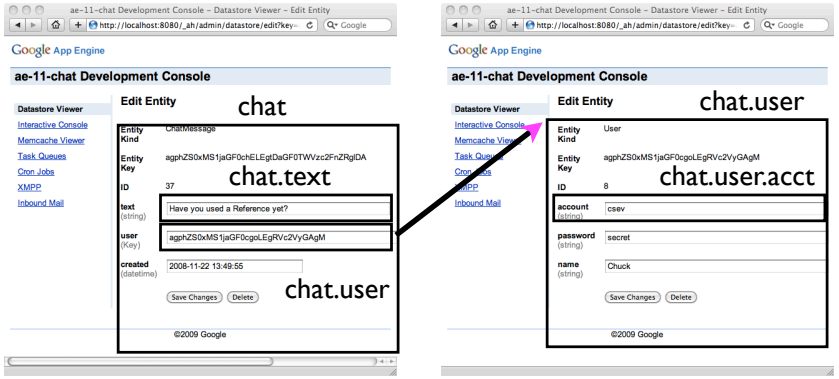
In the chatscreen.htm template, we loop through the context variable and process each chat message.

For a reference value we access the .user attribute and then the .acct attribute within the .user related to this chat message.

Walking a reference

- The chat_list contains a list of chat objects
- The iteration variable chat is each chat object in the list
- chat.user is the associated user object (follow the reference)
- chat.user.acct is the user's account

```
{% for chat in chat_list %}
  <p>{{ chat.text }} ({{chat.user.acct}})
      {{chat.created|date:"D d M Y"}}</p>
{% endfor %}
```




```
{% for chat in chat_list %}
  <p>{{ chat.text }} ({{chat.user.acct}})
      {{chat.created|date:"D d M Y"}}</p>
{% endfor %}
```

chatscreen.htm

```
{% extends "_base.htm" %}
{% block bodycontent %}
    <h1>Appengine Chat</h1>
    <p>
    <form method="post" action="/chat">
    <input type="text" name="message" size="60"/>
    <input type="submit" name="Chat"/>
    </form>
    </p>
    {% ifnotequal error None %}
     <p>
     {{ error }}
     </p>
    {% endifnotequal %}
    {% for chat in chat_list %}
      <p>{{ chat.text }} ({{chat.user.acct}})
          {{chat.created|date:"D d M Y"}}</p>
    {% endfor %}
{% endblock %}
```

To make the date format a little nicer we use a |date: formatter which shows the day of week, day of month, month, and year.

Summary

- All objects stored in the data store are given a primary key which we get from either the put() call or the key() call
- We place these keys in ReferenceProperty values to connect one model to another
- When an attribute is a reference property, we use syntax like chat.user.acct - to look up fields in the referenced object