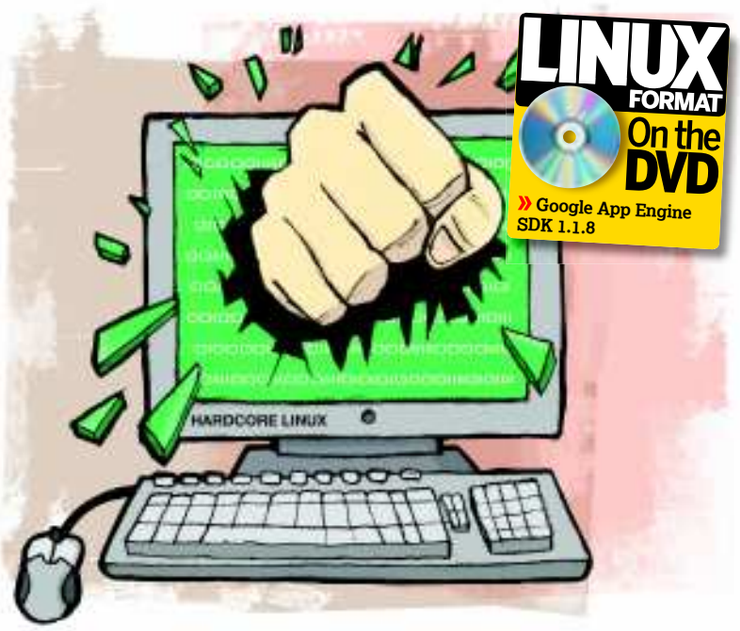
GAE: Deploy a

Google App Engine enables you to build scalable apps without worrying about scaling details. **Dan Frost** looks at how to get your first cloud app off the ground.



Google App Engine is a platform for developing applications on Google's infrastructure. Like other hugely scalable platforms, App Engine gives you a way of deploying to the cloud without the expense of running your own server farm.

Unlike other cloud computing solutions, Google App Engine is specialised – it's purely for building web applications. You can serve pages, store information and interact with external web servers, but you don't have access to files, disks and databases in the same way you would in a normal environment.

While this does mean you might have to re-engineer your application, you do have some very powerful tools to work with. Google App Engine supports Google user accounts, image manipulation, huge data stores and, using the Google data library, interaction with several Google data systems.

Google App Engine applications are currently only supported by Python. If this isn't your favourite programming language, keep an eye out for support coming to others platforms near you in the future.

Our expert

Dan Frost is technical director of 3ev, a Brighton-based web development agency. He has developed for the *TYPO3* CMS project and is currently working on the *Involve* CMS application for 3ev.

Working with App Engine

Start by working locally using a developer server – *dev_appserver.py* – which simulates the live environment. This environment, provided by the software development kit, gives you a working server, data storage, simulated user accounts and pretty much everything else you need to develop your app. Once you've made your latest Web 2.0 masterpiece, deploy the application to Google's servers using *appcfg.py*, which uploads the app to your App Engine account. (Create an account at **http://appengine.google.com**.)

It's now time to install the development environment. You'll need Python 2.5 installed, after which you can download App Engine for your OS from **http://code.google.com/appengine/downloads.html**. For Linux, unzip and add App Engine to your **$PATH**:

```
export PATH=$PATH:/path/to/google_appengine/
```

Check this worked by typing **dev_appserver.py** on the command line – you should get the usual page of help messages. Now create a directory for your application:

```
mkdir ~/myapp/
```

Next, create the key file for App Engine: **app.yaml**. This file tells App Engine where in the directory to find your application and how to treat each file. Create **~/myapp/app.yaml** containing:

```
application: mydemoapp
version: 1
runtime: python
api_version: 1
handlers:
 - url: /.*
   script: main.py
```

This tells App Engine that your application is called *mydemoapp* and that all requests should be passed to **main.py**. You can set any script to handle any URL, and you can even use patterns in the configuration to use different files in your app:

```
url: /browse/(.*?)/
script: /listings/\1.py
```

The configuration above passes everything to **main.py**, so the next step is to create this file and add the usual message:

```
print "Hello, World"
```

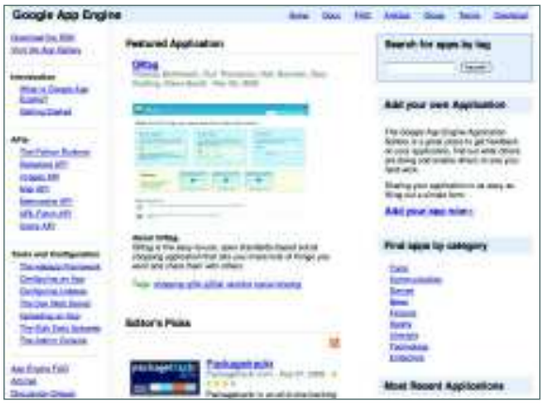
With this modest application ready to go, fire up the development server using **dev_appserver.py** and point your browser at **http://localhost:8080**.

```
dev_appserver.py  ~/myapp/
```

You can use most standard Python expressions within App Engine, with a few exceptions that mostly relate to filesystem

» **Last month** We used *Git* to manage distributed software versioning.

web application



› Shopping carts, memo pads and even SQL designers – to see some of what's possible, browse the featured apps.

access. Add some methods, classes, and even create modules and you'll find that the environment is quite familiar.

Using webapp

The App Engine environment comes pre-loaded with webapp, an MVC framework that enables you to build well-structured apps in just a few lines. Start by importing the framework with the line and creating a handler, which in webapp is just a simple class that extends **webapp.RequestHandler**:

```
from google.appengine.ext import webapp
class ExampleApp(webapp.RequestHandler):
  def get(self):
    self.response.out.write('Hello, well structured world')
```

The request handler has two important methods – **get()** and **post()**. Get is called for all **HTTP GET** requests, while post is called for all **HTTP POST** requests. Below the request handler, you need to register it with webapp and then run the webapp **main()** method:

```
application = webapp.WSGIApplication(
  [('/', ExampleApp)],
  debug=True)
  def main():
    run_wsgi_app(application)
    if __name__ == "__main__":
      main()
```

Revisit your app's URL and you'll see a rather unimpressive message. Let's improve it by moving the message into a template – create the file **index.html**:

```
<html>
<head><title>Hi there!</title></head>
<body><h1>Hello from the template</h1></body>
</html>
```

You then include this by altering the **get()** method:

```
def get(self):
  template_vars = {}
  self.response.out.write(template.render(path, template_
vars))
```

If you want to include stylesheets, JavaScript, images or any other static files, you first need to tell **app.yaml**. Add the following to the file, above the **url: /.*** handler:

```
- url: /style
  static_dir: style
```

Next, create a directory called **style** and then a file called **app.css** before including it in index.html:

```
<link rel="stylesheet" href="/style/app.css" type="text/css">
```

Add some style to your CSS file – I'll leave it for you to do so as you see fit.

Saving models

Scaling databases is tricky, but Google has found some clever ways of doing this that offer a different approach to traditional relational databases.

Google's *BigTable* is at the heart of App Engines's data storage system, which means that you can write applications that can scale to millions of users and page impressions. *BigTable* is a distributed storage system designed for managing "petabytes of data across thousands of commodity servers" (**http://labs.google.com/papers/bigtable.html**), but getting started with it in App Engine is simple. It all starts with models, which only take a few lines to create. All models in App Engine are classes that extend **db.Model**, whose properties are like fields in normal database tables. For example, a simple model might consist of:

```
class MyNote(db.Model):
  thenote = db.StringProperty(multiline=True)
  date = db.DateTimeProperty(auto_now_add=True)
```

The **date** property is automatically set to the current time thanks to **auto_now_add**, of which **thenote** is obviously a string. Models can also contain boolean, integer, floats, blob, emails and many other property types.

You don't have to do anything else before using this – just jump straight in by creating an instance of the class, setting the properties and calling **put()**:

```
note = MyNote()
note.thenote = "Just a quick note"
note.put()
```

For our example, we start by creating a model for storing comments called **Comment**: »

Logging

Get into the habit of using logger – 'import logging' at the top of your app and then use logging throughout:

```
logging.info("Something's happening...")
```

You can view logs via the dashboard – navigate to 'Logs' and drill down into the detail of each item.

Quick tip

If you're new to Python, but still want to try out Google App Engine, the first pitfall to avoid is setting your editor to use spaces instead of tabs. If you don't, App Engine will give you all kinds of colourful errors!

Quick tip

With the Google Data Services you can get lots of Google's data into your apps. Start by installing gdata in your app directory via **http://code.google.com/appengine/docs/usinggdataservices.html** and including it using **import gdata**.

```
class Comment(db.Model):
  content = db.StringProperty(multiline=True)
  date = db.DateTimeProperty(auto_now_add=True)
  author = db.UserProperty()
```

The next step is to get some data into this, list it and then add some more interesting properties to it. In order to create some comments, we need to create a form, save the data and then display it. Start by adding the form to **index.html**:

```
<form action="/" method="post" accept-charset="utf-8">
  <input type="hidden" name="parent" value="{{ comment.
key }}" />
  <textarea name="comment"></textarea>
  <input type="submit" value="Add"></div>
</form>
```

Next, add the method **post()** to the controller.

```
def post(self):
  c = Comment()
  c.content = self.request.get('comment')
  c.author = users.get_current_user()
  c.put()
  self.redirect('/')
```

GQL is used to get data back out of the data storage and, in a lot of cases, looks just like SQL. You need to replace the contents of **get()** with a call to the Comment's GQL method, which you then pass to the template:

```
comments = Comment.gql("ORDER BY date DESC ")
template_vars = {
  'comments' : comments
}
self.response.out.write(template.render('index.html',
template_vars))
```

Finally, you can loop through the comments to display the content from each:

```
{% for comment in comments %}
  <p>{{ comment.content }}</p>
{% endfor %}
```

For neatness – and because you're going to add functionality later – move the comment line into another file. Replace the second line above with **{% include 'comment.html' %}** and create a file **comment.html** that contains:

```
<div class="comment row1">
<p><strong>Posted on {{ comment.date }}</strong><br />
  {{ comment.content }} </p>
</div>
```

Using users

App Engine enables you to authenticate using existing Google accounts, so your users don't have to register for yet another webapp. This is done using the **users** package:

```
from google.appengine.api import users
```

Now you can access a user's nickname and email address using functions such as **users.get_current_user()**. If the user isn't logged in, you can redirect to the login screen:

```
if users.get_current_user():
  user = get_current_user()
  self.response.out.write("You are logged in as: " + user.
nickname())
else:
  self.redirect(users.create_login_url("/home"))
```

The argument **/home** is the URL that you want the user to come back to after logging in.

In the comments app, you can force users to log in before adding comments by redirecting them to the login screen:

```
if users.get_current_user():
  comments = Comment.gql("WHERE in_reply_to = :1
ORDER BY date DESC ", None)
  template_vars = {
    'comments' : comments
  }
  self.response.out.write(template.render('index.html',
template_vars))
else:
  message = ("<a href='%s'>Please login</a>." % users.
create_login_url("/rels"))
  self.response.out.write(message)
```

Refresh the application in your browser to see the authentication working.

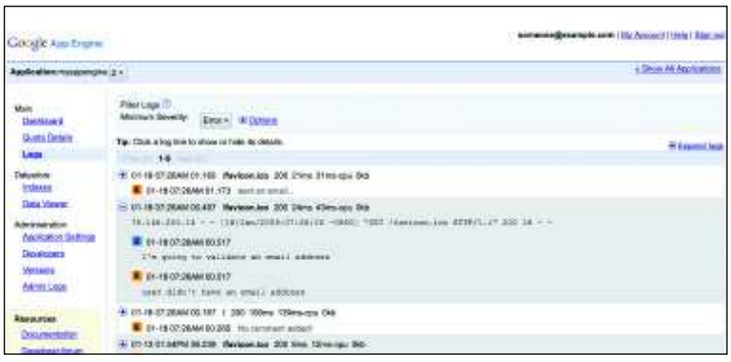
With users logged in, store each comment's author by adding the following to the **post()** method before **c.put()**:

```
c.author = users.get_current_user()
```

To display this, add some lines to **comment.html** that get the user's nickname and email address:

```
<p><b>Posted by: <b>
{% if comment.author.nickname %}
  {{ comment.author.nickname }} ({{ comment.author.email }})
{% else %}
  Anonymous
{% endif %}
on {{ comment.date }}
</p>
```

› The dashboard gives you access to hits, usage stats, logs and old versions of your app.



Relating entities

Relationships between entities are made inside the model using either **ReferenceProperty**, which refers to another model, or **SelfReferenceProperty**, which refers to the model itself. To link one model to another, do something like:

```
related_thing = db.ReferenceProperty(OtherThing)
```

If you want users to be able to reply to messages left on your noticeboard, each comment needs a **_parent_**, which we'll call **in_reply_to**. Declare this using the following, added to the Comment model:

```
in_reply_to = db.SelfReferenceProperty()
```

You can then add a form for each existing comment:

```
<div>
  <div id="comment-{{ comment.key }}" style="display : none;">
    <form action="/rels" method="post" accept-charset="utf-8">
      <input type="hidden" name="parent" value="{{ comment.key }}" />
      <textarea name="comment" rows="7" cols="30"></textarea>
      <input type="submit" value="Add">
    </form>
  </div>
  <a href="javascript:document.getElementById('comment-{{ comment.key}}').style.display='block';">Add comment</a>
</div>
```

... and then alter the **post()** method to save this:

```
if self.request.get('parent'):
  key_name = self.request.get('parent')
  p = db.get(db.Key(key_name))
  c.in_reply_to = p
else:
  c.in_reply_to = None
```

The 'parent' value from the HTML form is passed into the **post()** method and then the **db.Key()** method is used to find the object from the data store. If you view the HTML source, you'll see the keys are long strings rather than integer keys.

If you try replying to a comment, you'll see the replies appear at the bottom of this, so your next step is to display the comments hierarchically. You need to update the GQL, add a **comment.html** and add a method to the Comment model. Starting with the model, add **get_replies**:

```
def get_replies(self):
  comments = Comment.gql("where in_reply_to = :1 ORDER BY date DESC LIMIT 10", self)
  return comments
```

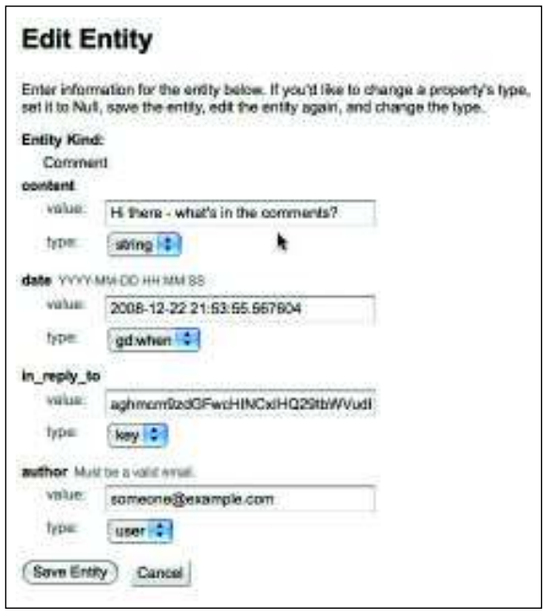
This returns all the comments that are in reply to the comment object **self**. The top-level comments should be those that don't have **in_reply_to** filled in, so we change the GQL in **get()** using Python's **None** constant:

```
comments = Comment.gql("WHERE in_reply_to = :1 ORDER BY date DESC ", None)
```

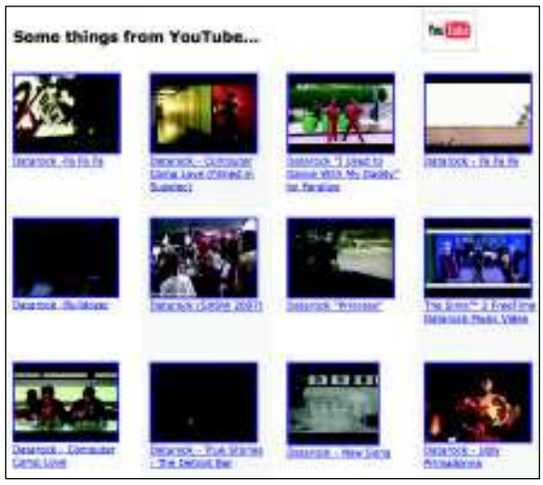
If you refresh now you'll see that only the top-level comments are displayed. The last step is to make **comment.html** hierarchical:

```
<p>
  {% for comment in comment.get_replies %}
    {% include 'comment.html' %}
  {% endfor %} </p>
```

Refresh to see the comments nested. Click on 'Add Comment' to reply to any of the comments.



› You can browse entities created by your users via the dashboard, so you can find out how the app is being used.



› You can pull all sorts of Google data into your GAE app: Base, Calendars, Documents, Contacts, YouTube and much more.

Uploading your app

Uploading is done using the second CLI tool, called **appcfg.py**. All you need to do is set the options for **update** and the location of your app.

```
~/appEngineProject/ $ appcfg.py update helloworld/
Loaded authentication cookies from /Users/you/.appcfg_cookies
Scanning files on local disk.
Initiating update.
Email: some_user@gmail.com
Password for some_user@gmail.com:
Saving authentication cookies to /Users/you/.appcfg_cookies
Cloning 21 application files.
Uploading 5 files.
Closing update.
Uploading index definitions
```

You'll be asked for your Google account details, and then the app will be uploaded to Google's servers. LXF

Indexes

Like any database-driven application, when things start to get big you need to add indexes. This is done via **index.yaml** in your project, which will be created for you. You'll need indexes when you start sorting entities, filtering over groups of entities and more complex queries. Start here: **http://code.google.com/appengine/docs/python/datastore/queriesandindexes.html**.

» **Next month** Stop processes hogging your precious bandwidth with Trickle.