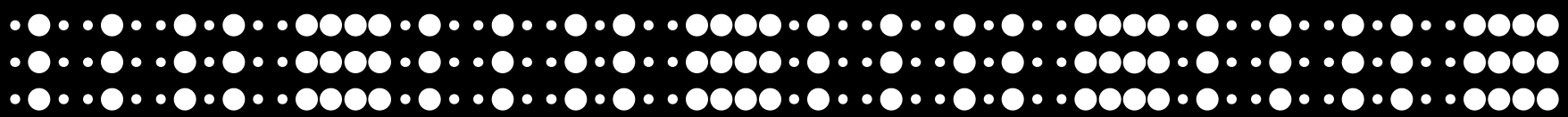

GOOGLE
I/O



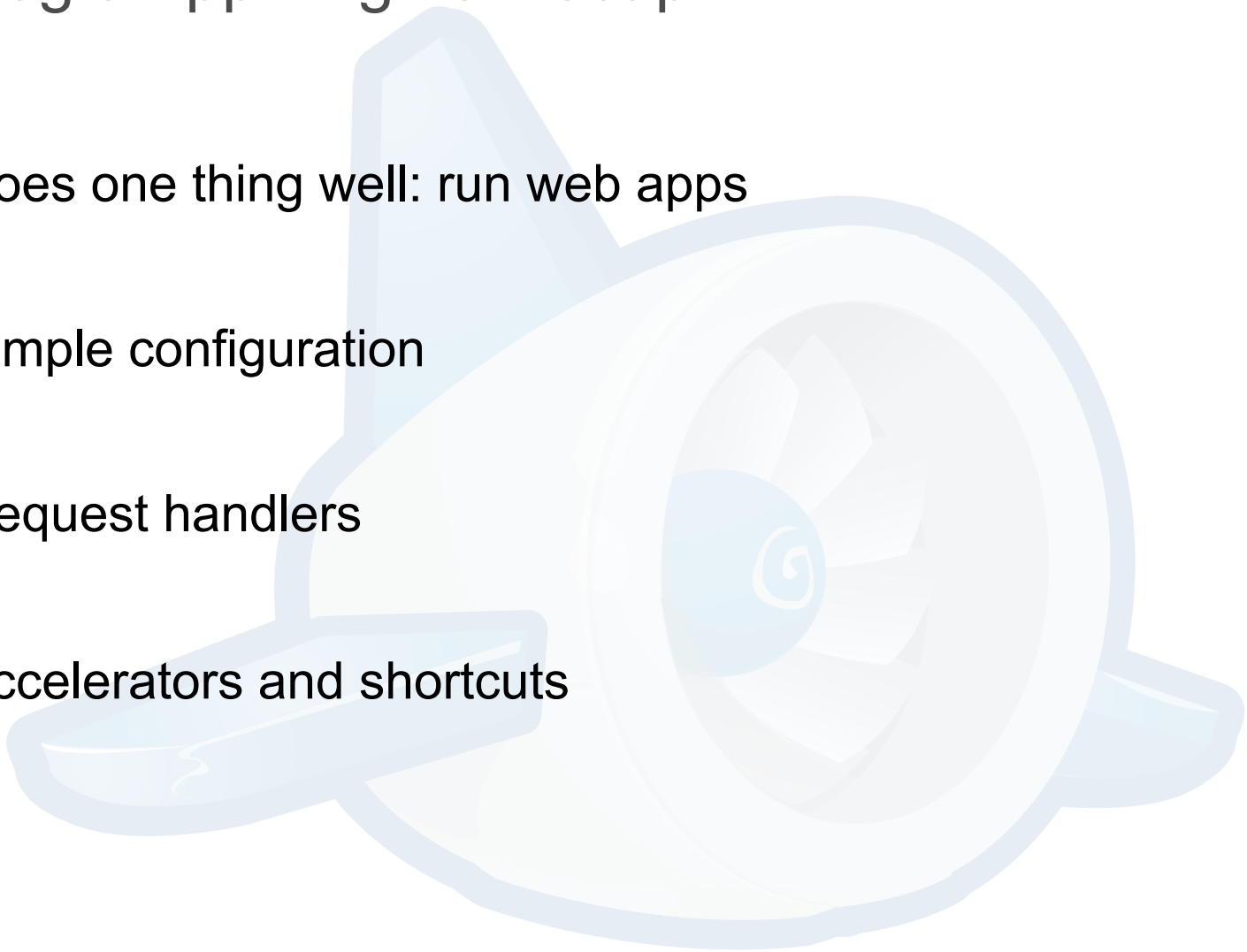
Rapid Development with Django and App Engine

Guido van Rossum
May 28, 2008

Talk Overview

- This is not a plug for Python, Django or Google App Engine
  - Except implicitly :)
- Best practices for using Django with App Engine
  - Project setup
  - The "Google App Engine Django Helper" module
  - Using App Engine's db.Model instead of Django's Model class
  - Writing unit tests
- Q & A
  - Anything goes!

Google App Engine Recap



- Does one thing well: run web apps
- Simple configuration
- Request handlers
- Accelerators and shortcuts

App Engine Does One Thing Well

- App Engine handles HTTP requests, nothing else
  - Think RPC: request in, processing, response out
  - Works well for the web and AJAX; also for other services
- Resources are scaled automatically
  - Requests may go to the same process, serially
  - Requests may go to different processes, in parallel or serially
- Highly scalable datastore based on Bigtable
  - Not SQL; no joins

Configuring an App Engine Application

- An application is a directory with everything underneath it
  - Symbolic links are followed!
- Single file app.yaml in app root directory
  - Defines application metadata
  - Maps URL patterns (regular expressions) to request handlers
  - Separates static files from program files
- Dev server (SDK) emulates deployment environment

Request Handlers

- URL patterns are mapped to request handlers by app.yaml
- Handler is invoked like a CGI script
- Environment variables give request parameters
  - E.g. PATH_INFO, QUERY_STRING, HTTP_REFERER
- Write response to stdout
  - Status (optional), headers, blank line, body

Request Handler Accelerators and Shortcuts

- CGI doesn't mean slow!
- Define a main() function
  - Module will remain loaded, main() called for each requests
  - Can use module globals for caching
  - Must use "if __name__ == '__main__': main()" boilerplate
- CGI doesn't mean clumsy!
- WSGI support layered on top of CGI: util.run_wsgi_app(app)

Django Project Setup for App Engine

- Django is a bit finicky about project structure
  - Code in one or more 'app' (application) subdirectories
    - Cannot have the same name as the project
  - Must have a single settings.py file
    - Found via DJANGO_SETTINGS_MODULE environment variable

- App Engine *vs.* Django
  - No SQL; must use App Engine's own database
  - Top-level directory appended at *end* of sys.path
  - Django 0.96.1 preloaded, which is pretty old
  - Slight differences between dev_appserver and real deployment

Boilerplate Files, Standard Project Lay-out

- Boilerplate files
  - These hardly vary between projects
  - app.yaml: direct all non-static requests to main.py
  - main.py: initialize Django and send it all requests
  - settings.py: change only a few settings from defaults

- Project lay-out
  - static/*: static files; served directly by App Engine
  - myapp/*.py: app-specific python code
    - urls.py, views.py, models.py, tests.py, and more
  - templates/*.html: templates (or myapp/templates/*.html)

Minimal app.yaml

```
application: myapp              # .appspot.com
version: 1

runtime: python
api_version: 1

handlers:
- url: /static
  static_dir: static
- url: .*
  script: main.py
```

Contents and Purpose of main.py

- Set DJANGO_SETTINGS_MODULE environment variable
  - Must do before importing Django
- Arrange for *your* version of Django to be loaded
- Monkey-patch some parts of Django (optional)
- Arrange to log all tracebacks using the logging module
- Define main() to invoke Django to handle one request

The Most Minimal main.py

```
import os
from google.appengine.ext.webapp import util


os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
from django.core.handlers import wsgi


def main():
    app = wsgi.WSGIHandler()
    util.run_wsgi_app(app)


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

Bare Minimal settings.py

```
import os

DEBUG = os.environ['SERVER_SOFTWARE'].startswith('Dev')

INSTALLED_APPS = (
  'myapp',
  )

MIDDLEWARE_CLASSES = ()

ROOT_URLCONF = 'myapp.urls'

TEMPLATE_CONTEXT_PROCESSORS = ()

TEMPLATE_DEBUG = DEBUG

TEMPLATE_DIRS = (
  os.path.join(os.path.dirname(__file__), 'templates'),
  )

TEMPLATE_LOADERS = (
  'django.template.loaders.filesystem.load_template_source',
  )

TZ = 'UTC'
```

Google App Engine Django Helper

- Separate open source project
  - By two Googlers, Matt Brown and Andy Smith
  - http://code.google.com/p/google-app-engine-django/
- Takes care of monkey-patching Django
  - Loads newer version of Django if available in app root dir
- Enables using standard Django project management tool
  - For unit tests and fixture loading (development only)
- Comes with boilerplate app.yaml, main.py, settings.py

Getting Started

- Download and unzip appengine_helper_for_django_rN.zip
- Rename directory to 'mysite' or whatever you like
- This becomes your project root; contents:
  - COPYING, KNOWN_ISSUES, Makefile, README
  - __init__.py (empty)
  - app.yaml (edit application: only)
  - appengine_django/... (helper code lives here)
  - main.py (generic bootstrap)
  - manage.py (Django management script)
  - settings.py (edit settings here)
  - urls.py (edit URL mappings here)

Changing the Standard Set-up

- Edit app.yaml to set your application id (can do this later)
- Create subdirectory myapp
- Edit settings.py, adding 'myapp' to INSTALLED_APPS
- In myapp, create:
  - __init__.py (empty)
  - views.py (add your view code here)
  - models.py (add your model definitions here)
  - tests.py (add your unit tests here)
- To use Django HEAD, copy the django package here
- /usr/local/google_appengine must be your App Engine SDK

Your First View

- Edit urls.py:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('myapp.views',
  (r'^$', 'index'),
  # ...more here later...
)
```

- Edit myapp/views.py:

```
from django.http import HttpResponse
def index(request):
    return HttpResponse('Hello world')
```

- Run it: `./manage.py runserver`
- Point your browser to: `http://localhost:8080`

Using Models

- Edit models.py:

```
from google.appengine.ext import db
from appengine_django.models import BaseModel
class Shout(BaseModel):      # subclass of db.Model
  title = db.StringProperty(required=True)
  text = db.TextProperty()
  mtime = db.DateTimeProperty(auto_now_add=True)
  user = db.UserProperty()
```

- Up next:
  - Write views to use models
  - Use forms derived from your models, with templates

Using Forms

- I tend to put forms in views.py; do as you please

```
from django.shortcuts import render_to_response
from google.appengine.ext.db import djangoforms
import models
class ShoutForm(djangoforms.ModelForm):
    class Meta:
        model = models.Shout
        exclude = ['mtime', 'user']
query = models.Shout.gql("ORDER BY mtime DESC")
def index(request):
    return render_to_response('index.html',
                {'shouts': query.run(),
                 'form': ShoutForm()})
```

Using Templates

- Put this in templates/index.html

```
{% for shout in shouts %}
<p>{{ shout.user }} : {{ shout.title }} <br>
{{ shout.text }} <br>
{{ shout.mtime|timesince }}<hr> </p>
{% endfor %}
<form method="POST" action="/post/">
<table>
{{ form }}
<tr><td><input type="submit"></td></tr>
</table>
</form>
```

The post() Method

- Add to urls.py:

```
(r'^post/$', 'post'),
```

- Add to views.py:

```
from google.appengine.api import users
from django.http import HttpResponseRedirect

def post(request):
    form = ShoutForm(request.POST)
    if not form.is_valid():
        return render_to_response('index.html',
                                  {'form': form})
    shout = form.save(commit=False)
    shout.user = users.get_current_user()
    shout.put()
    return HttpResponseRedirect('/')
```

Writing Tests

- Python has two test frameworks:

- doctest.py
  - Tests are interactive sessions captured in doc strings
  - Really easy to get started
  - Some caveats (output must be reproducible)

- unittest.py
  - Tests are methods in classes derived from unittest.TestCase
  - Modeled after JUnit

Running Tests

- Use whichever test framework you like

- `./manage.py test myapp`

- Runs all tests found in myapp

- Specifically:
  - looks for doctest and unittest tests
  - looks in models.py and tests.py

Example Doc Tests

- Put this in myapp/tests.py

```
r"""

>>> from django.test.client import Client
>>> from models import *

>>> c = Client()
>>> r = c.get('/')
>>> r.status_code
200
>>> r.content
'\n\n<form ...</form>\n'

>>> s = Shout(title='yeah', text='hello world')
>>> key = s.put()
>>> r = c.get('/')
>>> r.status_code
200
>>> assert 'yeah' in r.content
>>>

"""
```

Test Fixtures

- ./manage.py: manages the datastore used
- ./manage.py runserver: uses a persistent datastore in /tmp
- ./manage dumpdata: writes datastore contents to stdout
  - Format can be JSON (default) or XML (not fully supported yet)
- ./manage loaddata: loads datastore from a file
- Fixtures can also be added to TestCase classes:

```
from django.test import TestCase
class MyTestCase(TestCase):
    fixtures = ['test_data.json', 'more_test_data']
    def testSomething(self):
        ...
```

Q & A

- Your turn!


GOOGLE
I/O