

django

An Introduction



Safe Hammad
Python Sheffield
22nd February 2011

Look, Ma! No Django!

```
from BaseHTTPServer import *

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        path = [i for i in self.path.split('/') if i]
        result = sum(int(i) for i in path)
        html = '<html>{0}={1}</html>'.format('+'.join(path), result)
        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

Extract information from the URL

```
from BaseHTTPServer import *

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):

        # Extract information from the URL
        path = [i for i in self.path.split('/') if i]

        result = sum(int(i) for i in path)
        html = '<html>{0}={1}</html>'.format('+'.join(path), result)
        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

Do something useful with that information

```
from BaseHTTPServer import *

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        path = [i for i in self.path.split('/') if i]

        # A unit of work using that information
        result = sum(int(i) for i in path)

        html = '<html>{0}={1}</html>'.format('+'.join(path), result)
        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

Format some HTML

```
from BaseHTTPServer import *

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        path = [i for i in self.path.split('/') if i]
        result = sum(int(i) for i in path)

        # Format some HTML
        html = '<html>{0}={1}</html>'.format('+'.join(path), result)

        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

Return HTML to the web browser

```
from BaseHTTPServer import *

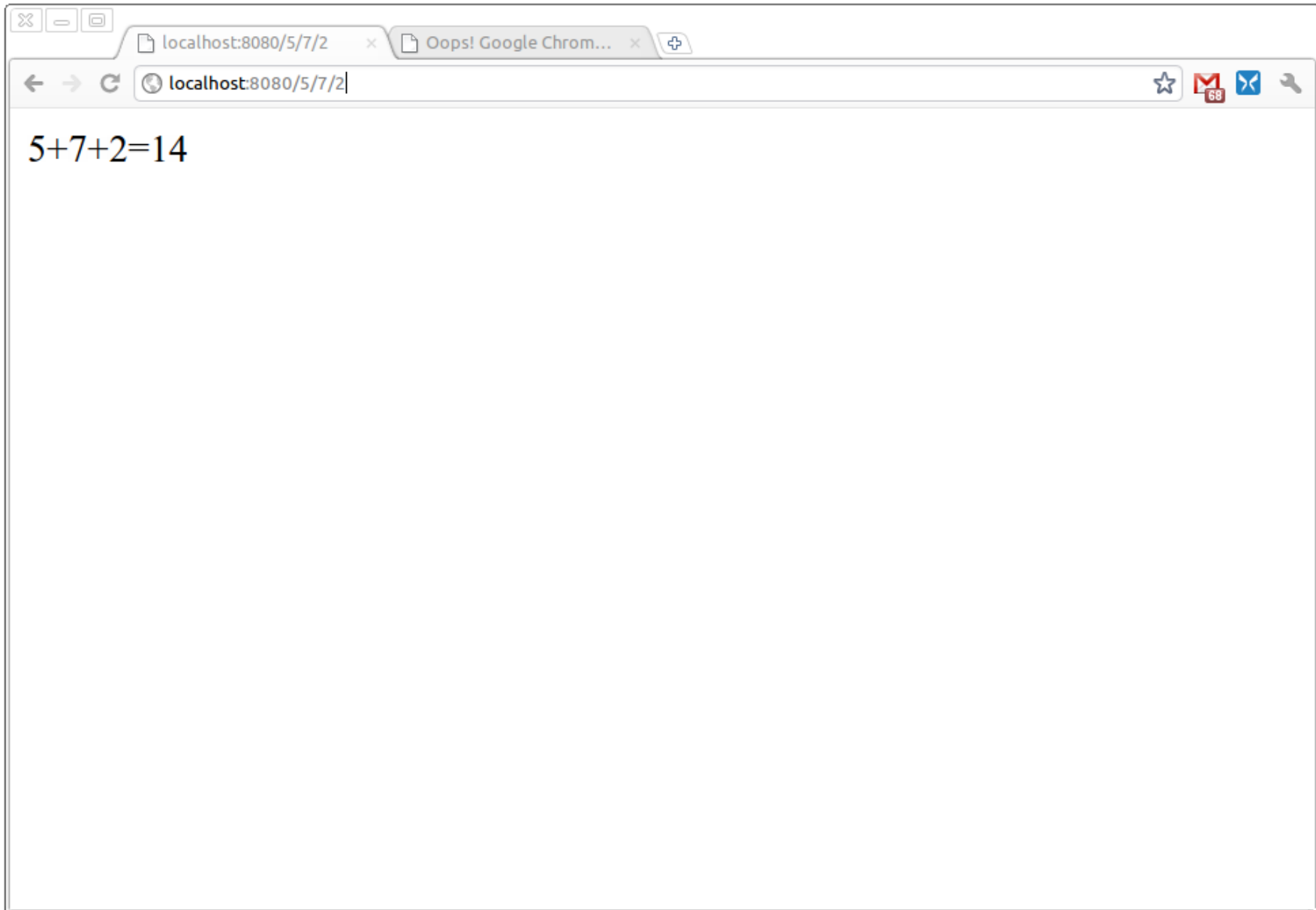
class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        path = [i for i in self.path.split('/') if i]
        result = sum(int(i) for i in path)
        html = '<html>{0}={1}</html>'.format('+'.join(path), result)

        # Return HTML to the web browser
        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

A Django-free web app



But what if we want to do
something a bit more
complicated?

Look, Ma! No Django :/

```
from BaseHTTPServer import *

class MyServer(BaseHTTPRequestHandler):

    def do_GET(self):
        path = [i for i in self.path.split('/') if i]
        action, values = path[0], path[1:]
        if action == 'add':
            try:
                result = sum(int(i) for i in values)
                html =
'<html><head><title>Addition</title></head><body>{0}={1}</body></html>'.format('+'.join(values), result)
            except ValueError:
                html = '<html><head><title>Error</title></head><body>Only integers are accepted for
addition.</body></html>'
        elif action == 'subtract':
            try:
                result = int(values[0]) - sum(int(i) for i in values[1:])
                html =
'<html><head><title>Subtraction</title></head><body>{0}={1}</body></html>'.format('-'.join(values), result)
            except ValueError:
                html = '<html><head><title>Error</title></head><body>Only integers are accepted for
subtraction.</body></html>'
        else:
            html = '<html><head><title>Error</title></head><body>Unknown operator:
{0}</body></html>'.format(action)

        self.send_response(200, 'OK')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(html)

if __name__ == "__main__":
    HTTPServer(('', 8080), MyServer).serve_forever()
```

It gets a bit messy!

We need a framework!

What do we want from a framework?

1. Call our code based on URL.
2. Template our HTML.
3. Help organise our code.

Oh ... and a bit of help with databases would be really nice!

Enter the Django!

“Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.”

Installation

- `pip install Django`
- `python setup.py install`
- `svn co http://code.djangoproject.com/svn/django/trunk/`

<http://docs.djangoproject.com/>



Create a project

- “A collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.”
- A project is effectively “your web site” with all the settings and code that go with it.

```
$ django-admin.py startproject mysite
```

Project files created

mysite/

__init__.py # makes mysite a package

manage.py # admin script

settings.py # project-wide settings

urls.py # url dispatch

The settings.py file

- It defines project settings as key/value pairs.
- The keys are well documented in the file.
- It's mandatory for all projects.
- It's just a Python module which you can edit!
- Important keys:
 - DATABASES
 - TEMPLATE_DIRS
 - INSTALLED_APPS

Run the development server

- Lightweight server with minimal/zero configuration.
- Automatically restarts with code changes.
- Not to be used in production environments!

```
$ ./manage.py runserver
```

```
Validating models...
```

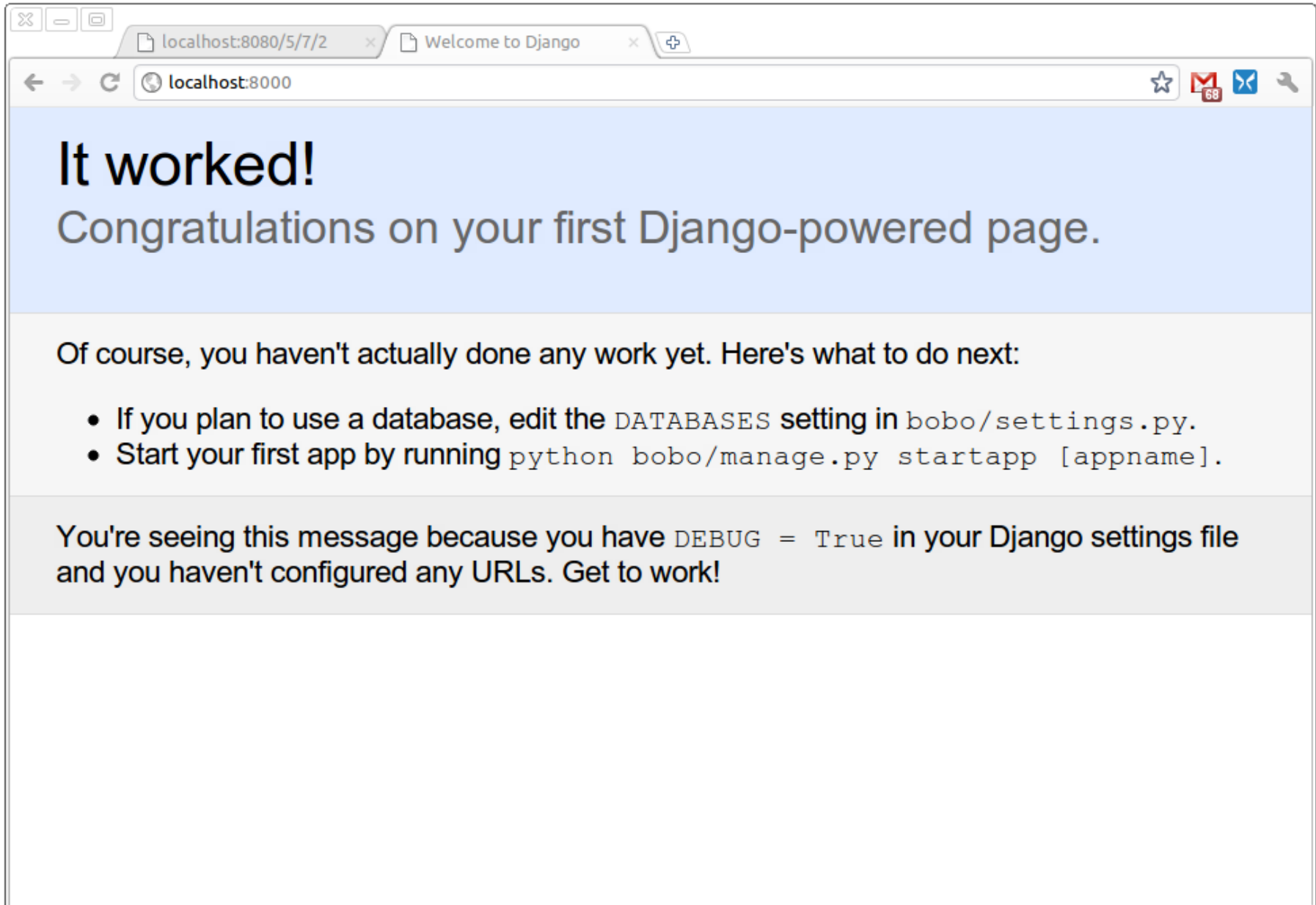
```
0 errors found
```

```
Django version 1.2.4, using settings 'mysite.settings'
```

```
Development server is running at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

Our first Django application!



Create an application

- A collection of data and code that exist to solve a single problem.
- Projects are composed of one or more applications.
- An application can be reused across multiple projects.

```
$ ./manage.py startapp myapp
```

Application files created

mysite/

`__init__.py`

`manage.py`

`settings.py`

`urls.py`

myapp/

`__init__.py`

`# makes myapp a package`

`models.py`

`# contains database models`

`tests.py`

`# contains tests`

`views.py`

`# contains app code!`

Add the application to settings.py

```
# mysite/settings.py
```

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'mysite.myapp',  
)
```

Update the DATABASES setting

```
# mysite/settings.py
```

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': 'data.sqlite',  
        'USER': '',  
        'PASSWORD': '',  
        'HOST': '',  
        'PORT': '',  
    }  
}
```

Models

- A model defines the objects you want to store and retrieve from the database.
- A model is a Python class which primarily holds field names and field types for each object.
- Models are declared in Python meaning you don't have to worry about SQL! Yay!

Create a model

```
# mysite/myapp/models.py

from django.db import models

class Author(models.Model):

    first_name = models.CharField(max_length=64)
    last_name = models.CharField(max_length=64)
    date_of_birth = models.DateField()
```

Sync the model to the database

- “Syncing” means telling the database about your model i.e. the necessary tables are created in the database.
- Every time a new model is added it must be synced to the database.

```
$ ./manage.py syncdb
```

```
...
```

```
Creating table myapp_author
```

```
...
```

The Model API

Create objects and save them to the database:

```
$ ./manage.py shell
```

```
>>> from mysite.myapp.models import Author
>>> from datetime import datetime
```

```
>>> # Create and save a new Author
```

```
>>> a = models.Author(first_name='Safe',
                       last_name='Hammad',
                       date_of_birth=datetime(1984, 1, 1))
```

```
>>> a.save()
```

```
>>> # Create and save another Author
```

```
>>> a = models.Author(first_name='Astrid',
                       last_name='Gilberto',
                       date_of_birth=datetime(1954, 1, 1))
```

```
>>> a.save()
```

The Model API (cont)

Retrieve objects back from the database:

```
>>> # The "objects" field is used for queries
>>> print Author.objects
<django.db.models.manager.Manager object at 0x1e66ad0>
```

```
>>> # Query for all authors
>>> authors = Author.objects.all()
>>> print authors
[<Author: Author object>, <Author: Author object>]
>>> print authors[0].first_name
Safe
```

```
>>> # Query for a subset of authors
>>> authors = Author.objects.filter(last_name='Gilberto')
>>> print authors
[<Author: Author object>]
>>> print authors[0].first_name
Astrid
```

Add methods to your model

A model is just a Python class:

```
from django.db import models

class Author(models.Model):

    first_name = models.CharField(max_length=64)
    last_name = models.CharField(max_length=64)
    date_of_birth = models.DateField()

    def __unicode__(self):
        """Return a string representation of this author."""
        return '{0} {1}'.format(self.first_name, self.last_name)

>>> # Query for all authors
>>> authors = Author.objects.all()
>>> print authors
[<Author: Safe Hammad>, <Author: Astrid Gilberto>]
```

Views

- This is where your application logic goes!
- Views are just functions which do something useful based on a user request then return a response to the user.
- It can be useful to think of each view as corresponding to a web page, though this won't always be the case.
- Views are usually defined in the `views.py` file.

Create a view

```
# mysite/myapp/views.py

from django.http import HttpResponse

from mysite.myapp.models import Authors

def list_authors(request):
    """Return a list of all authors to the user."""

    # Use the Model API to retrieve the list of authors
    authors = Author.objects.all()

    # Create some HTML
    html = '<html>{0}</html>'.format(''.join(str(a) for a in authors))

    # Return the HTML to the user
    return HttpResponse(html)
```

Q. So how does our view get called when the user navigates to a web page?

A. We hook up a URL to a view in the `urls.py` file!

The urls.py file (aka URLconf)

The urls.py file pairs URL patterns with view functions:

```
# mysite/urls.py

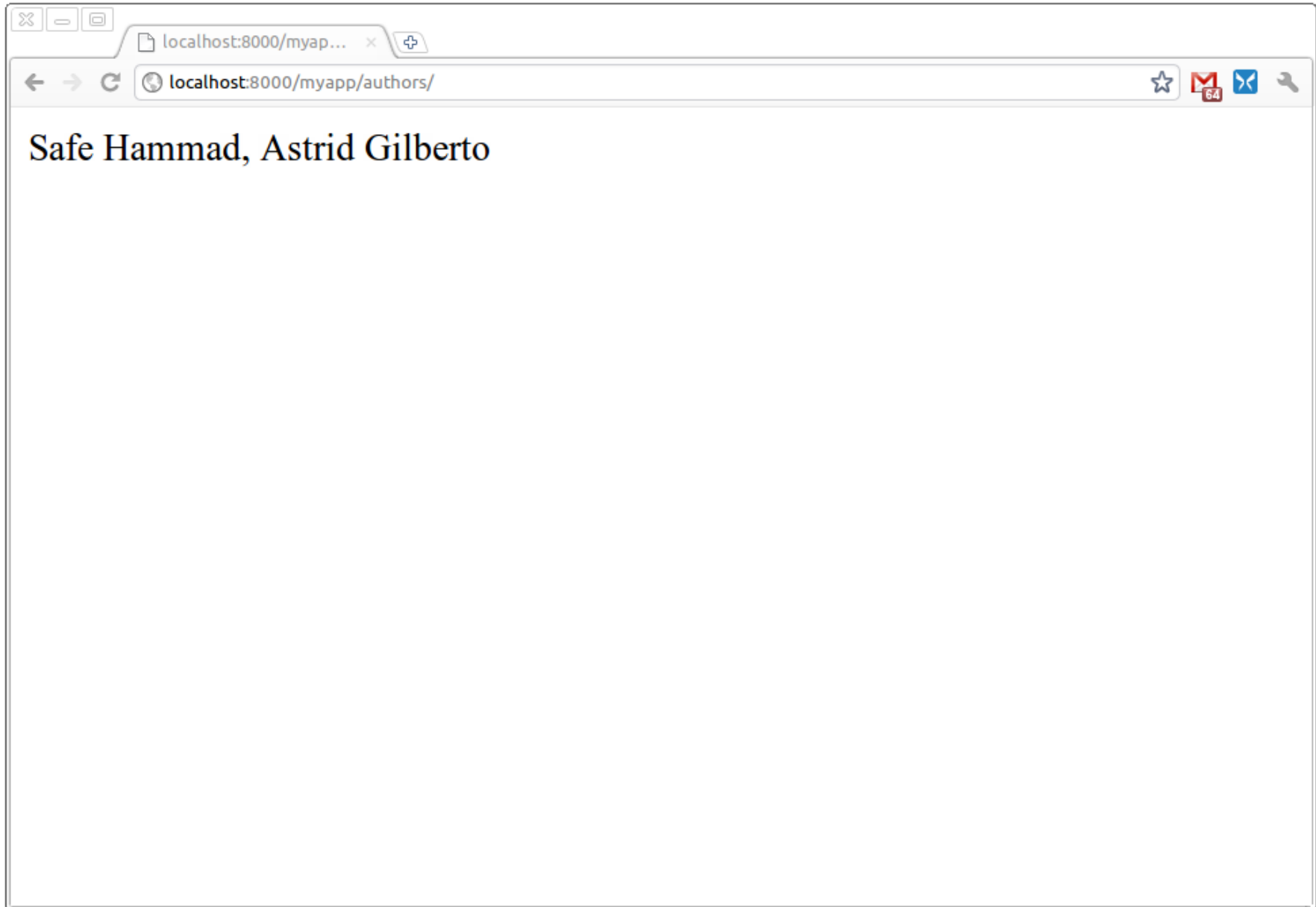
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^myapp/authors/$', 'mysite.myapp.views.list_authors'),
)
```

Regular expression
matching URL.
N.B. Excludes domain.

The view to be called as a
result of URL being matched.

Our first (real) Django application!



Other things we can do in urls.py

Capture values from the URL:

```
# mysite/urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^myapp/authors/$', 'mysite.myapp.views.list_authors'),
    (r'^myapp/authors/(<?P<last_name>\w+)/$',
     'mysite.myapp.views.filter_authors'),
)
```

Captured values in the view

Values captured in `urls.py` are presented to the view function as arguments.

```
# mysite/myapp/views.py

def filter_authors(request, last_name):
    """Return a list of all authors based on last name."""

    # Use the Model API to retrieve the subset of authors
    authors = Author.objects.filter(last_name=last_name)

    # Create some HTML
    html = '<html>{0}</html>'.format(', '.join(str(a) for a in authors))

    # Return the HTML to the user
    return HttpResponse(html)
```

Improvements to URLconf: Decoupling app from project

Create a `urls.py` in your app and reference it the project `urls.py`:

```
# mysite/urls.py
urlpatterns = patterns('',
    (r'^myapp/', include('myapp.urls')),
)
```

```
# mysite/myapp/urls.py
urlpatterns = patterns('',
    (r'^authors/$', 'myapp.views.list_authors'),
    (r'^authors/((<?P<last_name>\w+)/$', 'myapp.views.filter_authors'),
)
```



Templating

1. We'd like to cleanly generate our HTML.
2. And avoid the need to edit Python code when only the HTML needs updating.

```
# mysite/myapp/views.py

def list_authors(request):
    """Return a list of all authors to the user."""
    authors = Author.objects.all()

    # Create some HTML
    html = '<html>{0}</html>'.format(', '.join(str(a) for a in authors))

    return HttpResponse(html)
```

What do we want from templating?

1. Put the HTML template in its own file.
2. Pass in any data needed by the template via a Context object.
3. The convention is to give templates a .html extension and place them under a directory corresponding to the app name e.g.
myapp/authors.html

Templating code

```
# mysite/myapp/views.py
```

```
from django import template
```

```
def list_authors(request):
```

```
    """Return a list of all authors to the user."""
```

```
    authors = Author.objects.all()
```

```
    t = template.loader.get_template('myapp/authors.html')
```

```
    c = template.Context({'authors': authors})
```

```
    return HttpResponse(t.render(c))
```

Better templating code

Or more concisely ...

```
# mysite/myapp/views.py
```

```
from django.shortcuts import render_to_response
```

```
def list_authors(request):
```

```
    """Return a list of all authors to the user."""
```

```
    authors = Author.objects.all()
```

```
    return render_to_response('myapp/authors.html', {'authors': authors})
```

So where do template files live?

```
# mysite/settings.py
```

```
TEMPLATE_DIRS = (
```

```
    # Put strings here
```

```
    # Always use forward slashes, even on Windows.
```

```
    # Absolute paths, not relative paths.
```

```
     '/home/safe/projects/mysite/templates',
```

```
)
```

```
# os.path.join(os.path.dirname(__file__), 'templates'),
```

Our first template!

A Django template is nothing more than a standard HTML text file with some Django template tags.

```
# mysite/templates/myapp/authors.html
```

```
<html>
  <body>
    <ul>

      {% for author in authors %}
        <li>{{ author.first_name }} {{ author.last_name }}</li>
      {% endfor %}

    </ul>
  </body>
</html>
```

Django template variables

- `{{ xxx }}` are variables either pulled from the context or created within the template.

e.g.: `{{ author }}` will be replaced by the value of the author variable in the HTML file.

- The dot “.” can be used to do attribute lookups on variables.

For example, `{{ author.first_name }}` will print out the “first_name” attribute of author. In fact, `author.first_name`, `author['first_name']` and `author.first_name()` will be tried in turn until a value is found.

Context: `{'author': Author(first_name='Safe', last_name='Hammad')}`

Template: `{ a.first_name }`

Output: `Safe`

Django template tags

- `{% xxx %}` are tags and are used to control flow, perform loops or perform some logic in general, a bit like Python keywords.

For example, `{% for x in y %}{% endfor %}` will loop over items in `y`.

Context: `{'authors': authors}`

Template: `{% for a in authors %}`
`<p>{ a.first_name }</p>`
`{% endfor %}`

Output: Safe
Astrid

Django template filters

- Filters can be applied to variables to modify them for display by using the pipe “|” symbol after the variable name.

For example, `{{ author.name|upper }}` will print the author name in upper case.

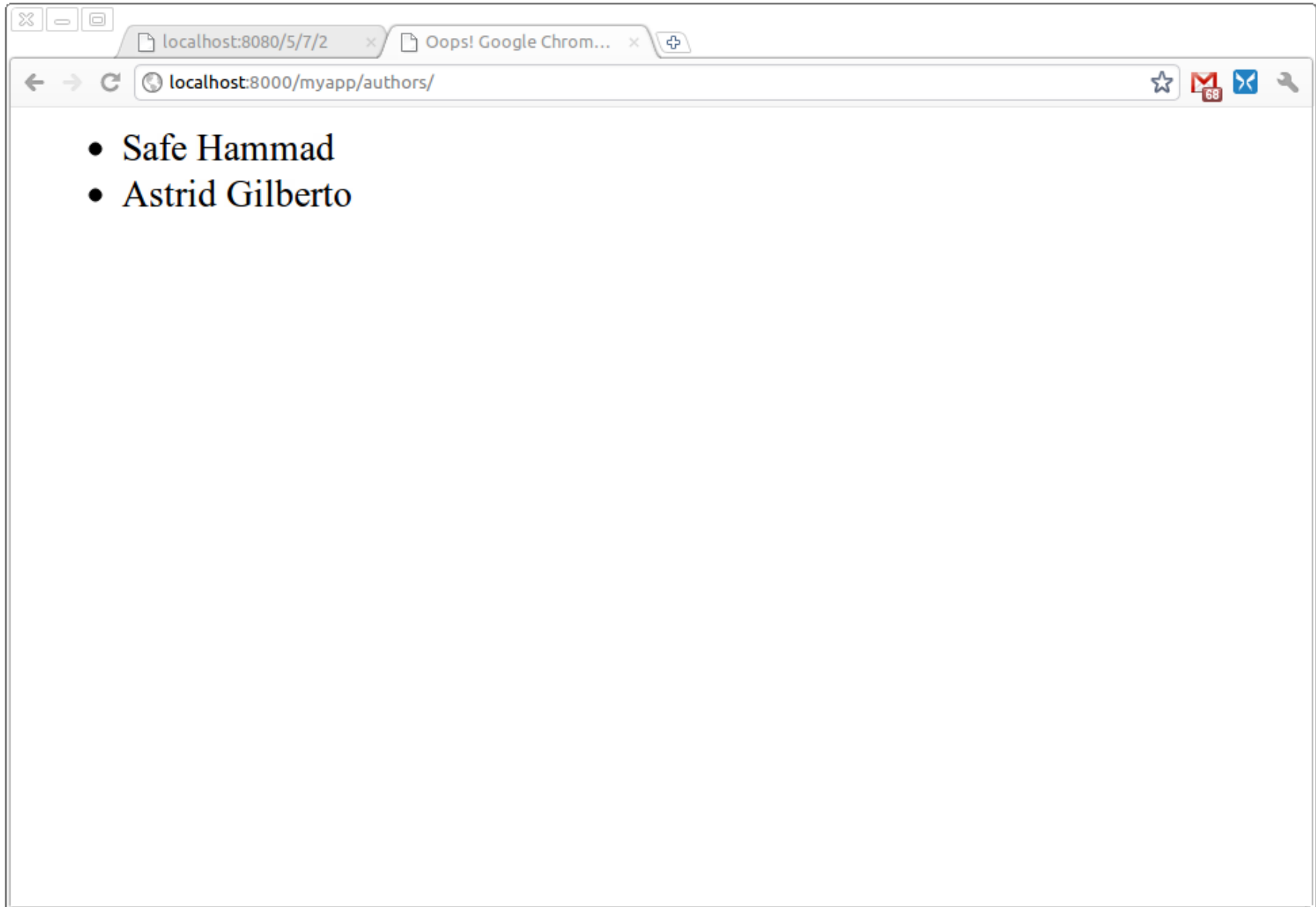
Context: `{'author': Author(first_name='Safe', last_name='Hammad')}`

Template: `{ a.first_name|upper }`

Output: `SAFE`



Our first templated application!



Template inheritance

- Pull out common HTML elements into their own file for other HTML pages to extend.
- Allows you to define a common page structure e.g. header, content, footer in a single HTML file for all other files to use.
- Sections of HTML to be overridden are declared using the `{% block %}` tag.



Template inheritance - example

```
# mysite/templates/myapp/base.html
```

```
<html>
  <head>
    <title>Authors</title>
    <body>
      <h1>The Authors Site</h1>

      {% block content %}
      {% endblock %}

      <p>Copyright blah!</p>
    </body>
  </html>
```

```
# mysite/templates/myapp/authors.html
```

```
{% extends 'myapp/base.html' %}
{% block content %}

  {% for author in authors %}
    <li>{{ author.first_name }}</li>
  {% endfor %}

{% endblock %}
```



Next steps

- **Template inheritance:**
 - Fully understand the importance and power of this feature.
- **Model API:**
 - Get to grips with the rich query language.
- **Forms.**
- **Generic views:**
 - Do away with boilerplate code for object list and object detail views.
- **Testing:**
 - The built-in test client mimics a web browser for calling URLs and checking response.
- **Django Admin:**
 - Automatic web interface.
- **Sessions.**
- **Authentication.**
- **Caching.**
- **Custom tags and filters.**
- **And more ... !**

Finally ...

- Read the tutorials and browse the docs. They're very good!
- Read other people's blogs and code. There's plenty out there!
- Post queries you have to the forums, IRC or Pysheff google group.
- *There's no substitute for getting stuck in!*

Thanks!

Safe Hammad
<http://safehammad.com>
@safehammad