Django - Heroku

Getting Started: Setup

- Install Python, Pip & Virtualenv
- Django

  $ pip install django

  $ pip install django-toolbelt
- Heroku - create a new account

  $ heroku login
- Download Heroku Toolbelt
- Postman - Chrome

Resolving Setup Errors/ Dependencies

- **brew install postgresql**

  (if not on the Mac, use Yum, or the postgresql binary)

- **pip install psycopg2**
  - OR **$ brew install psycopg2**
  - For the psycopg2 PATH error:
    - **sudo find / -name pg_config**
    - **export PATH=<Path>:$PATH**
- **pip install django**
- **pip install django-toolbelt**

django

The web framework for perfectionists with deadlines.

Django

- High-level Python Web framework
- Clean OOP design
- Free and open source
- Rapid development
  - authentication, content administration, site maps, RSS feeds —> out of the box
- Scalable
- Secure
  - Helps avoid SQL injection, cross-site scripting, cross-site request forgery and clickjacking.
  - Inbuilt user authentication system

Start a new project

- Create a directory for your project + virtualenv

```
$ mkdir myfolderproject

$ cd myfolderproject
```

Virtual Environment
(optional)

- **virtualenv** is a tool to create isolated Python environments.
- It creates an environment that has its own installation directories, that doesn't share libraries with other virtualenv environments
- We create the virtual environment with the flag **–no-site-packages** (for Virtualenv < 1.7), which indicates that only the packages installed in the virtual environment will be used.
- **$ virtualenv myenv <–no-site-packages>**
- **$ source myenv/bin/activate**

Django

- While you're in the virtual environment, install the Django Toolbelt

```
$ pip install django-toolbelt

$ django-admin.py startproject myproject
```

Procfile

- A Procfile is a text file that declares the commands, process types and entry points that will be run by your application on the Heroku platform.
- Create the file **Procfile** in the root directory of your app (at the same level where manage.py lives), and write:
- **web: gunicorn myproject.wsgi**
    - **web** process type — starting a web server
    - **gunicorn** the production web server recommended for Django

Requirements

- "Requirements files" are files containing a list of items to be installed using **pip install**
- Used to hold the result from **pip freeze** for the purpose of achieving *repeatable installations*. In this case, your requirement file contains a pinned version of everything that was installed in your virtualenv when pip freeze was run.

```
$ pip install gunicorn

$ pip freeze > requirements.txt

$ pip install dj-database-url

$ pip freeze > requirements.txt
```

Requirements

Here’s a sample requirements.txt file

```
dj-database-url==0.4.0
dj-static==0.0.6
Django==1.9.2
django-toolbelt==0.0.1
gunicorn==19.4.5
psycopg2==2.6.1
static3==0.6.1
wheel==0.24.0
```

Settings

Add your project name to the end of the INSTALLED_APPS list

INSTALLED_APPs: [….., ’myproject’]

To the end of the settings.py file add the following code…

```
# Honor the 'X-Forwarded-Proto' header for request.is_secure()
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

# Allow all host headers
ALLOWED_HOSTS = ['*']

# Static asset configuration
import os
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
STATIC_ROOT = 'staticfiles'
STATIC_URL = '/static/'

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static'),
)
```

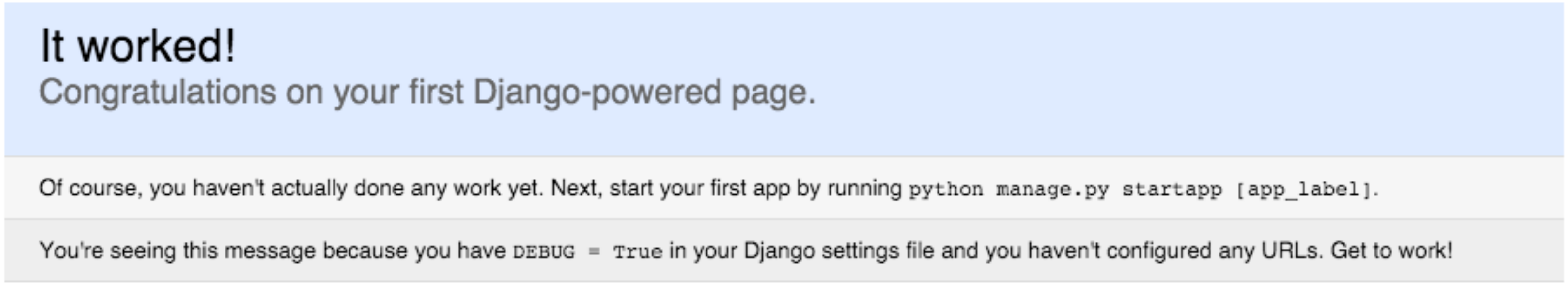
Test

```
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py runserver
```

Migrating Database:

- Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.
- These commands need to be run whenever a change is made.
- makemigrations: makes a list of migrations to be applied based on the db schema changes
- migrate: applies the migration

If you've done everything correctly, you should see this….



It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

models.py

Create a new models.py file at the same level as your settings.py

```
from django.db import models
import json, re

class DPUser(models.Model):

    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.CharField(max_length=50)

    def getResponseData(self):
        response_data = {}
        response_data["first_name"] = self.first_name
        response_data["last_name"] = self.last_name
        response_data["email"] = self.email
        return response_data
```

models.py

```
def __unicode__(self):
    return self.first_name

def __str__(self):
    return self.first_name

def __hash__(self):
    return self.id

def __cmp__(self, other):
    return self.id - other.id

class Meta:
    ordering = ('first_name',)
```

Optional methods:

- unicode & str: let you return a string representing the object instance.
- hash & cmp: are used by python's classes to let you compare two instances of a class
- Meta: being used to order the rows of the select query returned using a data attribute

UserManager

In your UserManager.py file (manager/UserManager.py)

```
import json
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

from ..models import DPUser



@csrf_exempt
def userRequest(request, user_id=None):
	return HttpResponse(json.dumps({‘success’:True}),
	content_type="application/json")
```

<u>Note:</u>

- Create a manager folder at the same level as the settings.py folder.
- Don’t forget to add in the __init__.py file into your manager folder for python to recognize the folder.
- Create a dummy userRequest method to be implemented later

urls.py

Add the following code into your urls.py file

<u>**Explanation:**</u>

- The 1st URL format calls a userRequest method in the UserManager file.
- The 2nd URL format accepts a GET parameter in the URL call and redirects control to the same userRequest method.

```
from manager import UserManager

#add the url pattern below
url(r'^api/user/$', UserManager.userRequest),
url(r'^api/user/(?P<user_id>\d*)/$', UserManager.userRequest)
```

UserManager

Explanation:

- The URLs we just created in the urls.py file redirect to this UserManager.userRequest method
- Replace our dummy code in the UserManager.py file with this to ensure we handle GET & POST requests correctly.
- Try calling the URL we defined in Chrome's Postman plugin with GET & POST calls

```
@csrf_exempt
def userRequest(request, user_id=None):
    if request.method == "POST":
        errorMessage = "TODO POST"
        response_data = {'success': True, "error":errorMessage}
    else:
        errorMessage = "TODO GET"
        response_data = {'success': True, "error":errorMessage}
    return HttpResponse(json.dumps(response_data), content_type="application/json")
```

UserManager

```
@csrf_exempt
def userRequest(request, user_id=None):
    if request.method == "POST":
        return createUser(request)
    else:
        return getUser(request, user_id)
```

Explanation:

- We handle the GET & POST separately. GET returns a user based on the user_id provided in the URL, POST creates a new user. We'll redirect the methods to other methods in the file accordingly.
- We assume the default user_id is None in GET methods to be safe.

UserManager

Explanation:

- We accept the POST parameters for the user (defaults == empty strings)
- Filter the current table by email to see if the user already exists
- If the user exists, we return an error

```
@csrf_exempt
def createUser(request):
	first_name = request.POST.get('first_name','')
	last_name = request.POST.get('last_name','')
	email = request.POST.get('email','')

	user = None
	existing_users = DPUser.objects.filter(email=email)

	if len(existing_users) > 0:
		# User Exists!
		user = existing_users[0]
		errorMessage = "Error! User with this email already exists."

		return HttpResponse(json.dumps({'success': False, "error":errorMessage}),
content_type="application/json")
```

UserManager

Explanation (continued):

- If the user doesn't exist, we create a new table entry in our DPUser() table.
- We add the values to the columns using the inbuilt Python setters
- save() adds the row to the table.
- The getResponseData() method on our DPUser class that we created earlier ensures that the data is properly formatted in a meaningful way and returns the same as an HTTPResponse object.

```
if user is None:
        user = DPUser()

    user.first_name = first_name
    user.last_name = last_name
    user.email = email

    user.save()

    response_data = user.getResponseData()

    return HttpResponse(json.dumps(response_data), content_type="application/json")
```

UserManager

```
@csrf_exempt
def getUser(request, user_id):
    response_data = {}

    if user_id:
        users = DPUser.objects.filter(id=user_id)


        if len(users)>0:
            user = users[0]
            response_data = user.getResponseData()

        else:
            errorMessage = "Error! This user doesn't exist."
            response_data = {'success': False, "error":errorMessage}

    return HttpResponse(json.dumps(response_data), content_type="application/json")
```

Explanation:

- Similarly, we check the user_id provided. If such a user exists, we return the user object using our predefined getResponseData() method on the model.

That's It!

- Run the migrations & runserver commands from the earlier slide to see your first API in action

Heroku

Heroku Account

- Login to your Heroku account
- Create new app (eg. masdjango)

On the Terminal:

- heroku login
- heroku git:clone -a masdjango
- git add —all
- git commit -m “Deploying to Heroku”
- git push heroku master

Links

Github for the django project