Dynamic Websites

Jamie Saxon

Introduction to Programming for Public Policy

November 7, 2016

Static and Dynamic Webpages

- Last week we built a **static** webpage, containing a single, fixed file and referencing other frozen resources (pictures).
  - Requesting that file's URL, we'll always see the same thing.
- Much of the web is **dynamic**: it evolves over time or as a function of its inputs, without someone literally rewriting the site.
  - N.B.: typically this means 'server side' dynamic: the server constructs the resource/site before sending it to you.
  - The individual resources (images, videos) in a site are usually static.
- Contrast to javascript (not covered), which can modify the page behavior or appearance as you click around (certain menus, web apps, formatting, etc.) and potentially request other resources.
  - Of course, those resources may be dynamic; e.g., gmail powered by AJAX (asynchronous javascript and XML, often actually JSON).

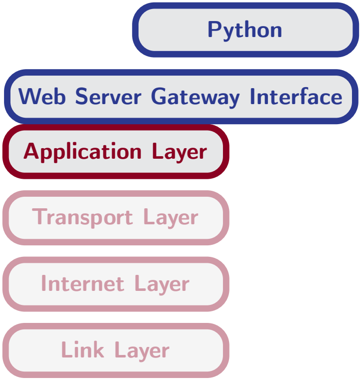
The Options for Dynamic Websites

- 'Traditional' dynamic web stack is LAMP:
  - Linux + Apache + MySQL (DB) + php.
- Many more options today: other servers (nginx), databases (Postgres, SQLite) and languages (python, Ruby on Rails).
- We'll continue using python, with **Django**.
  - Popular, powerful framework, used by Instagram, Pinterest, The Onion, Washington Post, etc.
  - Want more? Exceptional tutorials: **official** and **Django Girls**.

Web Server Gateway Interface

How do we 'hook up' python as a server?

- Server forwards requests to python through a 'gateway interface' module; python responds with a resource.
- Server should handle static resources itself (it's much better at it!); heavy server configuration is beyond the scope of this course.
- Luckily for us, Django provides a development server that is lightweight and easy to run.



What We'll Cover

Like python, a huge realm; choice of material and method very subjective.

- Initial set-up of django and a **site**.
- Create hello/goodbye world **views** and register its URLs.
- Return a **table** from pandas; the same, from csv.
- Build several very simple **dynamic** 'views.'
- Handling **GET** and **POST** data.
- An introduction to **forms**.
- Return a dynamic **plot** (v. generating 'all' the plots in advance).

These are basically the pieces you'll need to make your projects.

- On Wednesday, we'll make it pretty again, with HTML templates and hopefully a bit of bootstrap (pre-built css).

Getting Started with Django: A Site

If you have not already done so, install django:

```
■ conda install django
```

Navigate to a directory where you want to work, then:

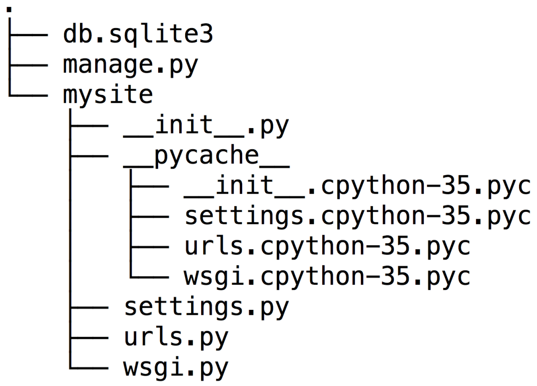
```
■ django-admin startproject mysite
■ cd mysite
```

Run the built-in development server, and check it out!!

```
■ python manage.py runserver
...
Django version 1.9.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
...
```

Site Resources

- This created a number of files for us.
- We've already run `manage.py`; we'll also use `mysite/settings.py` and `mysite/urls.py` a lot.



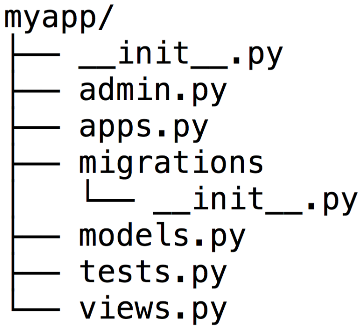
```
.
├── db.sqlite3
├── manage.py
└── mysite
    ├── __init__.py
    ├── __pycache__
    │   ├── __init__.cpython-35.pyc
    │   ├── settings.cpython-35.pyc
    │   ├── urls.cpython-35.pyc
    │   └── wsgi.cpython-35.pyc
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Getting Started with Django: An App

A site is a collection of (potentially) many **apps**. Let's make one!

- `python manage.py startapp myapp`

Just like `startproject`, this creates some files for us.



```
myapp/
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Getting Started with Django: A view

- **Views** are specific functions/pages within an app.
- Edit the file `myapp/views.py`, adding these lines:

```
from  django.http import  HttpResponse

def index(request):
  return HttpResponse("Hello Harris!!  A view!")
```

How Do We Find Our Site?

1. Tell the site about the app. In `mysite/urls.py`, add:
   - `from django.conf.urls import include`
   - `url(r'^myapp/', include('myapp.urls')),`
2. Tell the app about the view. In `myapp/urls.py`, put

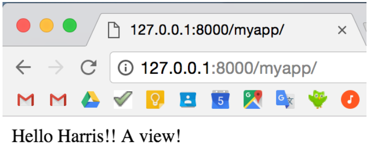
```
from django.conf.urls import url
from . import views

app_name = 'myapp'
urlpatterns = [
  url(r'^$', views.index, name='index'),
]
```

These are regular expressions, peeling off parts of the URL.

Now Run the Web Server Again:

```
python manage.py runserver
```

http://127.0.0.1:8000/myapp/



Hello Harris!! A view!

Let's Recap

1. Created a site with `django-admin startproject`.
2. Created an app with `python manage.py startapp myapp`.
3. Added a view to that app, which returned an `HttpResponse`.
4. Hooked up the `urls.py` for the site and app, to create the URL.
5. Have a site online!!

- Normally, we just create and register individual views.
- Let's make a 'farewell world' view (prepared for Wednesday!).

Using What We've Already Learned

Run functions on a website, leveraging the awesome power of python!

- Let's make a pandas table, and use `df.to_html()`.

```python
import pandas as pd, numpy as np

def table(request):

    df = pd.DataFrame(np.random.randn(10, 5),
                      columns = list("abcde"))

    table = df.to_html()

    return HttpResponse(table)
```

Loading Static Data

- With Django, it's easy to make dynamic sites. Unfortunately, using static data is a bit of a pain.*
- You will need to load CSV files for your homework and projets.

```
from os.path import join
from django.conf import settings
import pandas as pd

def csv(request):

  baby = join(settings.STATIC_ROOT, 'myapp/baby.csv')
  df = pd.read_csv(baby)

  return HttpResponse(df.to_html())
```

*Django has extraordinary database support with Object-Relational Mapping (a different interface to RDBMSs). If you loaded all your data into your database, this would be the way to go.

Collecting Your Static Files

- Let's make a small csv at `myapp/static/myapp/va_presidential.csv`.
- You also need to tell Django where your static files live.
- Set up the static files ROOT, in mysite/settings:
  - Add `'myapp.apps.MyappConfig'` to `INSTALLED_APPS`.
  - At the end, add `STATIC_ROOT = os.path.join(BASE_DIR, 'static')`
- Then run this (and accept):

```
■ python manage.py collectstatic
```

- We used the 'long' path, so that the 'collected' files would still have a recognizable pattern.

Dynamic Websites

Dynamic Websites: What's so great about Django?

- Strip variables out, in urls.py: this creates 'w':
  `url(r'^greet/(?P<w>[A-Za-z\- ]+)/$', views.greet)`
- Pass these to views:

```
def greet(request, w):

  return HttpResponse("Well hello, {}!".format(w))
```

- Let's create views to add and subtract numbers.

Templates

git clone git@github.com:harris-ippp/django-ex.git

What are Templates?

- Templates are reusable HTML code blocks, with insertion points for variables (or other large blocks), simple functions, and loops.
  - This reduces rewriting code (nav bar) and repetitive code (lists).
  - Philosophy: Don't Repeat Yourself (DRY).
- Template files live at `myapp/templates/ex.html`.
- 'Render' html code by inserting variables into the template by:
  `return render(req, "ex.html", {"key": value})`
- This is actually an `HttpResponse`.
- Best illustrated by example.

Prettifying and Codifying: Templates

- Let's start by using templates to insert values.
- Good to use variables, but also want to reuse e.g., nav bar.
- So we could write our greeting with a view, like this:

```python
from  django.shortcuts import  render

def greet_template(req, w):
  return render(req, "greet.html", {'who' : w})
```

- And a `myapp/templates/greet.html` as:

```
Well hello, {{ who }}!!
```

- Here, {{ `who` }} is variable, passed as an item in a dictionary.

Template Code: `for`, `if`

- Write for loops or if statements with:
  - `{% for x in xli %}<li>{{ x }}</li>{% endfor %}`
  - `{% if animal == "dog" %}ARFF{% endif %}`
- Let's create a view that renders a `pure_template.html`.
- This is very useful for elements in parts of forms, etc.

Template Code: Dictionary and List Items, Functions

- For functions, drop the parentheses.
  - For instance `{% for k, v in di.items %}` (instead of `di.items()`
- Access variables out of a dictionary or elements of a list using a dot, instead of brackets.
  - So: `di.cat` instead of `di["cat"]`.
- Still with `pure_template.html`, let's render the dictionary, `'di' : {"dog":"ARF", "cat":"MEOW", "tiger":"ROAR"}`

Template Inheritance

- I've copied my 'standard' header in `myapp/templates/base.html`.
  - The header will be reused every time.
- `{% block content %}` is a replaceable block of code.
- Swap it out by overwriting the bock, in `myapp/templates/view_table.html`.

```
{% extends "base.html" %}

{% block content %}
  {{ html_table | safe }}
  {% lorem 3 p %} <!-- some text -->
{% endblock %}
```

- New view: render table, using `html_table` and `view_table.html`.
- Very little html, to get a new page.

Additional Template Functions

- To load css files, scripts, or other static files, you may need:
    - `{% load staticfiles %}` and `{% static 'myapp/style.css' %}`.
    - The latter locates a static file, according to your static ROOT.
- There are many, many template filters, most notably, ‘`safe`,’ which just prints to html (instead of wrapping it as a string).
    - You’ve seen `{{ table_html | safe }}`.

Forms

Reading GET and POST

- You don't want users to have to edit the URL.
- The first step to fixing this is to be able to read the GET and POST requests that your website would send.
- This is pretty easy (and it is the same for POST):

```
def get_reader(request): # note: no other params.

  # if we knew the parameters...
  # state = request.GET.get('state', '')

  d = dict(request.GET._iterlists())
  return HttpResponse(str(d))
```

- So how do we send this data?

HTML Forms

- The standard HTML element for inputing data is a form: a collection of radio buttons, check boxes, text input fields of varying lengths, drop-down menus, etc. within `<form></form>` tags. Two important attributes:
  1. `action`: destination site.
  2. `method`: get or post.
- Take a look at the source for this form:

https://harris-ippp.github.io/lectures/form.html

HTML Forms and Django Models

- Django recognizes that there is likely a correspondence between data (objects/databases), forms (for creating objects), and views (for displaying them). They are all built from **Models**.
  - We won't cover this in depth – the databases are implemented with an Object Relational-Mapping (ORM, RDBMS interface wrapping SQL) that would be a good deal more work.
  - But you can learn more **here**.
- Nevertheless, Models are a useful tool.
  - Instead of manually writing each form, you can create a Django object that will do (a lot of) the work for you.

A Minimal Model

Imagine that we want a drop-down selection of US states.

- Many types (`CharField`, `IntegerField`, &c.) and display methods.
- Create a model inheriting from `django.forms.ModelForm` in `myapp/models.py`:

```
from django.db import models

# Create your models here.
STATES = ( ('AK', 'Alaska'), ('AL', 'Alabama'), # ...
           ('WV', 'West Virginia'), ('WY', 'Wyoming'))

class Input(models.Model): # our model inherits from Django.
  state = models.CharField(max_length=2, choices=STATES)
  name  = models.CharField(max_length=50)
```

A Minimal Form

Now writing the form is not so bad (but not intuitive, either).

- The `attrs` is for the html rendering of the element.

```
from django import forms
from .models import Input, STATES

class InputForm(forms.ModelForm):

  attrs = {'class' : 'form-control',
           'onchange' : 'this.form.submit()'}

  state = forms.ChoiceField(choices=STATES, required=True,
                            widget=forms.Select(attrs = attrs))
  class Meta:
      model = Input
      fields = ['state']
```

A Template: Display a Chosen State and an Option

- The elements of the form are ‘automatically’ by `.as_p` or `.as_table`, but the wrapping tags are not – they must be written out in the template.

```
{% if state %}
    I hear you, {{ state }}!!
{% endif %}

<form action="{{ form_action }}"
      method="{{ form_method }}">
  {{ form.as_p }}
</form>
```

A View

- We can pass all of the elements in the form to the template.
- Also use `reverse_lazy()` to look up the url of the site.

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic import FormView
from .forms import InputForm

def form(request):

    state = request.GET.get('state', 'PA') # PA = default
    params = {'form_action' : reverse_lazy('myapp:form'),
              'form_method' : 'get',
              'form' : InputForm({'state' : state}),
              'state' : STATES_DICT[state]}

    return render(request, 'form.html', params)
```

Dynamic Plots

Make and return plots on the fly. (Brilliant or misguided?)

- Each visitor wants a plot with specific parameters. Can't put them all in the same place (overwrite), but can't save them all, either (space).
- On the other hand, most of what you generate will be the ∼same. Making new plots for every single visitor is expensive.
- Depending on the parameters of your website, could generate 'all' possibilities and serve them statically. Faster but less flexible.

Other libraries (Bokeh) build plots with javascript from data... a good 'extension' in your projects.

Creating Plots on the Fly: A Recipe

Looks like a lot, but just two tricks:

1. Write to bytes instead of a file.
2. Return the bytes as `content_type = "image/png"`.

```
import matplotlib.pyplot as plt, numpy as np

def pic(request, c = "k"):

    t = np.linspace(0, 2 * np.pi, 30)
    u = np.sin(t)
    plt.figure()  # needed to avoid adding curves in plot

    plt.plot(t, u, color = c)

    # write bytes instead of file.
    from io import BytesIO
    figfile = BytesIO()

    # this is where the color is used.
    try: plt.savefig(figfile, format='png')
    except ValueError: raise Http404("No such color")

    figfile.seek(0) # rewind to beginning of file
    return HttpResponse(figfile.read(), content_type="image/png")
```