CIS192 Python Programming

Django and Web Development

Robert Rand

University of Pennsylvania

April 5, 2016

Outline

What Servers Do

- When a client makes a request a server creates the response
  - Client $\rightarrow$ Server $\rightarrow$ Client
- Server:
  - Interprets the request (Notices it'a a GET for `/somepage`)
  - Remembers who is making the request (which IP address)
  - Decides what to do based on the client and the request
  - Sends back a response to the client
- Servers also maintain data that can change (PUT, POST, DELETE)

What Can Go Wrong

- More users are making requests than the server can handle
  - Solution: Have more than just a single computer as the server
- Attack that specifically tries to overload server (DDoS)
  - Solution: Detect illegitimate requests and ignore those IPs
- Bug in the server:
  - Infinite Loop
  - Arbitrary code execution
- Requests and data from the internet can be harmful
  - Don't assume your server is getting good data

Outline

Flask Design Goals

- Based of Ruby's Sinatra
- Micro Framework:
  - The minimal code for requests of dynamic content
  - Doesn't include many extras
- Extensible:
  - Easy to extend with extra features (libraries)
  - Easy to replace the few built-in extras

Flask Features

- Built-in Features
  - URL routing with URL variables (`'/<variable>'`)
  - HTML templating (Jinja2)
  - Access to GET and POST parameters (`request.args`)
  - Save user specific data across requests (cookies)
  - Message Flashing
  - Logging
  - Thread safe global variables (`flask.g`)
- For more Flask, check out the previous year's Web API lecture.

Pyramid Design Goals

- Lightweight:
  - Allows for very minimal initialization.
  - A “Hello World” app takes only a few lines of code
- Highly extensible and configurable
  - Large add-on library
  - Deep configuration system and extension facilities
- Can get complex quickly.

Django Design Goals

- Easily create complex database-driven websites
  - Pre-made solutions to common web tasks
  - Many features turned on by default
  - A minimal Django app can do a lot
- Don't Repeat Yourself (DRY)
  - Reusable and Plug-able components
  - Plenty of abstraction allows for code reuse
- Extensible
  - Any reusable app can be plugged into a Django project
  - A reusable app must adhere to a list of requirements
  - reusable apps provide functionality like: search, API handling
  - A website is backed by a Django project
  - Projects can use multiple apps

Django Built-ins

Everything Flask has plus:

- Model View Controller (MVC) framework
  - Database backed (Model)
  - HTML Templating (View)
  - URL routing (Controller)
- Form validation
- Caching
- Object Relational Mapping (ORM)
- Database Backends (PostgreSQL, MySQL, SQLite, Oracle)
- Internationalization
- User Authentication
- Administrator interface to the database
- Site-map generation
- Security: XSS, SQL injection, SSL, ...
- ...

Example Django App (Project Setup)

- Django Website: This follows the Django tutorial.
- First Django will generate an initial setup for you
  - Makes a directory, 5 files with 138 lines
- `settings.py` contains default settings
  - Change the `TIME_ZONE` to `'America/New_York'`
- Initialize the database for built-in reusable apps
  - `python manage.py migrate`
  - Check out the database with `sqlite3 db.sqlite3`
  - Type `.schema` at the prompt
- Now the app is ready to run
  - `python manage.py runserver`
  - Listen on all ports:
    `python manage.py runserver 0.0.0.0:8000`

Example Django App (App Setup)

- Create an app inside the project
  - `python manage.py startapp some_app_name`
  - generates another directory and 6 files with 12 lines
- Define models (Things to be stored in the database)
  - edit `models.py` with Python Classes for each thing
  - Each class has class variables which map to Database types
  - Can add any methods you want: `__str__` is useful
- Add your app to the `settings.py`
- Update the database with your app
  - `python manage.py makemigrations some_app_name`
  - `python manage.py migrate`
  - Migrations can alter the DB schema while the app is live

Example Django App (Using the DB)

- `from app_name.models import YourClass, ...`
- Create database entries by constructing classes
  - set attributes with kwargs
  - Make sure to save your object to the DB
- Look at all objects with `YourClass.objects.all()`
- The built in admin page
  - Register your models to be admin editable
    - ★ In `your_app/admin.py`

```
from django.contrib import admin
from .models import M1, M2, ...
admin.site.register(M1)
```

  - Create an admin user: `python manage.py createsuperuser`
  - Start the server: `python manage.py runserver`
  - Go to the admin endpoint: `http://127.0.0.1:8000/admin`

Example Django App (Templating)

- Create a template dir by modifying `settings.py`
  - Add `'DIRS': [os.path.join(BASE_DIR, 'templates')]` to the `TEMPLATES` dictionary
- Make a templates dir in the same dir as `manage.py`
- create template files to match the urls
  `'/admin'` → `/templates/admin/something.html`
- copy templates from django source into that dir and modify
- The Django templater uses `{{ var }}` and `{% if %}` like Jinja

Example Django App (Views)

- Views are the functions to executed for a given url
- Similar to functions decorated with `@app.route()` in Flask
- Put the functions in `views.py`
  - View functions take in a request and output a response
- Create a `urls.py` file in the same directory
  - create a list of called `urlpatterns`
  - each element is a `url(regex, function, name=string)`
  - This maps urls that match the regex to the view function
- Tell the root `urls.py` that it should forward some urls to your app
  - `url(url(r'^your_app/', include('your_app.urls')))`
- Any captured groups in the regex get passed as args to the views
- Render a template from a view
  - `context = RequestContext, request, 'key': val`
  - `render(request, 'template_path', context)`