CIS192 Python Programming
More Webservers

Eric Kutschera
University of Pennsylvania

April 10, 2015
Final Project

You should all have received feedback from me on your proposal
  You don’t need to respond (but feel free)
  Get started writing code

All code is due Wed April 29 23:59
  Make another Pull request like you did for the proposal

Demos
  Planning on doing a few hour session each day for May 4-6
  Could also have demos during reading days
  Thoughts? I’m trying to submit final grades by May 7

CIS Demo Days
  Either Thursday or Friday April 30 / May 1
  Lunch will be served
  Show off your projects (Google/Comcast/Other companies)
  10 pts Extra Credit for going (0.1 * 0.25 = 2.5 points on final grade)
  Sign up sheet to follow once date is finalized
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
     - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
Behind the scenes a website can be separated into two tasks

Server
- Receiving HTTP requests over the network
- Sending responses back over the network

Web Application
- Deciding what it means to make a certain request to an endpoint

Web Server Gateway Interface (WSGI)
- Specifies an interface between web servers and web apps
- Goal is to allow any web framework to work with any web server
- This allows a developer to consider server and application trade-offs separately
Middleware

- WSGI specifies:
  - What a server must do
  - What a web application must do
- A full website is created by Initializing a server with an app
  - \texttt{website = wsgi\_server(wsgi\_app)}
  - Disclaimer: Can get much more complicated
- It is possible to implement both the server and web app interfaces
- Such a program would be an example of \texttt{middleware}
  - \texttt{website = wsgi\_server(middleware(wsgi\_app))}
- Middleware can provide features like:
  - Authentication (Log-in)
  - Altering HTTP headers
- Can nest multiple layers of middleware to form a middleware-stack
1 Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2 Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3 Web Frameworks
   - Flask
   - Django
from flask import Flask

app = Flask(__name__)

@app.route('/', methods=['GET'])
def home_page():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from my_app import app  # Flask app

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(5000)
IOLoop.instance().start()
from flask import Flask
from werkzeug.contrib.fixers import HeaderRewriterFix

app = Flask(__name__)
app.wsgi_app = HeaderRewriterFix(app.wsgi_app,
    add_headers=[('X-Powered-By', 'WSGI')])

@app.route('/', methods=['GET'])
def home_page():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
http.server replaces SimpleHTTPServer in Python3
http.server is not a WSGI compliant server
Just serves static files
No application logic
For when Flask is too complicated
python -m http.server
- serves up your files at or below the folder you run in
- Instead of emailing a file to the person next to you ...
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
Waitress

- Goal: Production WSGI server that always works
- Written in Python using only the standard libraries
- Supports Linux and Windows with Python 2.6+ and Python 3.2+
- Decent performance
- Not many features

```python
from waitress import serve
from my_app import app  # Flask app
serve(app)
```
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
Green Unicorn: WSGI server for Linux
Based on Ruby’s Unicorn
Goal: Use many workers to be fast
Uses a master process that creates and manages workers
Worker Types

- Gunicorn spawns a given number of worker processes on start
- Gunicorn allows you to choose among different worker types
- Sync Workers
  - Handles exactly one request at a time
- Async Workers
  - Cooperative multi-threaded workers
  - Can handle many requests simultaneously
- Tornado Workers
  - Workers from the Tornado web-framework
  - Designed to handle extremely high request counts
  - Potential issues using them outside the Tornado framework
1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
uWSGI

- Very Complicated and Very Fast
- Designed to handle languages other than Python
- Goal: Perform all types of web-server functionality very fast
- Docs are confusing (Since it does everything)
- Apparently it’s the fastest WSGI server
1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
Proxy Servers

- The WSGI servers
  - Route network operations to and from the app
  - Assume one machine is handling the entire website
- Putting the WSGI server behind a proxy allows the proxy to
  - Balance request workloads over multiple server machines
  - Handle web-based attacks (Viruses, DoS)
  - Conceal the IP of the actual server
  - Optimize performance by caching and buffering responses
  - Handle SSL encryption
Apache and Nginx (engine-x) are the most popular proxies. Plenty of blog posts about combining WSGI servers and proxies (Gunicorn behind Apache vs. uWSGI and Nginx).

Apache
- Older and more widely used
- Handles more requests with more processes/threads

Nginx
- More recent and second most used
- Designed specifically for handling lots of connections
- Handle requests with asynchronous events
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
Flask Design Goals

- **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
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`)
Outline

1. Web Server Gateway Interface
   - WSGI Goals and Spec
   - Flask Example

2. Web Servers
   - SimpleHTTPServer -> http.server
   - Waitress
   - Gunicorn
   - uWSGI
   - Proxy Servers

3. Web Frameworks
   - Flask
   - Django
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, ...
- ...

Eric Kutschera (University of Pennsylvania)
Example Django App (Project Setup)

- **Django Website**: I’m following the tutorial there
- 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`
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`
      ```python
      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`
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
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)