CIS192 Python Programming
Web Frameworks and Web APIs

Harry Smith

University of Pennsylvania

November 8, 2017
Outline

1. Web Servers
   - What’s a Web Server?
   - Flask

2. Web APIs
   - REST
   - Encoding and Encryption
Web Servers

- We’ve been talking about *making* HTTP requests
  - typing www.google.com in your browser
- What about *serving* them?
  - where is the code that powers www.google.com?
What Servers Do

- When a client makes a request a server creates the response
  - Client → Server → Client
- Server:
  - Interprets the request ( Notices it’s 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 Are Web Frameworks?

- Library that makes it easy to setup a web server
- i.e. Build your own web apps!
- We’ll be looking at Flask today
- Other examples you might’ve heard of:
  - Python: Django, Tornado, Bottle, ...
  - Ruby: Ruby on Rails, Sinatra, ...
  - JavaScript & Node.js: Express.js, Meteor, ...
Outline

1. Web Servers
   - What’s a Web Server?
   - Flask

2. Web APIs
   - REST
   - Encoding and Encryption
Why Flask

- **Micro Framework:**
  - The minimal code needed to accept and respond to requests
  - Doesn’t include many extras
  - Everything you need and nothing you don’t

- **Extensible:**
  - Easy to extend with extra features (libraries)
  - Easy to replace the few built-in extras

- **Actively Developed**
  - Support for latest Python version and popular tools
  - Bug fixes

- **Active Community**
  - Can find answers on Stack Overflow
Configuring the Server

- **Install:** pip3 install flask
- If pip3 isn’t working, click here for alternatives.
- Create server:
  ```python
  from flask import Flask
  app = Flask(__name__)
  ```
- Set options:
  ```python
  app.debug = True
  ```
Handling our first Request

- First step: Create an Endpoint
  - This defines where the client is supposed to "go"

```python
@app.route('/', methods=['GET'])
def my_page():
    return 'Hooray Flask is working!'
```

Run the server: `app.run()`

If you run the file, it will say what url to use

```
$ python lec11.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```
What are the endpoints supposed to look like? Jinja for Fancy HTML

- Flask has Jinja2 as a built-in Template Engine
- Allows you to write (something like) python inside HTML

```python
return render_template('my_template.html', arg='my_value')
```

- returns an HTML page by running a template
- Templates can take arguments

- Need to put templates in a templates directory
Jinja Features 1: Take What’s Nice about Python and Throw it in the Garbage

- Evaluate variable or expression with `{expr}`
- A block names part of a template
  ```
  {% block name %}
  some HTML
  {% endblock %}
  ```
- for loop
  ```
  {% for item in things %}
  <li>{{ item }}</li>
  {% endfor %}
  ```
if statements

{% if date %}
{{ date }}
{% elif other %}
{% else %}
{% endif %}

Inheritance uses parent HTML but can redefined blocks

{% extends "my_template.html" %}
{% block some_block_name_in_parent %}
replacement content
{% endblock %}
from flask import request, redirect, url_for
@app.route('/submit', methods=['GET', 'POST'])
check which method with request.method == 'POST'
access POST data with request.form['key']
Access GET params with request.args.get('key')
return redirect(url_for('my_page'))
  Sends the user to the Flask endpoint my_page
Arguments in URL

A route can contain variables

```python
@app.route('/say/<message>', methods=['GET'])
def url_param_example():
    return render_template('message.html', message=message)
```
Logging

- Useful for reporting errors, security incidents, timestamps, etc.

```python
l_name = my_flask.log
log_handler = logging.FileHandler(l_name)
log_handler.setLevel(logging.DEBUG)
app.logger.addHandler(log_handler)

app.logger.debug('the message to log')
```

- Uses the `logging` standard library
Other Web Frameworks for Python

- **Django**
  - A lot of work done for you: batteries included
    - Admin interface
    - User model
    - Easy database integration
  - Great for prototyping, rapid app development

- **Pyramid**
  - Configurable, flexible
  - Too many options? Intimidating to start new projects
Outline

1. Web Servers
   - What’s a Web Server?
   - Flask

2. Web APIs
   - REST
   - Encoding and Encryption
Principles

- A REST API → Representational State Transfer
- Client-Server: Separation of tasks (data storage vs. user state)
- Stateless: Each client request has all necessary info
- Layered system: Client can’t tell if connected directly to server
- Uniform interface:
  - All resources named the same way (URLs)
  - All messages describe how to process themselves
  - State transitions are determined dynamically from the resources
Example REST API

- Dropbox: Dropbox Core API
- Primarily uses GETs and POSTs
- The endpoints allow for variables in the url
  - https://api.dropbox.com/1/metadata/auto/<path>
- Uses OAuth 2.0 for login
- Responses are encoded with JSON
Outline

1 Web Servers
   • What’s a Web Server?
   • Flask

2 Web APIs
   • REST
   • Encoding and Encryption
Review of JSON

- Many Web APIs transmit data in JSON
- JSON → JavaScript Object Notation
- Data Types:
  - Numbers: 25, 167.6
  - Strings: "firstName"
  - Boolean: true, false
  - List: [25, "firstName", true]
  - Dictionary with String keys: {"fst": 1, "snd": 2}
  - Empty Value: null
- Always wrap your JSON in a top-level dictionary
  - {"data": original_JSON}
  - JavaScript Bug allows top-level arrays to be hacked
JSON in Python

- The JSON standard library: `import json`
- `json.dumps(obj)` returns a JSON string of `obj`
- `json.dump(obj, f_handle)` writes the JSON to the file
- `json.loads(s)` returns a Python object from a JSON string
- `json.load(f_handle)` returns Python object from a file

Flask has JSON: `from flask import jsonify, json`
  - use Flask’s `json.dumps()`/`loads()`
  - `return jsonify(d)` sends a JSON response from a `dict`
  - Takes care of details like headers and encoding

requests has JSON
  - `r = request.get(...)`
  - `r.json()` parses out a Python object
If your Web App contains sensitive data → Protect It
Making users login is a good first step
But ... other people can listen in on HTTP requests
HTTPS uses ssl (Secure Sockets Layer)
  ▶ Fancy encryption for sending messages
  ▶ Standard way to protect data on the Web
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