Outline

1. Web Servers
   - Purpose of Web Servers
     - Flask

2. Web APIs
   - REST
   - Encoding and Encryption
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 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

1. Web Servers
   - Purpose of Web Servers
   - Flask

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

- Based off Ruby’s Sinatra
- 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:** `pip install flask`
- **Create server:**
  ```python
  from flask import Flask
  app = Flask(__name__)
  ```
- **Set options:**
  ```python
  app.debug = True
  ```
Handling a Request

- **Create an Endpoint**

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

- **Run the server:**

  ```python
  app.run()
  ```

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

  ```bash
  $ python lec11.py
  * Running on http://127.0.0.1:5000/
  * Restarting with reloder
  ```
Flask has Jinja2 as a built-in Template Engine
Allows you to write (something like) python inside HTML
\[
\text{return render_template('a_temp.html', arg1=val1)}
\]
  - returns an HTML page by running a template
  - Templates can take arguments
Need to put templates in a templates directory
Jinja Features 1

- 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 %}
  ```
Jinja Features 2

- if statements

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

- Inheritance uses parent HTML but can redefined blocks

```jinja
{% extends "a_temp.html" %}
{% block from_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
A route can contain variables

```python
@app.route('/log/<msg>/<mode>')
def searching(msg='', mode='info'):
```
Can log out to a file

```
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
  - Overkill for small applications
  - Possibly confusing for new web developers

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

1. Web Servers
   - Purpose of Web Servers
   - 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
   - Purpose of Web Servers
   - Flask

2. Web APIs
   - REST
   - Encoding and Encryption
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
Python Handles JSON For You

- **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
Not in Standard Library: `pip install crypto`

```python
>>> from Crypto.Hash import MD5
>>> m = MD5.new()
>>> m.update('abc')
>>> m.hexdigest()
'900150983cd24fb0d6963f7d28e17f72'
```

Crypto also has:
- More hashes: MD5 SHA HMAC
- Symmetric Key (AES ...) Public/Private Key (RSA)
- Signature creation and verification

`base64` is in Standard Library

- `base64.encode(byte_string)`
- `base64.decode(byte_string)`