Web Development with Flask

Instructor: Jorge Mendez
Flask

• Lightweight WSGI web application framework
  • WSGI: Web Server Gateway Interface — specification for server-app communications

• Built on top of Werkzeug (WSGI toolkit) and Jinja2 (web template engine)

• Installation
  • pip install Flask
  • conda install –c anaconda flask
More on Flask

• It is a micro-framework: does not restrict the tools or libraries

• Must use third-party libraries for database abstraction, form validation, upload handling, authentication
  • Integration with such tools is simple

• Used for back-end (server-side) development

• Used in the stacks of Pinterest and LinkedIn
Hello world

• `app = Flask(__name__)` — an instance of the Flask class
  • `__name__` tells Flask where locate templates and static files (directory tree)
• `@app.route('/')` — decorator to tell the app which URL calls the function
• `return ‘Hello, World!’` — return message to display in browser
• To run this simple server
  • In Linux/Mac: `export FLASK_APP=hello.py; flask run`  
  • set instead of export in Windows  
  • For non-terminal users: can replace ; with a linebreak
Make your server visible

• By default, sites can only be viewed locally (i.e., from our own machine)

• Why? In debug mode, users can run arbitrary Python code on our machine! (dangerous)

• Add `--host=0.0.0.0` to the command to make it visible
Debug mode

• Typically, we’d need to re-run the server after every change to our code
• If you use debug mode, the server is reloaded upon each change to the code
• Debugger shows stack trace on the browser
• Add export/set FLASK_ENV=development before flask run
  • If you want to go back, export/set FLASK_ENV=production
App directory structure

• Flask applications can be either a single Python module or a Python package
• If it is a module, Flask assumes there is a `static/` and a `templates/` directory next to the module for finding static files and templates
• If it is a package, Flask assumes the `static/` and `templates/` directories are inside the package
• You can modify this behavior in Flask configurations
Routing

• `@app.route(str)` specifies which URL calls each function

• Use meaningful URLs so it’s easy for users to locate them (e.g., ‘google.com/gmail’ or ‘google.com/flights’)

• ‘/’ is the “main page”

• `@app.route(/path/<varname>)` — pass varname as keyword argument to function
  • Optionally include type — `str:varname` or `int:varname`
  • Use `path:` if you need to pass a string that contains /
Routing

- Trailing slashes: like directory names (sites with sub-sites)
  - If you access URL without slash it just re-directs to the URL with the slash
- No trailing slashes: like filenames
  - If you access URL with trailing slash it gives 404 error
  - Avoids search engines indexing same page twice
Routing

- If you need to use the URL elsewhere, use `url_for(function_name, kwargs)`
- If your application requires static files (e.g., `index.html`), include them in `/static` directory
  - Flask redirects `/static` URLs there
- Use `url_for('static', filename=filename)`
- Usually, you’ll instead serve static files from the server directly with calls to `app.send_static_file(filename)` (for fixed paths) or `app.send_from_directory(dir, path)` (for user-provided paths, for security)
HTTP methods

- HTTP — Hypertext Transfer Protocols
- Used for communication between server and client
  - Client issues requests, and server responds
- Flask creates responses from function returns that conform to HTTP
- HTTP methods refer to the type of requests received from clients
  - GET — retrieve information from the server
  - POST — pass information to the server
  - OPTIONS — get available communications options (implemented by Flask)
  - HEAD — like GET, but only return the header (implemented by Flask from GET)
HTTP methods on Flask

• By default, implement only GET
• Otherwise, pass list of methods to the route decorator

```python
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```
How do HTTP methods work? Example

• When user goes to my_site/login, a GET request is issued
  • At this point, the server will display an HTML form

• If the user enters a username and password, and hits the “Log In” button, a POST request is issued, with username and password as arguments
  • For this, the HTML form will need to include a button with a POST action

• The server can then handle the POST request and log in the user if the credentials are valid, or display the login page with an error
Rendering templates

• Writing HTML code in Python is painful
• To make applications secure, you need to make sure variables you are passing in are not HTML code
• Flask uses Jinja2 for this, so you don’t have to manually handle HTML code
• `render_template(filename, kwargs)`
• Looks for `filename` in `templates/` directory
Accessing request data

- Request data contains information the client passes to the server
- The request object is a global Flask object (from Flask import request)
  - request.method — ‘PUT’, ‘GET’, ‘OPTIONS’...
  - request.form — dictionary with keys in a form posted by client
    - E.g., for a login page, the form would include username and password
  - request.args.get(key, default) — get keys from URLs that contain ?key=value
- Fare more methods and attributes...
Example of the request object

```python
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'], request.form['password']):
            return log_the_user_in(request.form['username'])
    else:
        error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```
File uploads

• Uploaded files are in the request.files dictionary
• Each file behaves like a Python file
• Additional .save() method to store the file
• Additional .filename attribute that gives the filename on the client side
  • Do not trust .filename! use secure_filename(f.filename) if you want to use the same name
Cookies

- HTTP communication on its own is stateless
- Cookies are pieces of information used to keep track of the state of a user’s connection
- The idea is the server sends them to the client, and the client sends them back, so the server can check what the previous state was
- `request.cookies.get(key, default)` returns the value of a cookie
- To send a cookie, you need access to the HTTP response (we will see this later) and do `resp.set_cookie(key, value)`
Redirects and errors

• A function can re-direct to a different URL with `redirect(url_for(function))`

• To throw an HTTP error, use `abort(401)` for access denied, `abort(404)` for page not found...
  • You can create your own error handlers with the decorator `@app.errorhandler(error)`
Responses

• HTTP responses must conform to the protocol
• They contain a header and data
• The header contains information about the response
  • Status — whether everything was okay
    • Default: 200 OK
  • MIME type — how to interpret the data
    • Default: text/html
    • Others: text/plain, application/pdf, application/octet-stream (for download), application/json
Responses

- Flask supports the following return types from functions
  1. Response objects — directly returned
  2. Strings — used as the response’s data
  3. Dicts — converted to a response with `jsonify()`
  4. Tuple including response and status and/or headers
  5. WSGI application

- If you need access to the response in your function (e.g., to add a cookie), call `resp = make_response()` on any of the above
Sessions

• Beside request, there is a global session object
• Built on top of cookies, but is encrypted
• Always use sessions instead of cookies for session handling
• Call `app.secret_key = os.urandom(16)` to set the encryption key
• Then, set and get values from session as a dictionary
  • `session[‘username’] = request.form[‘username’]`
Takeaways

• Flask makes it easy to handle HTTP requests and responses
• It allows you to configure anything and everything, but provides sensible defaults for most applications
• It adds layers of security (encryption) whenever needed