THE UNIVERSITY OF WESTERN AUSTRALIA

Topic 9: Flask Applications

CITS3403 Agile Web Development

Adapted from the Flask Mega-Tutorial, by Miguel Grinberg:
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial

Semester 1, 2019

Full Stack Development?

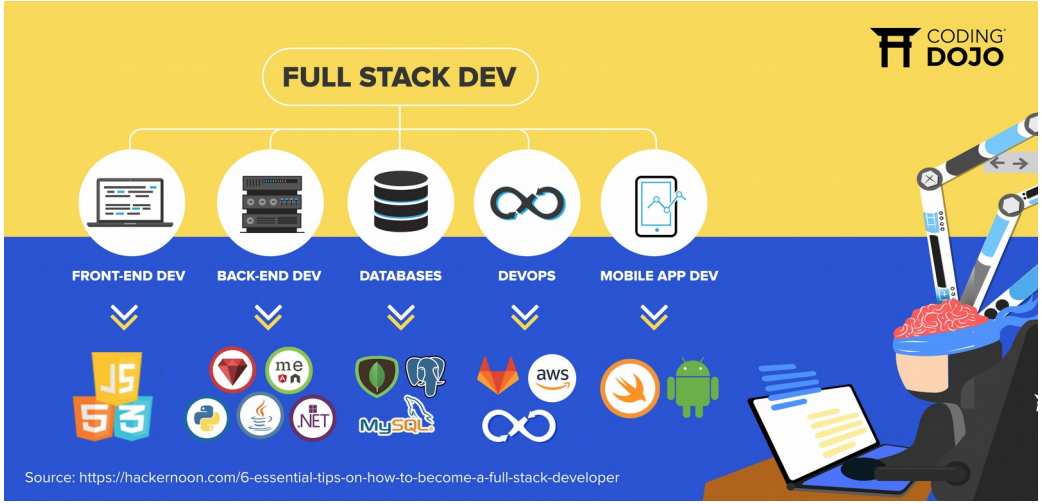
THE UNIVERSITY OF WESTERN AUSTRALIA

- Full stack development refers to developing all parts of a web application: database, web server, application logic and front end.
- There are various "Full stacks" people use to develop:
  - LAMP (Linux, Apache, MySQL and PHP)
  - Ruby on Rails
  - Django (Python)

- We're going to use a number of tools in this unit:
  - **Flask:** is a micro framework, that allows us to write our backend in Python. It contains its own lightweight webserver for development
  - **SQLite:** is a lightweight database management system
  - **AJAX** and **JQuery**: We have already seen these. We will use these for making responsive web pages.

Full-stack development

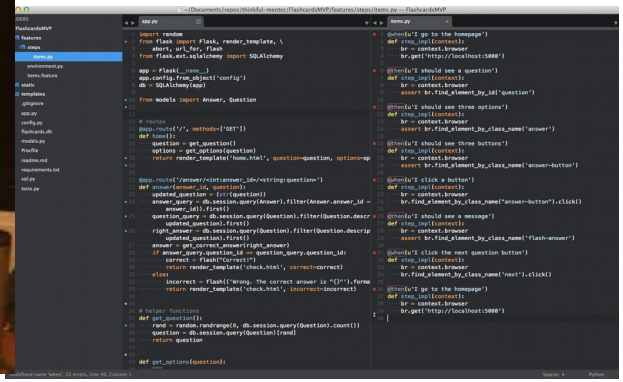
- Fullstack development refers to developing every part of the web application.
- It involves knowing all the technologies used from mobile and front end, (HTML, CSS, javascript frameworks) though to the backend logic, security and database models used at the backend.
- Most developers are specialised in one part of the stack.



Development environment

THE UNIVERSITY OF WESTERN AUSTRALIA

- A lot of web development is done from the command line, since traditionally servers didn't need a graphical front end.
- We can use Git to develop on laptops and push code to the server, but we still rely heavily on command line tools.
- By now, every one should have a good text editor that does syntax highlighting etc, some tool to allow them to compile or run code with the command line, and a browser with developer tools to view source, and debug javascript.
- You should also have a Git client to regularly commit your code, and push to others.



Getting started with Flask

- Flask is a micro-framework to run on the server, but it will run on any machine, and has few dependencies.
- You will require python3 installed in your operating environment, with pip.
- Use pip to install venv (virtual environment) first and initialise the environment.
- Now install Flask. Any required modules will be preserved by the virtual-environment.
- You can now run flask by typing `flask run`, but the app doesn't know what to run.
- Write the following into `app.py`, run the program again.
- Use a browser to see your app in action! (`http://localhost:5000`)

```
drtnf@drtnf-ThinkPad:$ python3  -m venv tmp-env
drtnf@drtnf-ThinkPad:$ source tmp-env/bin/activate
(tmp-env) drtnf@drtnf-ThinkPad:$ pip install flask
Collecting flask
```

```
1 from flask import Flask
2 app = Flask(__name__)
3 @app.route("/")
4 def hello():
5     return "Hello world!"
6 if __name__ == "__main__":
7     app.run()
```

app.py

```
(tmp-env) drtnf@drtnf-ThinkPad:$ flask run
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Application structure

- Our app.py file doesn't look like much. It has a method to return 'Hello world!' that is *decorated* with `@app.route('/')`.
- `app` is an instance of the class `Flask`. When it runs it listens for requests, and if the route matches a decorator, it executes the corresponding function. A request object is passed to the method.
- The return of the function becomes the response.
- But this structure doesn't scale well.

```
1 from flask import Flask
2 app = Flask(__name__)
3 @app.route("/")
4 def hello():
5     return "Hello world!"
6 if __name__ == "__main__":
7     app.run()
~
~
```

app.py

```
from flask import request

@app.route(...)
def login():
    username = request.args.get('username')
    password = request.args.get('password')
```

A better application structure

THE UNIVERSITY OF WESTERN AUSTRALIA

- A better structure is to create a package `app` that will contain all the code we need for the web app.
- It has an `__init__.py` file to create an instance of the `Flask` class.
- We can create a file `routes.py`, to contain the request handlers.
- Finally, we need a file at the top level to import the app. We set the system variable `FLASK_APP` to the name of this file, so flask knows what to run.
- Now the `app` package can contain files for handling routes, modules, templates, tests and anything else our application requires.
- Typically, *models* will contain the database schema, *templates* will contain the html, and *static* will contain the css and javascript to be served.

app/__init__.py: Flask application instance

```
from flask import Flask

app = Flask(__name__)

from app import routes
```

app/routes.py: Home page route

```
from app import app


@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

microblog.py: Main application module

```
from app import app
```

Server-side Rendering

- Our app will now listen for requests, and we can use python functions to build html pages to return as a response.
- However, this mixes the logic and the presentation.
- A typical pattern to use is to have a `template` or `views` directory to have some html that references objects and code, and a rendering function that will take a template and some data and builds the html dynamically.
- Flask uses jinja for this task, but there are many alternatives (pug, handlebars, typescript)

app/routes.py: Return complete HTML page from view function

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return '''
<html>
    <head>
        <title>Home Page - Microblog</title>
    </head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!</h1>
    </body>
</html>'''
```

```
22  <h3>Registered project list</h3>
23  <table class='table table-striped table-bordered'>
24    <tr>
25      <th>Project Team</th>
26      <th>Project Description</th>
27      <th>Demo location</th>
28      <th>Demo time</th>
29      {% if not current_user.is_anonymous %}
30        <th>Action</th>
31      {% endif %}
32    </tr>
33    {% for p in projects%}
34      <tr>
35        <td>{{p['team']}}</td>
36        <td>{{p['description']}}</td>
37        <td>{{p['lab']}}</td>
38        <td>{{p['time']}}</td>
39        {% if not current_user.is_anonymous %}
40          <td>
41          {% if p['project_id']== current_user.project_id %}
42          <a href='{{url_for("delete_project") }}'>delete</a>
43            <a href='{{ url_for("edit_project") }}'>edit</a>
44          {% endif %}
45          </td>
46        {% endif %}
47    </tr>
48    {% endfor %}
49  </table>
```

Using Jinja

- We separate presentation and logic by having a template directory to contain annotated html, and specify a rending function in the routes.py file
- When a request is received flask will look for the matching template (in the directory templates) and convert the template to pure html using named variables in the function.
- Two {{curly braces}} are used to distinguish html from python variables, and jinja does the substitution

app/templates/index.html: Main page template

```
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

app/routes.py: Use render_template() function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Home', user=user)
```

Jinja Loops and Conditionals

- Depending on the parameters passed, we may want to display the data differently.
- Jinja provides loops and conditionals to allow the display to adapt to data.
- For example, it is common to pass in an array of objects, and then present them in a table.
- Or we may want the display to vary depending on who is logged in.

```
22  <h3>Registered project list</h3>
23  <table class='table table-striped table-bordered'>
24    <tr>
25      <th>Project Team</th>
26      <th>Project Description</th>
27      <th>Demo location</th>
28      <th>Demo time</th>
29      {% if not current_user.is_anonymous %}
30        <th>Action</th>
31      {% endif %}
32    </tr>
33    {% for p in projects%}
34      <tr>
35        <td>{{p['team']}}</td>
36        <td>{{p['description']}}</td>
37        <td>{{p['lab']}}</td>
38        <td>{{p['time']}}</td>
39        {% if not current_user.is_anonymous %}
40          <td>
41          {% if p['project_id']== current_user.project_id %}
42          <a href='{{url_for("delete_project") }}'>delete</a>
43            <a href='{{ url_for("edit_project") }}'>edit</a>
44          {% endif %}
45          </td>
46        {% endif %}
47    </tr>
48    {% endfor %}
49  </table>
```

```
 9 @app.route('/')
10 @app.route('/index')
11 def index():
12   print('index')
13   if current_user.is_authenticated:
14     projects = get_all_projects()
15   else:
16     projects = []
17   return render_template('index.html', projects=projects)
```

Jinja Control Statements

- The syntax for control statements is to use `{% braces %}`.
- Conditionals use `if`, `else`, `elif`, as well as `endif`, since whitespace scoping doesn't work for html.
- We can also use for and while loops for iterating through collections.

app/templates/index.html: Conditional statement in template

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog!</title>
        {% endif %}
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

app/routes.py: Fake posts in view function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html', title='Home', user=user, posts=posts)
```

app/templates/index.html: for-loop in template

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog</title>
        {% endif %}
    </head>
    <body>
        <h1>Hi, {{ user.username }}!</h1>
        {% for post in posts %}
        <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
        {% endfor %}
    </body>
</html>
```

Jinja Inheritance

- Since we often want the titles, menus, footers in an application to be the same, we can have the templates inherit from each other.
- The `block xxxx` is left unspecified for other templates to fill in, and they can extend the ase template by just specifying how they would fill in `xxxx`

app/templates/base.html: Base template with navigation bar

```
<html>
    <head>
      {% if title %}
      <title>{{ title }} - Microblog</title>
      {% else %}
      <title>Welcome to Microblog</title>
      {% endif %}
    </head>
    <body>
        <div>Microblog: <a href="/index">Home</a></div>
        <hr>
        {% block content %}{% endblock %}
    </body>
</html>
```

- This principle is refered to as DRY: *dont repeat yourself*

app/templates/index.html: Inherit from base template

```
{% extends "base.html" %}

{% block content %}
    <h1>Hi, {{ user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Forms

- To build PUT requests, we typically use forms. Flask uses the WTForms module to validate Post Requests
- Install `flask-wtf` with pip and create a new file in app, `forms.py`
- There are three parts to the form: the form class, the template containing the form, and the route for processing the form.
- There are numerous validators available including regular expressions, and checking two fields are equal.

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'you-will-never-guess'
# ... add more variables here as needed
```

Flask apps should have a secret key to protect against cross site request forgery (CSRF). You can set in app.py, but there are better ways.

app/forms.py: Login form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

Rendering Forms

- Jinja works with `flask-wtf` to put the appropriate input elements in the page.
- The form.hidden_tag() is used to protect against CSRF attacks
- The form elements are defined by the `forms.py` class
- Attributes can be appended to the elements in brackets.
- If a form doesn't validate, the errors are accesisible in a list, but are rendered server side. Faster client side validation can be applied using javascript.
- The `url_for()` maps back from the function name to the route.

```
{% extends "base.html" %}

{% block content %}
<h2>Login</h2>

  <form name='login' action='' method='post'>
    <div class='form-group'>
      {{form.hidden_tag()}}
      <p>
      {{ form.student_number.label }}<br>
      {{ form.student_number(size=8) }}
      {% for error in form.student_number.errors %}
      <span style="color:red;">[{{ error}}]</span>
      {% endfor %}
      </p>
      <p>
      {{ form.pin.label }}<br>
      {{ form.pin(size=4) }}
      {% for error in form.pin.errors %}
      <span style="color:red;">[{{ error}}]</span>
      {% endfor %}
      </p>
      <p> {{form.remember_me() }} {{form.remember_me.label }}</p>
      <p> {{ form.submit() }}</p>
    </div>
  </form>
  <p>To register <a href={{ url_for('register') }}>click here</a></p>
  {% endblock %}
```

```
<h2>Login</h2>

  <form name='login' action='' method='post'>
    <div class='form-group'>
      <input id="csrf_token" name="csrf_token" type="hidden" value="ImU2NzU5ODlhMDg2YWE3NzE4ZW
      <p>
      <label for="student_number">Student Number</label><br>
      <input id="student_number" name="student_number" required size="8" type="text" value="">

      </p>
      <p>
      <label for="pin">Pin Code</label><br>
      <input id="pin" name="pin" size="4" type="password" value="">

      </p>
      <p> <input id="remember_me" name="remember_me" type="checkbox" value="y"> <label for="rem
      <p> <input id="submit" name="submit" type="submit" value="Sign In"></p>
    </div>
  </form>
  <p>To register <a href=/register>click here</a></p>
```

Processing Forms

- To process a form, we configure a route for the `POST` method.
- We define an instance of the form class, for both rendering and wrapping posted data.
- A `GET` request won't validate, so it will jump to the last line, and render the page.
- If a `POST` request validates, a flash message is created, and the page is redirected to the index.
- The flash messages are just a list that can be accessed by other pages.
- To actually check a users passwords, we need a database (next lecture).

app/routes.py: Receiving login credentials

```
from flask import render_template, flash, redirect

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for user {}, remember_me={}'.format(
            form.username.data, form.remember_me.data))
        return redirect('/index')
    return render_template('login.html', title='Sign In', form=form)
```

```
    <body>
        <div>
            Microblog:
            <a href="/index">Home</a>
            <a href="/login">Login</a>
        </div>
        <hr>
        {% with messages = get_flashed_messages() %}
        {% if messages %}
        <ul>
            {% for message in messages %}
            <li>{{ message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
        {% endwith %}
```

App Configuration

- Storing the secret key in a source file isn't a good idea. Secret keys and user credentials should always be manually configured, and never part of the repository. Setting them as system vairables is a good approach.
- Create a configuration file to store all config variables.
- This can then be loaded when the app runs.
- The environment variables can also store database locations and credentials, and keys for third party services

config.py: Secret key configuration

```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY')
```

app/__init__.py: Flask configuration

```
from flask import Flask
from config import Config

app = Flask(__name__)
app.config.from_object(Config)

from app import routes
```

```
(virtual-environment) drtnf@drtnf-ThinkPad:$ export SECRET_KEY='poor_secret'
(virtual-environment) drtnf@drtnf-ThinkPad:$ echo $SECRET_KEY
poor_secret
(virtual-environment) drtnf@drtnf-ThinkPad:$ flask shell
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
App: app [production]
Instance: /Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/instance
>>> print(app.config['SECRET_KEY'])
poor_secret
>>>
```

Debugging and the Flask Shell

THE UNIVERSITY OF WESTERN AUSTRALIA

- The Flask shell is a useful way to test small functions and there integration with flask, without using a browser.
- It loads the flask app, and all the dependencies, but doesn't need the server running. You can set the shell context to have variable predefined when you start the shell.
- Debug mode is also very useful. Set the system variable `FLASK_DEBUG=1` to get a trace of the errors when the server crashes.

```
(venv) $ export FLASK_DEBUG=1
```

```
1 from app import app, db
2 from app.models import Student, Project, Lab
3
4 @app.shell_context_processor
5 def make_shell_context():
6   return {'db':db, 'Student':Student, "Project":Project, 'Lab':Lab}
```

```
(virtual-environment) drtnf@drtnf-ThinkPad:$ export SECRET_KEY='poor_secret'
(virtual-environment) drtnf@drtnf-ThinkPad:$ echo $SECRET_KEY
poor_secret
(virtual-environment) drtnf@drtnf-ThinkPad:$ flask shell
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
App: app [production]
Instance: /Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/instance
>>> print(app.config['SECRET_KEY'])
poor_secret
>>>
```

builtins.NameError

```
NameError: name 'FlaskForm' is not defined
```

Traceback (most recent call last)

File "/Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/virtual-environment/lib/python3.6/site-packages/flask/_compat.py", line 35, in reraise
  raise value

File "/Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/pair-up.py", line 1, in <module>
  from app import app, db

File "/Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/app/__init__.py", line 14, in <module>
  from app import routes,models

File "/Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/app/routes.py", line 4, in <module>
  from app.forms import LoginForm, RegistrationForm, ProjectForm

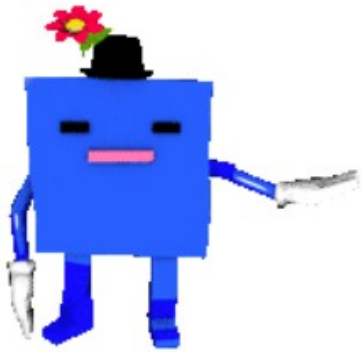
File "/Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/app/forms.py", line 7, in <module>
  class LoginForm(FlaskForm):

NameError: name 'FlaskForm' is not defined

Suggested Reading

Read “What is Code” by Paul Ford:

http://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/



There are bugs in your code! Click the line of code that looks like it’s bug-free. But be careful: Any time you don’t fix a bug, a new one is born.

```
var salesPlusFour = 4 + sales;
```

```
var salesPlusFour = "4" + sales;
```

```
for (var i = 0; i < 10; i++)
```

```
for (var i = 0; i < 10 i++)
```