THE UNIVERSITY OF WESTERN AUSTRALIA

Flask and Server-Side Rendering

CITS3403 and CITS5505 - Agile Web Development

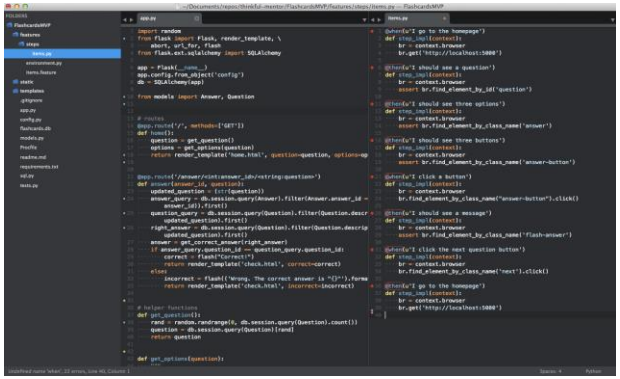
Adapted from the Flask Mega-Tutorial, by Miguel Grinberg:
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial

Semester 1, 2024

Full-stack development

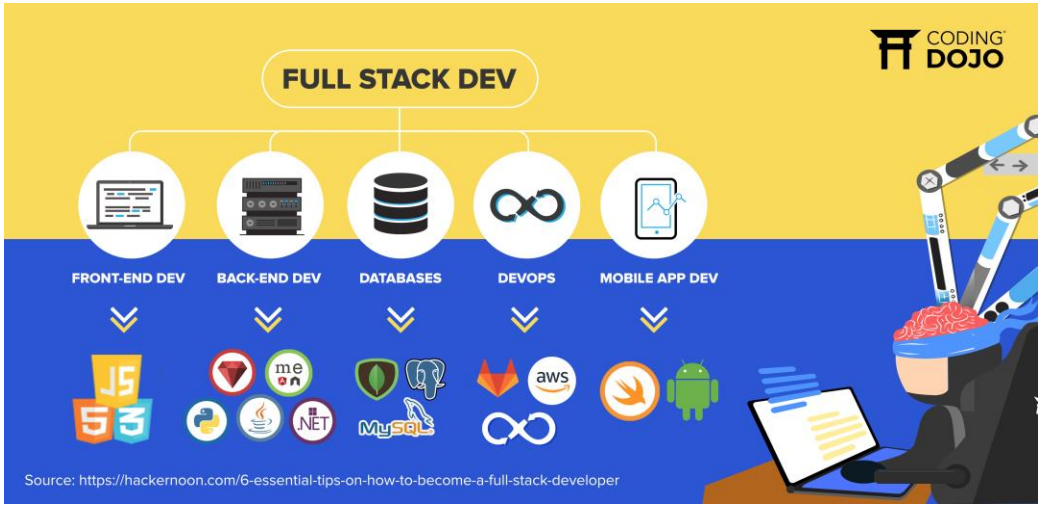
What is a web server?

- Up until now, we have only looked at programs running in the browser, but we will now shift focus to programs running on the server.
- A web server is simply a program running on a computer connected to the internet that responds to requests from other computers on the network.
- A lot of server web development is done from the command line, since traditionally servers didn't need a graphical front end.
- By now, everyone 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.



Full-stack development

- So far, we have looked at front-end technologies for displaying static pages in the browser.
- We will now look at back-end web development, which makes up the programs that run on the server (databases, application logic, etc.)



- A stack refers to a complete list of technologies required to implement both a front-end and a backend.

Different stacks

- There are various backend "stacks" people use to develop:
  - LAMP (Linux, Apache, MySQL and PHP)
  - Ruby on Rails
  - Django (Python)

- Full-stack development involves knowing a full set of technologies used for both front-end and the back-end. Most developers are specialised in one part of the stack.

- We're going to use several tools for the backend 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: provides the method for transmitting data between the browser and the server after the page has been loaded.
  - jQuery: provides nice syntax and bindings for AJAX requests on the client side.

Flask basics

Getting started with Flask

THE UNIVERSITY OF WESTERN AUSTRALIA

- Flask is a micro-framework that can be used to create a server program that will run on any machine and has few dependencies.
- You will require python3 and pip installed in your operating environment.
- Initialize a new Python virtual environment and activate it.
- Now install Flask. Any required modules will be kept in the virtual-environment.

```
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
```

- You can now run flask by typing `flask run`, but the app doesn't know what to run.

Basic Flask application structure

THE UNIVERSITY OF WESTERN AUSTRALIA

- The basic "Hello world" Flask application is a very simple `app.py` file.

```
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

- The file has a method to return 'Hello world!' that is *decorated* with @app.route('/').

- The variable `app` is an instance of the class `Flask`. When it runs it listens for requests, and if the route matches a function's decorator, it executes that function.

- The return of the function becomes the response.

Running a Flask application

- If you create the `app.py` file and then run Flask again.

```
(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)
```

- You can then use a browser to see your app in action! (http://localhost:5000)

- Although it seems similar, to the "Hello world!" example we had at the start of the course when we were first learning HTML, it is very different.

- Instead of the browser simply opening a local HTML file, the browser is now making a request to a server program (which happens to be on the same computer) and rendering the result.

A more scalable application structure

THE UNIVERSITY OF WESTERN AUSTRALIA

- But this single file structure doesn't scale well.
- A better structure is to create a Python package that will contain all the code we need for the web app.
- The __init__.py file creates an instance of the Flask class and routes.py contains the request handlers.
- Finally, we need a file at the top level to import the app.
- Now the `app` package can contain files for handling routes, modules, templates, tests and anything else required.

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
```

Flask endpoints

THE UNIVERSITY OF WESTERN AUSTRALIA

- The functions in `routes.py` annotated with @app.route(…) are known as endpoints.
- An endpoint can be the destination of multiple URLs:

```
@app.route('/')
@app.route('/index')
def hello(userid):
    return 'Hello world!'
```

- They can also accept parameters using the `<variable>` syntax which then get automatically passed to the Python function:

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username
```

- Non-string parameters can be parsed automatically by using a converter:

```
@app.route('/user/<int:userid>')
def show_user_profile(userid):
    # show the user profile for that user
    return 'User %d' % userid
```

Endpoints, HTTP methods and url_for

THE UNIVERSITY OF WESTERN AUSTRALIA

- By default, endpoints are only available as GET requests.

- However, you can set them as end-points of various types using the `methods` parameter:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

- The endpoints form the backbone of your web application, and you will frequently want to link to them in HTML. However, you don't want to have to duplicate the long URLs everywhere in your program.

- Therefore, you can use the function url_for to retrieve the URL for a given endpoint:

Getting endpoint URLs with url_for

THE UNIVERSITY OF WESTERN AUSTRALIA

- The endpoints form the backbone of your application, and you will use them frequently! However, you don't want to be writing the URLs everywhere in your application:
  - The URLs can be very long.
  - If the files in your directories change, the URL has to change.
  - If you change a URL, you want to change it one place only.

- Therefore, you can use the function `url_for` to retrieve the URL given the name of an endpoint:

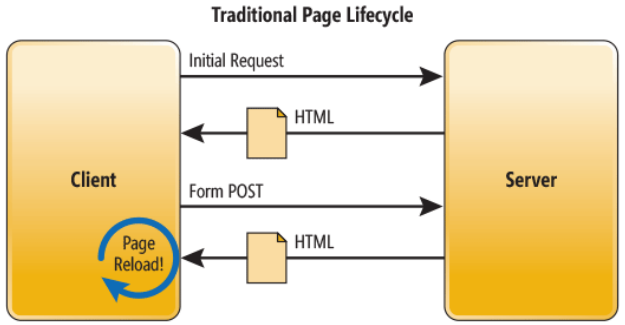
```
@app.route('/looping', methods=['GET', 'POST'])
def loop():
    thisURL = url_for('loop')
    return "<a href={}> Let's go back to {} </a>".format(thisURL, thisURL)
```

Let's go back to /looping

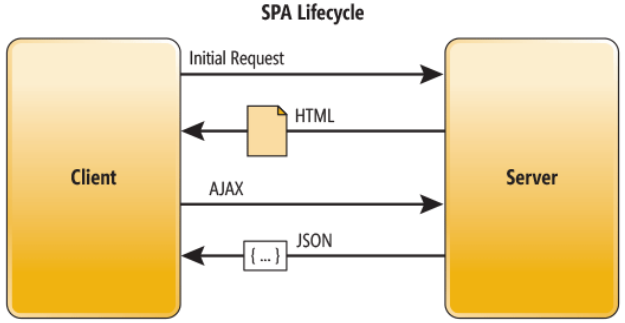
- Next question: how do we return complex web-pages from the server?

Server-side vs Client-side rendering

- There are two approaches to serving dynamic HTML:
  1. Server-side rendering - the server builds the HTML when it receives the request and sends it to the client.



  2. Client-side rendering - the server sends JavaScript and an HTML skeleton to the client, and the client can then request JSON and build the HTML using AJAX and jQuery.



- Server-side rendering is the traditional approach. However, client-side rendering is more flexible and allows greater support for non-browser devices. Flask supports both!

Flask server-side rendering

Basic server-side rendering

THE UNIVERSITY OF WESTERN AUSTRALIA

- The most basic approach to server-side rendering listens for requests and uses Python functions to directly build HTML pages to return as a response.

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>'''
```

- However, this mixes the logic and the presentation!

Server-side templates

THE UNIVERSITY OF WESTERN AUSTRALIA

- A better approach is to use HTML templates that references variables, and a rendering function that will take a template and an assignment to those variables and builds the HTML dynamically.

```
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>
```

- Flask uses Jinja for this task, but there are many alternatives (pug, handlebars)

Getting started with Jinja

THE UNIVERSITY OF WESTERN AUSTRALIA

- We separate presentation and logic by having a template directory that contains HTML files annotated with placeholder variables distinguished by `{{ ... }}` braces.

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)
```

- When a request is received Flask will look for the matching template (in the directory templates) and convert the template to pure HTML using `render_template` function.

- The values of the named arguments to this function are then substituted by Jinja into the locations indicated by the matching placeholder variables.

Dynamically Jinja templates

THE UNIVERSITY OF WESTERN AUSTRALIA

- 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 the 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 conditional control statements

THE UNIVERSITY OF WESTERN AUSTRALIA

- The syntax for control statements is `{% ... %}`.

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>
```

- Conditionals use `if`, `else`, `elif`, as well as `endif`, since whitespace scoping doesn't work for HTML.

Jinja loop control statements

THE UNIVERSITY OF WESTERN AUSTRALIA

- We can also use `for` and `while` loops for iterating through collections.

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

THE UNIVERSITY OF WESTERN AUSTRALIA

- Since we often want the titles, menus, footers in an application to be the same, we can have the templates inherit from each other.

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 is done using the `{% block <blockName> %}` construct which indicates that the named block is left unspecified for other templates to fill in.

Jinja inheritance

- You can indicate that one template inherits from another by using the following syntax:

  {% extends <parentTemplateName> %}

- The contents of each named block can then be provided using the same block syntax:

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

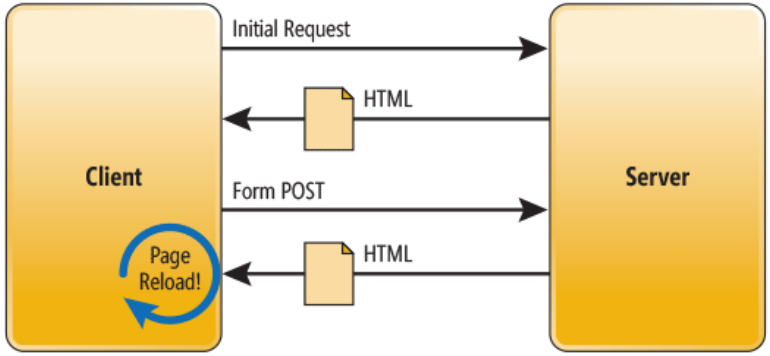
```
{% extends "base.html" %}

{% 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 %}
```

- This is an instance of the general DRY principle: *don't repeat yourself*

Flask forms

Forms on the server

- Forms are used to move data from the client to the server, and when you hit the "Submit" button builds a POST request and sends it to the server.



- The server then (hopefully!) does something with the data and then returns a response. Often this involves directing the browser to navigate to a new page.

- Flask uses the WTForms module to both insert the form into the Jinja template and then to validate POST requests from the rendered form. Install `flask-wtf` with `pip`.

Securing the form

- Web forms are vulnerable to attacks known as cross site request forgery (CSRF).
- The exact mechanism of this attack will be covered in the "Security" lecture.
- For now, it is enough to know that a server should include a secret key in every form it generates.

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

- For now, we will set this in the `app.py`, file
- This is not secure as this file will be pushed to GitHub!

- Again, we will cover better ways to store your secret keys in the "Security" lecture.

Data model for a form

THE UNIVERSITY OF WESTERN AUSTRALIA

- There are three parts to creating a form on the Flask server:
  1. Use WTForms to create a class describing the data model for the fields
  2. Update the Jinja template to describe how the form should render.
  3. Update the routes.py file with a function to handle the submitted form.

- Python classes for forms typically live in a new file `forms.py` in app:

```
from flask_tf import FlaskForm
from wtforms import IntegerField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    student_number = IntegerField('Student ID', validators=[DataRequired()])
    pin = IntegerField('PIN', validators=[DataRequired()])
    remember_me = BooleanField('Remember me')
    submit = SubmitField('Sign in')
```

Rendering a form

- Jinja works with flask-wtf to put the appropriate input elements in the page.
- The `form.hidden_tag()` entry is used to protect against CSRF attacks
- The form elements are those defined by the `forms.py` class
- Attributes can be appended to the elements in brackets.
- The `url_for()` is used to reference the URL for the correct endpoint.

```
{% 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 a form

- To process a form, we configure a route for the POST method.

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)
```

- We create 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.
- To check a user's passwords, we need a database (see "Databases" lecture).

Displaying flash messages

THE UNIVERSITY OF WESTERN AUSTRALIA

- The flash messages are just a list that can be accessed by other pages.

```
<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 %}
```

- If a form doesn't validate, the errors are accessible in a list, but are rendered server side. Faster client-side validation can be applied using JavaScript.
- To check a user's passwords, we need a database (see "Databases" lecture).

Other

Debugging in the Flask shell

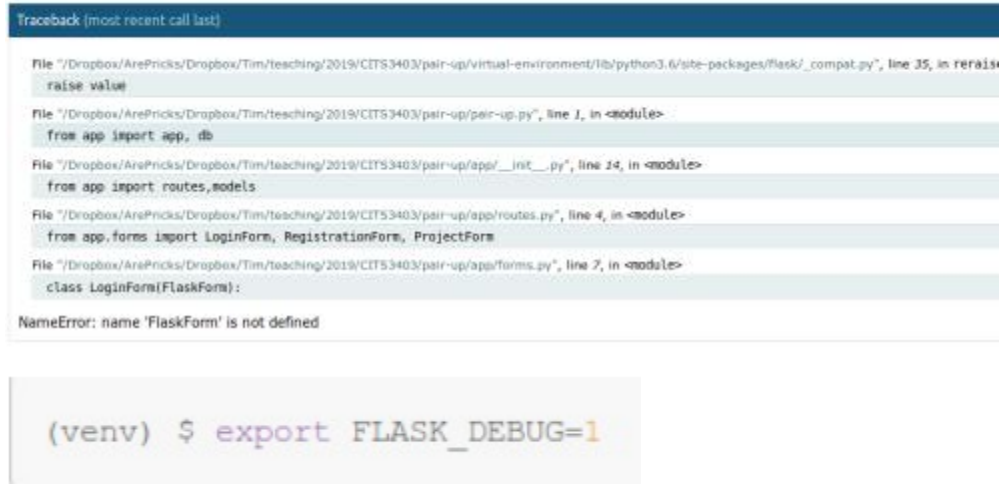
- The Flask shell is a useful way to test small functions and their integration with flask, without using a browser.

- It loads the flask app, and all the dependencies, but doesn't need the server running. You can set the shell context to have variables 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.

```
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
>>>
```

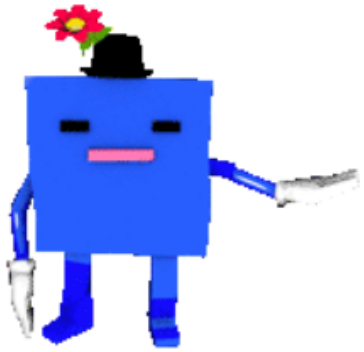



```
(venv) $ export FLASK_DEBUG=1
```

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++)
```