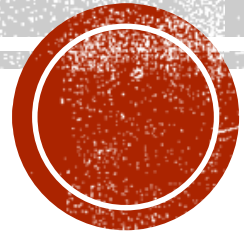
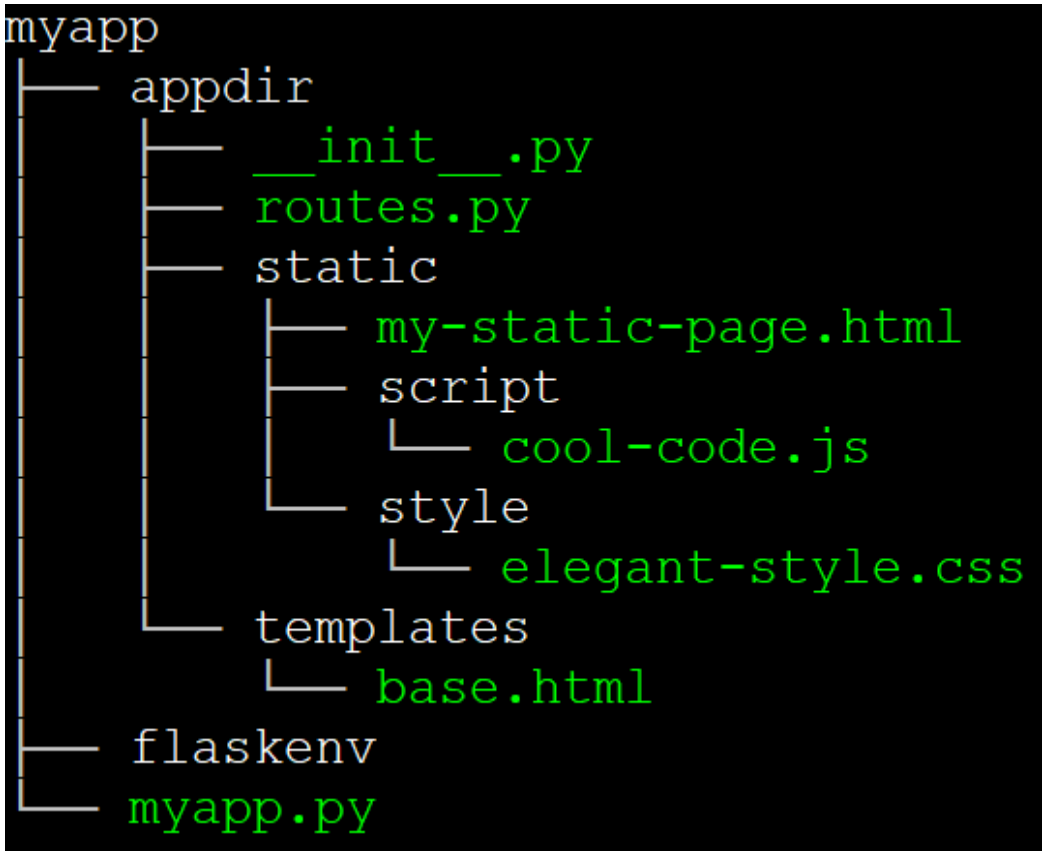
FLASK AND DATABASES



ABOUT THE PROJECT - REMINDER

- Deadline: Last lab of term: 12-December-2019 5pm
- Technologies to use: Flask, Javascript, CSS, HTML
- Domain: Any domain of your choice
- How to submit:
  - All files in the following structure
  - Change file names to suit your website
  - But keep the same directory structure
  - Zip the top-level directory
  - Submit the zip file **only** on Moodle



GRADING SCHEME

- The UCD grading scheme applies by default

Highest	Lowest	Letter
100.00 %	76.00 %	A+
75.99 %	73.00 %	A
72.99 %	70.00 %	A-
69.99 %	66.00 %	B+
65.99 %	63.00 %	B
62.99 %	60.00 %	B-
59.99 %	56.00 %	C+
55.99 %	53.00 %	C
52.99 %	50.00 %	C-
49.99 %	46.00 %	D+
45.99 %	43.00 %	D
42.99 %	40.00 %	D-
39.99 %	36.00 %	E+
35.99 %	33.00 %	E

STATIC FILES

- Make a directory called `static` inside the `blogapp` directory

```
week11>cd microblog/
microblog>ls
blogapp  flaskenv  microblog.py
microblog>cd blogapp/
blogapp>ls
config.py   __init__.py  templates
forms.py    routes.py
blogapp>mkdir static
```

- Put your static html files insides this directory

A SAMPLE PLAIN.HTML

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>A Simple Static Page</title>
        <link rel="stylesheet" href="style/mystyle.css">
    </head>
    <body>
        <h3>A headline of no consequence</h3>
        <p>Some paragraph with not much information. Purely here for
        decorative purposes.
        </p>
    </body>
</html>
```

ADD IT TO BASE TEMPLATE

```
<body>
    <div>
        Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        <a href="{{ url_for('login') }}">Login</a>
        <a href="{{ url_for('static', filename='plain.html') }}">Static Page</a>
    </div>
    <hr>
```

ADDING CSS TO `base.html`

- CSS is also a static file (The procedure for Javascript is the same)
- The procedure is the same:
  - Create a directory inside `static` called `style`
  - Add your css file here
  - Call it using `url_for` inside your base template (or whichever template you want)

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - microblog</title>
        {% else %}
        <title>microblog</title>
        {% endif %}
        <link rel="stylesheet" href="{{ url_for('static', filename='style/mystyle.css') }}">
    </head>
    <body>
```

DATABASES IN FLASK

- Python supports many databases, both relational and non-relational
- Flask, deliberately, does not make a choice of databases, so you can choose whichever one you want
- We will use Flask-SQLAlchemy extension
  - Wrapper around the SQLAlchemy package, which is an ORM
- ORM or Object Relational Mapper allows applications to manage a database using high-level constructs like classes, objects and methods, instead of SQL and tables
- SQLAlchemy supports multiple database engines, including MySQL, PostgreSQL and SQLite

INSTALL FLASK-SQLALCHEMY

```
(flaskenv) microblog>pip install flask-sqlalchemy
Building wheels for collected packages: SQLAlchemy
  Running setup.py bdist_wheel for SQLAlchemy ... error
  Complete output from command "/c/Users/vivek/OneDrive - University College Dubl
in/ucd/2019/teaching/bdic/web-app-dev/lecture-slides/sample-code/week9/microblog/
flaskenv/bin/python" -u -c "import setuptools, tokenize;__file__='/tmp/pip-build-
elpy7dy5/SQLAlchemy/setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.
read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" bdi
st_wheel -d /tmp/tmpr7t0jyqepip-wheel- --python-tag cp37:
  usage: -c [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
     or: -c --help [cmd1 cmd2 ...]
     or: -c --help-commands
     or: -c cmd --help

  error: invalid command 'bdist_wheel'

  ----------------------------------------
  Failed building wheel for SQLAlchemy
  Running setup.py clean for SQLAlchemy
Failed to build SQLAlchemy
Installing collected packages: SQLAlchemy, flask-sqlalchemy
  Running setup.py install for SQLAlchemy ... done
Successfully installed SQLAlchemy-1.3.10 flask-sqlalchemy-2.4.1
```

DATABASE: SQLITE

- Since, we're using Python 3.7+, sqlite is already included.
- We do not need to install it
- Sqlite has no installation procedures, no server needs to be run or configured
- Sqlite creates a database in an ordinary file, which can be seen on the filesystem
- If the file has read permissions, then sqlite can read the database
- If the file has write permissions, then sqlite can write into the database
- Database files can easily be backed up, by simply copying to a USB stick or shared via email

CONFIGURATION OF DATABASE

Change the `config.py` to reflect where the database will be

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'

    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'blogdb.db')

    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

EXPLANATION

- The Flask-SQLAlchemy extension takes the location of the database from the `SQLALCHEMY_DATABASE_URI` variable

- Being good software engineers, we give it a default value in case there is no environment variable called `DATABASE_URL`

- In this case, we have given it the location of an sqlite database called `blogdb.db`, which is located in the main directory of the application, stored in the variable called `basedir`

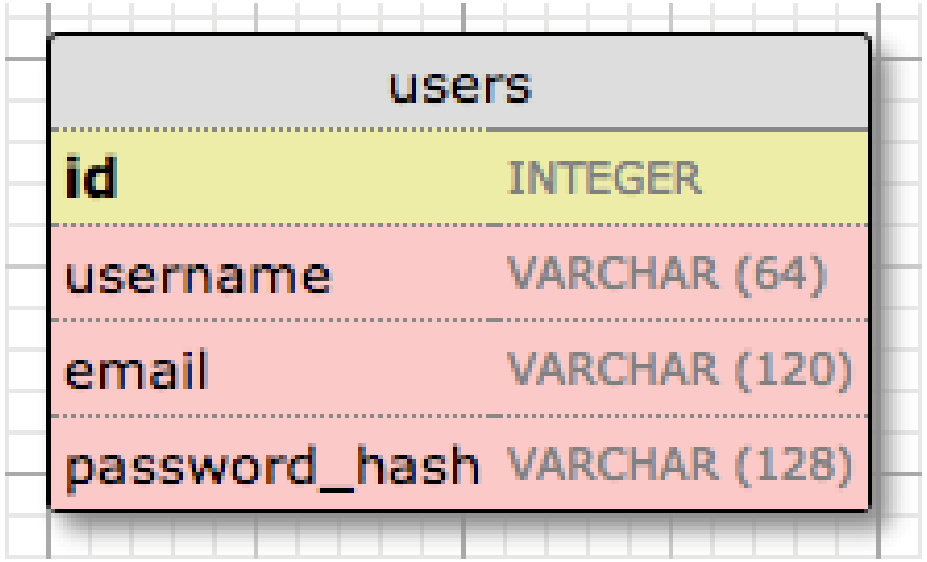
NOW, MODIFY `__init__.py`

```
from flask import Flask
from blogapp.config import Config
from flask_sqlalchemy import SQLAlchemy

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

from blogapp import routes, models
```

EXPLANATION

- Now, we need to add the database to the application and make it aware that there is a database that we are going to us
- The database is going to be represented in the application by a *database instance*
- Finally, we also import a new module called '`models`'. This will define the structure of the database
- These objects are created immediately after the application is created. So we modify the `__init__.py` file in the `blogapp` directory

THE FIRST TABLE: USERS

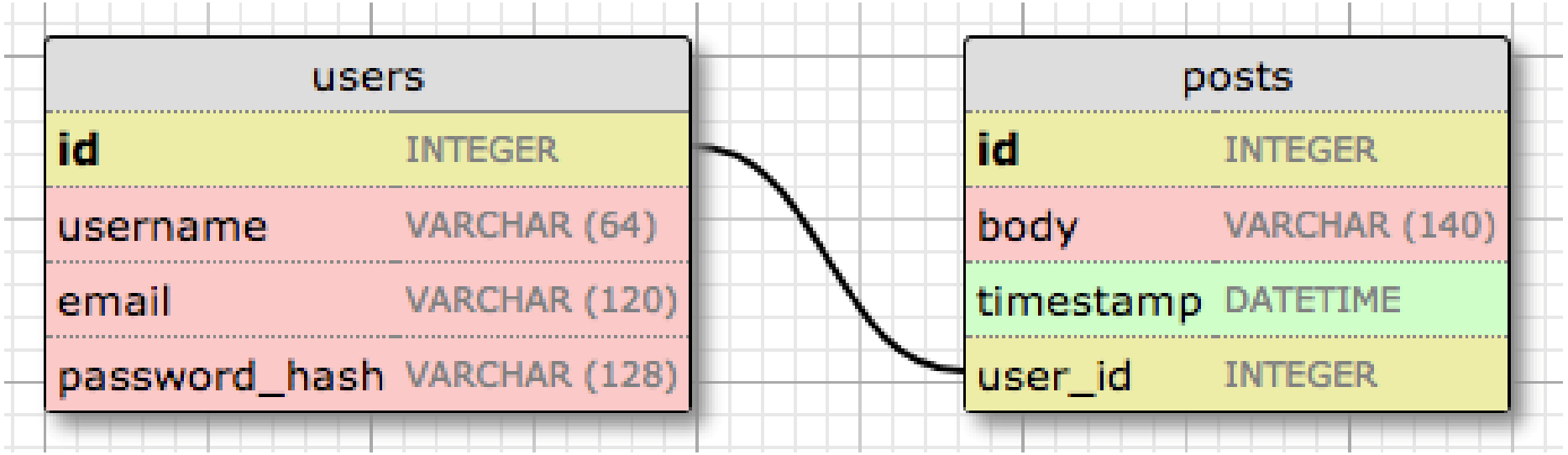


id – Primary key (Mostly automatically assigned by database)
username – a string with max length of 64
email – a string with a max length of 120
password_hash – store hash of the user's password (NEVER store the actual password)

CREATE models.py

```
from blogapp import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

SECOND TABLE - POSTS



The Posts table has a *foreign key* called 'user_id' which links a post to the user.

MODIFY models.py

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

MODIFY models.py

```
from datetime import datetime
from blogapp import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

Note: posts is not an actual field. It is just a relationship

NOW INITIALIZE DATABASE

- We should not initialize the database from within the application. Why?
- So we run a flask shell

```
microblog>source flaskenv/bin/activate
(flaskenv) microblog>flask shell
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
App: blogapp [production]
Instance: /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teac
hing/bdic/web-app-dev/lecture-slides/sample-code/week11/microblog/instance
>>>
```

- Note: the shell now knows that it is running a Flask application

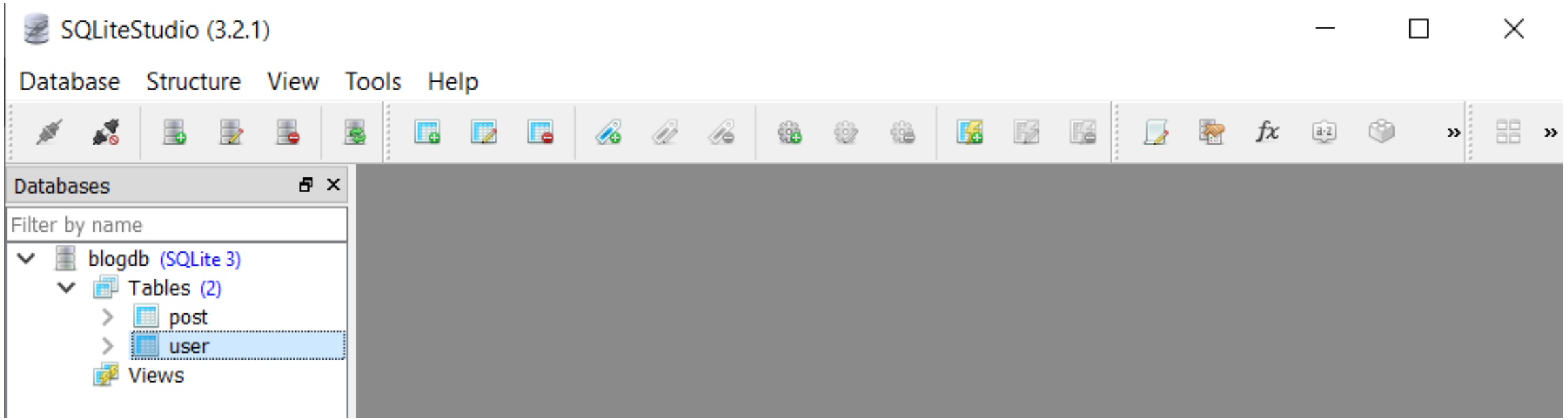
INITIALIZE THE DATABASE – II

- From the application, we first `import db`
- And then call, `db.create_all()`
- At this point, in your filesystem, you should see a file called `blogdb.db`

```
(flaskenv) microblog>flask shell
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
App: blogapp [production]
Instance: /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teac
hing/bdic/web-app-dev/lecture-slides/sample-code/week11/microblog/instance
>>> from blogapp import db
>>> db.create_all()
```

VIEW THE DATABASE – OPTIONAL!!

- There is a database manager GUI that is available to view the database
- If you want, you can download it for your platform at: https://sqlitestudio.pl/index.rvt?act=download
- You can view that the database has been created, along with tables inside it



TEST THE DATABASE

- You can test if the database is working from the flask shell, by importing classes User and Post

```
(flaskenv) microblog>flask shell
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
App: blogapp [production]
Instance: /c/Users/vivek/OneDrive - University College Dublin/ucd/2019/teac
hing/bdic/web-app-dev/lecture-slides/sample-code/week11/microblog/instance
>>> from blogapp import db
>>> db.create_all()
>>> from blogapp.models import User, Post
```

TEST THE DATABASE – II

- Create a couple of users, and commit them to the database

```
>>> from blogapp.models import User, Post
>>> u1 = User(username='vivek', email='vivek.nallur@ucd.ie')
>>> u2 = User(username='dave', email='dave@web.com')
>>> db.session.add(u1)
>>> db.session.add(u2)
>>> db.session.commit()
```

- Now query the database and print them out to see if they exist

```
>>> users = User.query.all()
>>> for u in users:
...     print(u.email, u.username)
...
vivek.nallur@ucd.ie vivek
dave@web.com dave
```

TEST THE DATABASE – III

- Create a post and link it to a user

```
>>> u = User.query.get(1)
>>> p = Post(body='Flask!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

- Note the '`author`'. It was not in the `Post` class (or table)!
- SQL-Alchemy creates this virtual key from our definition of relationship between `User` and `Post` [see `db.Relationship` on slide #19]

TEST THE DATABASE – IV

- Check if you can access the post from the user

```
>>> for u in users:
...     print (u.username + " has the following posts: ")
...     for p in u.posts.all():
...         print(p.body)
...
vivek has the following posts:
Flask!
dave has the following posts:
```

- Note: We are now accessing the database tables using Python code, with no SQL!!

CLEANING UP

- Now, we clean up the database so that we can return to our Flask application
- We delete all the users and posts and commit the transaction

```
>>> for u in User.query.all():
...     db.session.delete(u)
...
>>> for p in Post.query.all():
...     db.session.delete(p)
...
>>> db.session.commit()
```

MORE THINGS TO DO WITH QUERIES

CODE TO ADD A USER

- As has been our pattern so far, we have to modify two files:
  - `routes.py` → So that we can create a new URL and handle data coming in
  - `forms.py` → So that we can create a new class to store the data from the web-page

And create one new template:

- `register.html` → So that we can display a new web-page to the user

CODE FOR forms.py

```
class SignupForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField('Repeat Password', validators=[DataRequired()])
    accept_rules = BooleanField('I accept the site rules', validators=[DataRequired()])
    submit = SubmitField('Register')
```

CODE FOR routes.py

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    form = SignupForm()
    if form.validate_on_submit():
        if form.password.data != form.password2.data:
            flash('Passwords do not match!')
            return redirect(url_for('signup'))
        user = User(username=form.username.data, email=form.email.data)
        db.session.add(user)
        db.session.commit()
        flash('User registered with username:{}'.format(form.username.data))
        return redirect(url_for('login'))
    return render_template('signup.html', title='Register a new user', form=form)
```

CODE FOR signup.html

```
<p>
    {{ form.password2.label }}<br>
    {{ form.password2(size=35) }}

    {% for error in form.password2.errors %}
    <span style="color: red;">[{{ error }}]</span>
    {% endfor %}

</p>
<p>{{ form.accept_rules() }} {{ form.accept_rules.label }}</p>
    {% for error in form.accept_rules.errors %}
    <span style="color: red;">[{{ error }}]</span>
    {% endfor %}
<p>{{ form.submit() }}</p>
```

MODIFY base.html

```
<div>
    Microblog:
    <a href="{{ url_for('index') }}">Home</a>
    <a href="{{ url_for('signup') }}">Signup</a>
    <a href="{{ url_for('login') }}">Login</a>
    <a href="{{ url_for('static', filename='plain.html') }}">Static Page</a>
</div>
<hr>
```

TO-DO IN CLASS

- Download the code shown in the class from Moodle and make it work on your machine
- Modify base.html to add another link to a static file that includes a javascript file
- Modify mystyle.css to change the style of all forms displayed on the site (make any modification you like). Check that all forms now show a different style

- You can see more things to do with queries at:

http://flask-sqlalchemy.palletsprojects.com/en/2.x/queries/