CS6
Practical
System
Skills
Fall 2019 edition
Leonhard Spiegelberg
lspiegel@cs.brown.edu
Midterm results & Logistics

⇒ Midterm I average: 78.8%
⇒ Midterm II average: 83%
⇒ Final project presentations on 15th Dec @ CIT 3pm - 5pm
  → room to be defined
  → 15 min presentation + 10 min questions
  → Your TA is there to help you.
All examples available at

github.com/browncs6/FlaskExamples
21.01 User logins in flask

⇒ Website often need to authenticate users, there are several packages available to help achieve this in Flask
- **flask_login** helps to guard routes/require user login for them.
- **Werkzeug** tool to help with hashing password.
- **itsdangerous** generates safe tokens, e.g. for account confirmation or expiring links

⇒ Following slides are based on Chapter 8-9, Flask book
21.02 Login system via Flask

How to store user login information? User + passwd?

⇒ Don't store clear passwords!

⇒ Instead store a hash computed via

\[
\text{hash(password + salt)}
\]

Some cryptographic secure hash function (use a well-tested library)

user supplied password

random value, added for each password
from werkzeug.security import generate_password_hash
from werkzeug.security import check_password_hash

h = generate_password_hash('secret password')

# Result h looks similar to this
# pbkdf2:sha256:150000$QmqUMoy8$f8de105257426cbfb533f9db8ecf85921cd544541ec2df2def8d8ea123b83fc2

check_password_hash(h, 'secret password')
Often we want to assign / get values. Not necessarily does a value need to be represented by a member variable always.

A wide-spread pattern used is a pair of a getter and a setter function.

Using getters/setters allows to use patterns like

1. lazy computation
2. enforcing constraints
3. avoiding redundancy
21.04 Properties in Python

class Pokemon:
    def __init__(self, name):
        self.name = name
        self.setCategory('unknown')

    def __repr__(self):
        return '{self.name}[self.category]'.format(self.name, self.category)

    def getCategory(self):
        return self._category

    def setCategory(self, category):
        self._category = category

    category = property(getCategory, setCategory)

allows us to call the getter/setter via .category or .category = ...
21.04 Properties in Python with decorators

⇒ instead of using `property(...)`, you can also use `@property` and `@value.setter` to declare them

```python
class Pokemon:
    def __init__(self, name):
        self.name = name
        self.category = 'unknown'

    def __repr__(self):
        return '{}[{}]'.format(self.name, self.category)

@property
def category(self):
    return self._category

@category.setter
def category(self, category):
    self._category = category
```

note the naming convention here
21.05 A simple password model w. properties

class User(db.Model):

    id = db.Column(db.Integer(), primary_key=True)
    password_hash = db.Column(db.String(128))

@property
def password(self):
    raise AttributeError('password is write-only')

@password.setter
def password(self, password):
    self.password_hash = generate_password_hash(password)

def verify_password(self, password):
    return check_password_hash(self.password_hash, password)
21.06 Flask_login

⇒ Flask extension which helps to protect routes & automate everything related to user authentication

⇒ documentation: https://flask-login.readthedocs.io/en/latest/

⇒ support for remember_me cookies builtin

⇒ protect routes by adding @login_required decorator! E.g.,

```python
@app.route('/
@login_required
def index():
    return 'Hello world'
```
21.07 Flask login

⇒ Setup a default path which is displayed to login, via login_view

```python
from flask_login import LoginManager, login_required, login_user, logout_user

app = Flask(__name__)

login_manager = LoginManager(app)
login_manager.login_view = 'login'  # route or function where login occurs...

@app.route('/login')
def login():
    ...
    user = User(...)
    login_user(user, remember=True)
    ...

@app.route('/')
@login_required
def index():
    ...
```
21.07 Flask login user model

⇒ flask login requires a user class to implement several properties/methods:

⇒ derive from UserMixin class to implement useful defaults

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_authenticated</td>
<td>True if user has valid credentials, False otherwise</td>
</tr>
<tr>
<td>is_active</td>
<td>True if user is allowed to login (i.e. use to confirm an account or block it)</td>
</tr>
<tr>
<td>is_anonymous</td>
<td>False for regular users, True for special anonymous user</td>
</tr>
<tr>
<td>get_id()</td>
<td>must return a unique identifier for each user, encoded as str</td>
</tr>
</tbody>
</table>
class User(UserMixin, db.Model):

    id = db.Column(db.Integer(), primary_key=True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))

@property
    def password(self):
        raise AttributeError('password is write-only')

@password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
21.07 Writing the login logic

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()

        if user is not None and \
            user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            return redirect(url_for('secret_page'))

    flash('invalid username or password.')

    return render_template('login.html', form=form)
```

- **query user info via SQLAlchemy**
- **password check**
- **login**
Demo!
Deploying Flask
21.08 Why do we care?

* Serving Flask app "login" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat

⇒ Flask’s builtin webserver is provided for development purposes only.
⇒ Serves one request at a time. What about multiple ones?
WSGI = Web Server Gateway Interface (pronounce: whiskey)

⇒ There are multiple python frameworks (e.g. Flask, Django, Tornado, …) and multiple options for production servers (e.g. Gunicorn, uWSGI, Gevent, Twisted Web, …)

⇒ WSGI is a standard protocol/interface for a production web server to communicate with your web application.
21.06 Web production serves

⇒ There exist multiple production webservers for a web application written in Python, we'll be using gunicorn (Green unicorn)

⇒ easiest way to deploy flask, is to run

```
gunicorn project:app
```

name of your application, e.g. here project.py or project/

the app object created via Flask(__name__)

More information: flask.palletsprojects.com/en/1.1.x/deploying/wsgi-standalone/
### 21.06 Gunicorn - options

Gunicorn provides many options (check via `-h` / `--help`), most important are:

- `-w 4` specify how many worker processes to use
  - **Formula:** $2 \times \text{CPU cores} + 1$
  - (long option: `--worker`)

- `-b 127.0.0.1:4000` specify to which address/port to bind
  - (long option: `--bind`)

- `-e key=value` set environment variable key to value
  - (long option: `--env`)
21.07 The pain of actual deployment

When trying to deploy an actual application, it's a pain because

- Multiple frameworks, multiple versions, …
- different compiler versions, OS versions, …

⇒ How to package, how to deploy?

⇒ two popular solutions
- Virtual Machines
- Containers → we'll be using containers!
21.08 Docker

⇒ allows to package applications with all dependencies into an image

⇒ run via "lightweight virtual machine", however uses less space and memory than a real VM because OS is shared amongst multiple containers
21.09 Creating a container

⇒ containers/images are defined via a Dockerfile

⇒ general usage: `docker COMMAND PARAMS`

  → `docker COMMAND --help` to get help for `COMMAND`

⇒ to create image from a Dockerfile stored in . use

```
docker build -t login:latest .
```

specify a name and tag for this image, format is `name:tag`
21.10 Listing & running images

⇒ to get an overview of created images, use
   docker images

⇒ to start an image use docker run IMAGE
   → there are multiple helpful options when starting a container
   → quit via Ctrl + C or docker stop CONTAINER
   → to get list of running containers, use docker ps
## 21.11 Running containers

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--rm</code></td>
<td>remove container when stopped (else you need to use <code>docker rm IMAGE</code>)</td>
</tr>
<tr>
<td><code>--env ENV / -e ENV</code></td>
<td>pass environment variable to container ⇒ use this for passwords, config etc.</td>
</tr>
<tr>
<td><code>-p &lt;local_port&gt;:&lt;container_port&gt;</code></td>
<td>map local_port to container_port</td>
</tr>
<tr>
<td><code>--publish &lt;local_port&gt;:&lt;container_port&gt;</code></td>
<td>map local_port to container_port</td>
</tr>
<tr>
<td><code>--name NAME</code></td>
<td>give container a name</td>
</tr>
<tr>
<td><code>-v &lt;local_path&gt;:&lt;container_path&gt;</code></td>
<td>mount a volume, i.e. make local_path available within container under container_path</td>
</tr>
<tr>
<td><code>--volume &lt;local_path&gt;:&lt;container_path&gt;</code></td>
<td>mount a volume, i.e. make local_path available within container under container_path</td>
</tr>
</tbody>
</table>
21.12 Starting a shell to work within a container

Sometimes it is useful, to "login" to a container.

⇒ use docker exec -it CONTAINER bash to start an interactive shell session for a specific container

⇒ get CONTAINER via docker ps
21.12 Running postgresql in a container

⇒ Dockerhub provides many prebuilt images, you can get them after registering & logging in via `docker pull`
   → if you want, you can also push your images with `docker push`
   → Note: DO NOT STORE passwords in your Dockerfiles/images!
   → run `docker pull postgres` to get postgres image

To start a postgres database:

1. `mkdir db-data` # create dir where to store data
2. `docker run --name postgres -e POSTGRES_PASSWORD=docker`

⇒ connect via `postgresql://postgres:docker@localhost/postgres`

⇒ more info on the postgres image: https://hub.docker.com/_/postgres
21.13 Packaging a flask app in a docker file

FROM python:3.7

# install requirements as root
COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

# make this available for e.g. flask shell use
ENV FLASK_APP login.py

# run web app as web user
RUN adduser --disabled-password --gecos "web" web

USER web

WORKDIR /home/web

COPY login.py login.py
COPY templates templates
COPY run.sh ./

# runtime
EXPOSE 5000
ENTRYPOINT ["./run.sh"]
21.14 Running Flask & PostgreSQL

1.) Start postgresql container
   ...

2.) Start flask container & link to postgresql container

   docker run --name login -p 8000:5000 \
   --link postgres:dbserver \
   -e DBURI='postgresql://postgres:docker@dbserver/postgres' \
   -e APP_SECRET='test' \
   --rm login:latest
End of lecture.

Next class 3rd Dec: Tue, 4pm @ CIT 477