420-951-VA Transaction Web Applications

Web Programming with Flask

Author: Denis Rinfret

Flask Sessions

- Sessions are used to remember some data between user requests
- By design, the *Hypertext Transfer Protocol (HTTP)*, and the *Internet Protocol (IP)* on which HTTP is built are **stateless**
  - the server doesn't normally retain data about the client
- When a session is created, a special *token* will be saved as a *cookie* on the client side (normally in a web browser)
- The server will associate data to this token and keep it in local storage (server-side)
- When the client makes another request to the server, the session token will be checked by the server to match it with associated data
- Without sessions, it would be harder to maintain secure sessions between the client and the server
  - users surely don't want to re-enter their password on every request

First Session Example

- The first session example is quite simple
  - it is a variation on the *Hello World!* example
- The `/` endpoint is displaying, by default, the simple *Hello Guest!* page
- But if there's a name saved in the session, say for example *Alice*, then it will display the page *Hello Alice!* instead
- To be able to set the name inside the session, use the dynamic endpoint `/name/<name>`
  - it will take the given name and save it in the session, then redirect to `/`

```
from flask import Flask, session, redirect

app = Flask(__name__)
app.secret_key = 'allo'
```

```python
@app.route('/')
def hello_world():
    return 'Hello {}!'.format(session.get('name', 'Guest'))


@app.route('/name/<name>')
def set_name(name):
    session['name'] = name
    return redirect('/')


if __name__ == '__main__':
    app.run()
```

- The `session` object is like a dictionary
  - we can use the square bracket notation `[]` to access or set values associated to some keys
  - `session['name'] = name` sets the value associated to the `'name'` key
  - we could simply use `session['name']` in the `/` endpoint to get the value associated to the name key
    - but if the name key doesn't exist, it will throw a `KeyError`
  - instead, we call the `get` function on the session, and provide a default value
    - `session.get('name', 'Guest')` will get the value associated to the `name` key
    - and if there's nothing associated to it, it will return the default value `'Guest'`

Login Form V1

File `02_app.py`

- The first version of a login form to open an authenticated session has 3 endpoints:
  - `/` : display a welcome message, with either a login or logout link
  - `/login` : GET and POST methods accepted
    - GET: display a login form
    - POST: if the submitted form data validates, then start a session for the user and redirect to `/`
  - `/logout` : clear the session data and redirect to `/`

`LoginForm`

- *Note*: to keep the example simple, some validators have been omitted

```
class LoginForm(FlaskForm):
    username = StringField('username', validators=[InputRequired()])
    password = PasswordField('password', validators=[InputRequired()])
    submit = SubmitField('login')
```

`/login` endpoint

- If the submitted form data is valid, we simply, for now, set the `username` key in the `session` object
- When the user is redirected to `/`, or when a subsequent request is made, it will be possible to get the `username` from the `session` object
- If there is no username in the session, then we assume the user is not logged in
  - this is not the best way to handle logins, it is only the first step to get to a better way

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # should check if username/password pair is valid
        # for now, just accept everything
        session['username'] = form.username.data
        return redirect('/')
    return render_template('login.html', form=form)
```

`/logout` Endpoint

- The only thing we need to do here is to call the `clear` function on the `session` object to clear (remove) all key-value pairs in the session
- Then we redirect to `/`

```
@app.route('/logout')
def logout():
    session.clear()
    return redirect('/')
```

`/` Endpoint

- This endpoint simply renders the `index1.html` template, with the `username` variable set to `session.get('username')`
- In this case, we didn't provide a default value to `get`, so we will get `None` if the username is not set in the session

- this will, again, help us avoid a `KeyError` if the provided key is not in the session
- `None` will evaluate to `False` when it is used in an `if` statement, such as
  - `if username` in a Python file
  - `{% if username %}` in a template

```
@app.route('/')
def index():
    return render_template('index1.html',
                           username=session.get('username'))
```

File `index1.html`

- This template checks if there is a `username` or not
  - if the `username` is `None`, then it will evaluate to `False`
    - so we will jump to the `else` block
    - and a default welcome message with a link to the *login* page will be displayed
  - if the `username` is **not** `None`, then we have a username to work with
    - so we welcome the user by a personalized welcome message
    - then we provide a *logout* link

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Session Examples</title>
</head>
<body>
{% if username %}
<h1>Welcome {{ username }}!</h1>
<a href="/logout">Logout</a>
{% else %}
<h1>Welcome Stranger!</h1>
<a href="/login">Login</a>
{% endif %}
</body>
</html>
```

Login Form V2

File `03_app.py`

- The only change in this version compared to the previous one is how we handle the form data
- Instead of accepting any username/password pair, we have a passwords file containing the registered users' password
- In the `/login` endpoint, if the submitted form data validates, we call the `check_password` function with the provided username/password pair
  - if we have a match, then we set the username in the session as before
  - if not, we *flash* an error message
    - *flash*ed messages will be automatically available in the templates (refer to the `login.html` template below)
    - *flash*ed messages will be displayed just before the login form

`/login` Endpoint

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        if check_password(form.username.data, form.password.data):
            session['username'] = form.username.data
            return redirect('/')
        else:
            flash('Incorrect username/password!')
    return render_template('login.html', form=form)
```

`check_password` Function

```
# not a safe way to store and verify passwords, but it will do for now
def check_password(username, password):
    with open('data/passwords.csv') as f:
        for user in csv.reader(f):
            if username == user[0] and password == user[1]:
                return True
    return False
```

File `passwords.csv`

```
denis, 1234
rob, allo
```

- To verify passwords, we open a CSV file for reading

- The first element of each row is a username
- The second element is the corresponding password
- If both the submitted username and password match the current record in the CSV file, then return `True`
- If not, then keep searching
- If we reach the end of the file without a match, then return `False`

File `login.html`

- The only new thing in this template is the loop through flashed messages
- We do a for loop on all the messages returned by the call to the built-in function `get_flashed_messages`
- And we simply display all of them separated by a break `<br/>`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Session Examples</title>
</head>
<body>
{% for message in get_flashed_messages() %}
{{ message }}<br/>
{% endfor %}
<form action="/login" method="post">
    {{ form.csrf_token }}
    {{ form.username.label }} {{ form.username }}<br/>
    {{ form.password.label }} {{ form.password }}<br/>
    {{ form.submit }}
</form>
</body>
</html>
```

Better Login System: `flask_login` Package

- File `04_app.py`
- As it is the case for most common tasks required to run a website, there exists a package to help out managing logins and sessions
- `flask_login` helps managing logins, users and permissions, without being too restrictive

Step 1: `LoginManager`

- We need to start a `LoginManager`, which we import from `flask_login`, in this way:

```
login_manager = LoginManager()

login_manager.init_app(app)
```

Step 2: `User` class

- We also need to define a `User` class to represent our users
  - the easiest way is to subclass `UserMixin` (imported from `flask_login`)
  - for now, we only need a username, and since we must define an `id` for each user with this package, we set `self.id` to be the username in the constructor

```
class User(UserMixin):
    def __init__(self, username):
        self.id = username
```

Step 3: `load_user`

- We need to define a `load_user` function to retrieve user details from a user id
- For now, a user is defined only by its username, so we create a `User` object with the username provided
  - in a more complete application, the user information will be more detailed (name, email address, profile photo, …)
  - this user information will normally be read from a file or from a database
- The decorator `@login_manager.user_loader` is needed in order to let the login manager know which function to call to load a user

```
@login_manager.user_loader
def load_user(user_id):
    return User(user_id)
```

Step 4: `/login` endpoint

- The `/login` endpoint is very similar to the previous example, except that we need to call the `login_user` function, imported from `flask_login`, to let the login manager know that the user has logged in successfully
- The only other difference, besides the additional flashed message, is how the user is redirected to another page after a successful log in
  - when a user tries to access a protected page without being logged in, the user will be redirected to the login page

- then after a successful login, it is often more convenient to redirect the user to whatever page she was trying to access before being redirected
- the page to be redirected to will be stored in the session, with the key `next`
- so we get the `next` page from the session instead of always redirecting to `/`

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        if check_password(form.username.data, form.password.data):
            login_user(User(form.username.data))
            flash('Logged in successfully.')

            # check if the next page is set in the session by the
        @login_required decorator
            # if not set, it will default to '/'
            next_page = session.get('next', '/')
            # reset the next page to default '/'
            session['next'] = '/'
            return redirect(next_page)
        else:
            flash('Incorrect username/password!')
    return render_template('login.html', form=form)
```

Step 5: protect endpoints with `@login_required`

- `@login_required` is a decorator imported from `flask_login`
- If we want an endpoint to be accessible only to logged-in users, then we *decorate* the endpoint with `@login_required`

```
@app.route('/protected')
@login_required
def protected():
    return render_template('protected.html')
```

- If the user requesting the protected endpoint is not logged in, he will be automatically redirected to the `/login` endpoint
- If already logged in, then it will just go through and execute the endpoint function as normal
- In order for these redirects to work correctly, we need the following 2 lines (just after setting up the login manager)

```
login_manager.login_view = 'login'
```

```
app.config['USE_SESSION_FOR_NEXT'] = True
```

- Without setting the `login_view`, attempting to access `@login_required` endpoints will result in an error
- It is more convenient to use sessions to store the `next` parameter, so that's why `'USE_SESSION_FOR_NEXT'` is set to `True` in the app configuration dictionary

File `protected.html`

- Showing only the contents of the `<body>` element

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Session Examples</title>
</head>
<body>
{% for message in get_flashed_messages() %}
{{ message }}<br/>
{% endfor %}

<h1>You have to be authenticated to view this</h1>
<h3>current_user.id: {{ current_user.id }}</h3>
<h3>current_user.is_authenticated: {{ current_user.is_authenticated }}</h3>
</body>
</html>
```

- After displaying the flashed messages, this page is showing how to refer to the current user
  - simply access the `current_user` object
  - and access the fields you need, such as `current_user.id`
- use `current_user.is_authenticated` to determine if the current user is authenticated of not
  - by default, `flask_login` uses a special anonymous user when no user is logged in
    - its `is_authenticated` field will always be false
  - when is a user is actually logged in, `is_authenticated` will be true
- The file `non_protected.html` is almost identical

Step 6: `/logout` endpoint

- When logging out, instead of clearing the session directly, it is better to call the `logout_user` function (also imported from `flask_login`), to let the login manager handle

it

- Note that this endpoint is also decorated with `@login_required`, since it does not make sense to log out if the user is not currently logged in

```
@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect('/')
```

File `index2.html`

- This template uses `current_user.is_authenticated` to display different things to logged-in users and anonymous users
- Note that this template is used by both this example and the next, so that's why it is checking if the current user has an email and a phone number
  - in this example, a user doesn't have these fields
  - but in the next, a user will have these fields
- Showing only the contents of the `<body>` element and skipping the display of flashed messages

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Session Examples</title>
</head>
<body>
{% for message in get_flashed_messages() %}
{{ message }}<br/>
{% endfor %}
{% if current_user.is_authenticated %}
<h1>Welcome {{ current_user.id }}!</h1>
{% if current_user.email and current_user.phone %}
<dl>
    <dt>email</dt>
    <dd>{{ current_user.email }}</dd>
    <dt>phone</dt>
    <dd>{{ current_user.phone }}</dd>
</dl>
{% endif %}
<a href="/logout">Logout</a>
{% else %}
<h1>Welcome Stranger!</h1>
```

```
<a href="/login">Login</a>
<a href="/register">Register</a>
{% endif %}
<a href="/non_protected">Non Protected</a>
<a href="/protected">Protected</a>
</body>
</html>
```

Better Password Handling and User Registration

- File `05_app.py`
- In this example, we improve on the previous example by having a more complete `User` class, and with a much better way to save and retrieve users data, including a properly hashed password, from a CSV file
- There is also a new registration page

`User` class

- 3 new fields are added to the `User` class: `email`, `phone` and `password`

```
class User(UserMixin):
    def __init__(self, username, email, phone, password=None):
        self.id = username
        self.email = email
        self.phone = phone
        self.password = password
```

`load_user` function

- The `load_user` function calls the `find_user` to search for a user with the given user id (actually the username in our example)
- The password will be included in the user object returned by the `find_user` function
  - but the login manager doesn't need it
  - so we hide it by setting it to `None` to avoid some potential security issues
- If `find_user` doesn't find the user, it will return `None`
  - therefore `load_user` might return `None`

```
@login_manager.user_loader
def load_user(user_id):
    user = find_user(user_id)
    # user could be None
    if user:
```

```
        # if not None, hide the password by setting it to None
        user.password = None
    return user
```

`find_user` function

```
def find_user(username):
    with open('data/users.csv') as f:
        for user in csv.reader(f):
            if username == user[0]:
                return User(*user)
    return None
```

- The first part of this function is similar to the `check_password` from the previous example
  - it reads a CSV file line-by-line, each line representing a user
  - `user[0]` is still the username
  - the following fields are, in order, the email address, the phone number, and the password
  - then the function returns a new user object
  - if the username is not found, it returns `None`
- The password is saved in a special form called a *hashed password*
  - the `bcrypt` package is used in this example
  - more details on this later on
- The special notation `*user` is often called *unpacking an argument list*
  - `user` is a list containing the *username*, *email* address, *phone* number and *password* of the user
  - the `User` constructor's arguments have to be given in the same order
  - so `*user` will *unpack* the list elements into the constructor arguments
  - we could write instead `return User(user[0], user[1], user[2], user[3])`
    - but `return User(*user)` is much cleaner

List unpacking example

```
def f(a, b, c):
    print("a = {}\nb = {}\nc = {}\n".format(a, b, c))


data = [2, "Hello", 5.6]
f(*data)
```

```
a = 2
b = Hello
c = 5.6
```

`/login` endpoint

- The same logic is kept in the `/login` endpoint, except that the password is not checked in the same way
- We use the `bcrypt` package to verify the hashed password

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = find_user(form.username.data)
        # user could be None
        # passwords are kept in hashed form, using the bcrypt algorithm
        if user and bcrypt.checkpw(form.password.data.encode(),
                                   user.password.encode()):
            login_user(user)
            flash('Logged in successfully.')
            next_page = session.get('next', '/')
            session['next'] = '/'
            return redirect(next_page)
        else:
            flash('Incorrect username/password!')
    return render_template('login.html', form=form)
```

- First, to check the password, we need to find the user (same as in `load_user`)
- Second, we check if we have a user and if yes, we use `bcrypt` to check the password
  - if `user` is `None`, then it will evaluate to `False` and the whole condition will be `False`
  - if we have a user, then we will check if the submitted password matches the stored password
    - in order for the `checkpw` to work correctly, the password strings have to be encoded properly
  - if we have a match, then we proceed in the same way as before

User Registration

- To register a new user, we need a `RegisterForm` and a corresponding `/register` endpoint
  - note that in this example, the forms have been moved to another file `forms.py`

- This form is reusing some fields and validators already used in other examples
  - the *username, email, phone, password, submit* fields and the `validate_password` function are identical to previous examples
- The only new field is the `password2` field
  - this field is used to confirm that the password was typed correctly
  - besides `InputRequired`, we use the `EqualTo` validator
    - `EqualTo('password', message='Passwords must match.')`
    - it specifies that it must be equal to the `password` field

```
class RegisterForm(FlaskForm):
    username = StringField('Username',
                           validators=[InputRequired(),
                                       Length(4, 64),
                                       Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
                                              'Usernames must start with a
        letter and must have only letters, numbers, dots or underscores')])
    email = EmailField('Email', validators=[InputRequired(), Email()])
    phone = StringField('Phone number', validators=[InputRequired()])
    password = PasswordField('Password', validators=[InputRequired(),
                                                     Length(8)])
    password2 = PasswordField('Repeat password',
                              validators=[InputRequired(),
                                          EqualTo('password',
                                                  message='Passwords must
        match.')])
    submit = SubmitField('Login')

    def validate_password(self, field):
        with open('data/common_passwords.txt') as f:
            for line in f.readlines():
                if field.data == line.strip():
                    raise ValidationError('Your password is too common.')
```

`/register` endpoint

- The `/register` endpoint is quite similar to the `/login` endpoint in many ways
  - they both render and handle forms
  - they both look for a user with a given username
    - the difference is that in `/login`, we need to find a given user to be successful
    - while in `/register`, we do *not* want to find a given user since we don't want 2 different users with the same username
  - in `/login` we need to verify the submitted password,

        - while in `/register` we need to hash the submitted password
    - in `/register`, instead of reading a CSV files, we need to append some data to the CSV file

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        # check first if user already exists
        user = find_user(form.username.data)
        if not user:
            salt = bcrypt.gensalt()
            password = bcrypt.hashpw(form.password.data.encode(),
                                     salt)
            with open('data/users.csv', 'a') as f:
                writer = csv.writer(f)
                writer.writerow([form.username.data,
                                 form.email.data,
                                 form.phone.data,
                                 password.decode()])
            flash('Registered successfully.')
            return redirect('/login')
    else:
        flash('This username already exists, choose another one')
        return render_template('register.html', form=form)
```

- If we do *not* find a user with the submitted username, `user` will be `None` and `not None` will evaluate to `True`
    - to hash a password with `bcrypt`, we need to generate a random *salt*, then we use this random salt to hash the password
    - the password needs to be encoded before being hashed
    - we then open the `users.csv` file in *append* mode to add a user row at the end of the file using a CSV writer
    - the fields need to be put in this order: *username, email, phone* and *password*
    - the password has to be decoded before saving it to the file
    - after writing the data to the file, the user is redirected to the login page

Converting the project to use Flask-SQLAlchemy

The MVC pattern in a Flask Project

- **MVC** pattern: **Model-View-Controller**

- In Flask,
  - the *templates* are the *views*
  - the *routes* (more precisely, the functions decorated with the `@app.route` decorator) are the *controllers*
  - the *models* can be implemented using *SQLAlchemy*
- The `Flask-SQLAlchemy` package is a wrapper around the `SQLAlchemy` package
- Check the documentation for many more details
- The models are usually placed in a separate file from the `app.py` containing the call to `app.run()`, usually named `models.py` when the number of models is small, or in many files if there are many models

Changes to the project

In `app.py`: 1. Import `SQLAlchemy` from `flask_sqlalchemy` 2. Configure the database URI in `app.config['SQLALCHEMY_DATABASE_URI']` 3. Create an instance of `SQLAlchemy` 4. Import the model 5. Modify the places where the DB was accessed 1. in the `find_user` function 2. in the `register` route - note that it might be a good idea to define a `register_user` function, and call it from the `register` route

In `models.py`: 1. Import the instance of `SQLAlchemy` from the app 2. Define a subclass of `db.Model` for the user 1. define fields of type `db.Column` for each column 2. define the `__repr__` function (similar to the `toString` method in Java) 3. (optional) define the table name

`find_user` function

```
def find_user(username):
    res =
        db.session.execute(db.select(DBUser).filter_by(username=username)).first(
    if res:
        user = SessionUser(res[0].username, res[0].email, res[0].phone,
        res[0].password)
    else:
        user = None
    return user
```

Instead of connecting directly to the DB, and executing directly an SQL statement on the connection, we go through the DB session, and call the `execute` method. We end up executing a `SELECT` statement anyway, but through a `db.select`. The `filter_by` is the equivalent of the `WHERE` condition. The main advantage of doing it this way is that data passed to the query,

coming from the string typed into a text field inside a form, will be escaped automatically for us, to help us avoid SQL injection attacks.

Another advantage is that we don't have to worry about which DBMS we connect to exactly. The differences between the DBMSs will be handled transparently by SQLAlchemy for us. It makes it much easier to migrate to another DBMS if we need to.

`register` route

Similar changes need to be made in the `register` route, except that we don't query the DB, but we insert, or add, a new user if a user with that username doesn't already exist. Instead of connecting directly to the DB and executing an `INSERT` statement, we create a new instance of `DBUser`, then we add it to the DB through the `db.session.add` method. We must not forget to call `commit` on the session to make sure that our new user is saved correctly in the DB.

```
    user = DBUser(username=form.username.data, email=form.email.data,
phone=form.phone.data,
                  password=password.decode())
    db.session.add(user)
    db.session.commit()
```

`manage.py` file

After creating a new model, if the corresponding table in the DB doesn't already exist, we need to create it. We could write a `CREATE TABLE` statement to create, but it would be easy to make a mistake and end up with a table that doesn't exactly match the model, because of non-matching names or data types for example. The easiest way is to tell SQLAlchemy to automatically create the tables corresponding to our models. It will only create the missing tables, so it is easy to add a new model and create its corresponding table without messing up the existing models/tables.

The `manage.py` file includes a function that will be called automatically if we start a *flask shell*. Open a terminal window in Pycharm, then execute the command `flask shell`. From there, you will be able to call `db.create_all()` to create the tables for the new models without touching the existing tables. Don't forget to commit your changes after creating the tables.

You could also create new `DBUser` instances here, and add them to the DB, as shown in the `register` route.

```
from app import app
from models import db, DBUser


@app.shell_context_processor
def make_shell_context():
    return dict(app=app, db=db, DBUser=DBUser)
```