

## Flask

### Part 2: Deployment on OpenShift. Posts. Forms.

#### Lab Objectives

Objectives of this lab are to carry on developing our **Blogging site**, by:

- adding individual pages for blog posts, which are accessed from the home page;
- implementing 'User Account' functionality using Flask forms.
- deploying the website on OpenShift (**NB:** VPN connection is required for OpenShift);

**NB:** It is advisable to do all the work with your virtual environment **activated**.

#### General Comments

As before, these exercises are not assessed. If you do not manage to finish all the tasks, please attempt to finish them in your own time.

Use the suggested resources and links, provided throughout this document and on the last page, to help you understand the code. A snapshot of the state of the project at the end of this lab is available on Learning Central. You can download this to help you if you are stuck, to check your progress, and, perhaps, at times to save you typing *\*from scratch\**. However, please make sure you understand each line of the code!

It is okay to discuss the solutions with your peers (these exercises are not assessed!), however, make sure you understand everything by yourself.

**IMPORTANT!!** These exercises will give you some basic understanding of how to develop a website in Flask. To get a more comprehensive understanding of how this 'stuff' works, it is strongly advised that you read the suggested documentation, 'quickstarts', recommended book, as well as complete a few tutorials. However, these are just suggestions and the list is non-exhaustive! There are lots of other resources and tutorials on the Web.

#### Abbreviations used in this document

- **dir** – directory (folder)
- **venv** – virtual environment.
- **db** – database
- **NB** – Nota bene

## INDIVIDUAL POST PAGES

In our online Blog, each individual post is accessed by using ‘dynamic’ URLs in the form of `post/<post_id>`, e.g. for the first post in our database with the URL is `post/1`. To enable our website visitors to access each post’s page, we need to:

- create a new `post.html` template,
- and then update `home.html` as follows:

1. In `blog/templates` dir, create an empty `post.html`, make sure it inherits all the elements of our site’s layout (i.e. navigation, etc.), i.e.:
2. In `{% block content %}` section of the page, specify that we want the page to display each post’s image, content, titles and author (similar to what we did previously in `home.html` page).

```
{% extends "layout.html" %}
{% block content %}

<p>"{{ post.title }}" &nbsp; Author: {{ post.user.username }}</p>
<p>{{ post.content }}</p>
{% endblock content %}
```

3. In `home.html`, update the template in such a way that when the user clicks on a post’s title that post’s individual page is displayed. This is accomplished by using a `href`, e.g.:

```
...
<a href="{{ url_for('post', post_id=post.id) }}">"{{ post.title }}"</a>
...
```

4. Now, we need to update `routes.py` to tell the server where to redirect to `post.html` when the user clicks on the post’s title:

```
...
@app.route("/post/<int:post_id>")
def post(post_id):
    post = Post.query.get_or_404(post_id)
    return render_template('post.html', title=post.title, post=post)
```

5. Find an image, name it ‘`default.jpg`’ and put it in the directory `./flask-blog/blog/static/img/`, i.e.:

```
./flask-blog
|
+---blog
|
+---static
|
\---img
      default.jpg
```

6. Test it works by clicking on a post’s title to check that it redirects to that post’s page.
7. Update `home.html` to also enable the user to click on a post’s image to redirect to that post’s page.

## USER ACCOUNTS

We want our visitor to create their user account, so that they can register, log in and log out. Later on, we will also be using the user accounts for ‘posting a comment’. Implementation of the user accounts is accomplished by using **Flask-Login**<sup>1</sup>, which will be responsible for handling user authentication. We also need **Flask-WTF**<sup>2</sup>, which is an integration of Flask and **WTForms**<sup>3</sup> - for creating forms.

**NB:** Before you start working on the tasks in this section, you need to check if you have these packages installed, and if not, install it using **pip** in your **venv**.<sup>4</sup>

**NB:** Most of the implementation in this section is based on **Flask-Login**<sup>5</sup> documentation and Chapter 8 ‘User Authentication’ of M. Grinberg’s book “Flask web development”, O’Reilly Media, 2014.

### Initial Modifications

8. To start with, we need to initiate **Flask-Login** in our app:
  - (a) Open `__init__.py`. Add an import for **LoginManager** and initialise its object:

```
...
from flask_login import LoginManager
...
login_manager = LoginManager()
login_manager.init_app(app)
...
```

### DB Update

9. We will **store the user’s login credentials** in our db. To enable this, we need to modify an existing table **User** for the db, by: (a) modifying `models.py`, and (b) writing the changes to the db:


```
(a) ...
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(15), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(128))
    password = db.Column(db.String(60), nullable=False)
    post = db.relationship('Post', backref='user', lazy=True)

    def __repr__(self):
        return f"User('{self.username}', '{self.email}')"

    @property
    def password(self):
```

---

<sup>1</sup><https://flask-login.readthedocs.io/en/latest/> 

<sup>2</sup><https://flask-wtf.readthedocs.io/en/stable/> 

<sup>3</sup><https://wtforms.readthedocs.io/en/2.3.x/> 

<sup>4</sup>**NB:** If you are getting an error message when you use **pip** to install a python package, you might need to use `--user` option, i.e. `pip install --user <PACKAGE>`.

<sup>5</sup><https://flask-login.readthedocs.io/en/latest/>

```

        raise AttributeError('password is not a readable attribute')

    @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)

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

```

The above code requires three *imports*:

```

...
from blog import login_manager
from werkzeug.security import generate_password_hash, check_password_hash
from flask_login import UserMixin
...

```

- (b) Back up the tables `post` and `user` first - in case you need to revert. Then, drop the two tables. SQL commands to drop the tables:

```

mysql> DROP TABLE post;
mysql> DROP TABLE user;

```

Update the db, using the python shell (see the instruction for the previous lab, i.e. "Flask 1: Essentials"/ "DATABASE (DB)" section).<sup>6</sup>

Python shell commands to update the database:

```

> python
>>> from blog import db
>>> db.create_all()

```

Confirm that you now have the `User` table updated.

The table `user` should now look as follows:

```

mysql> describe user;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id             | int(11)       | NO   | PRI | NULL    | auto_increment |
| username       | varchar(15)   | NO   | UNI | NULL    |                |
| email          | varchar(120)  | NO   | UNI | NULL    |                |
| password_hash  | varchar(128)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+

```

---

<sup>6</sup>Alternatively, you might want to use any other suitable method

## User Registration

To register a new user, we need to create a **user registration** form. It will have the following fields: `username`, `email`, `password`, `confirm_password`, and `submit` button.

10. Create a new file `forms.py` in the `blog` dir.

(a) Start with importing `FlaskForm` from `flask_wtf`:

```
from flask_wtf import FlaskForm
```

(b) Use class `RegistrationForm(FlaskForm)` to declare the first field, `username`, as follows:

```
...
class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(),
    ↪ Length(min=3, max=15)])
```

**NB:** this statement requires several *imports*, such as: `StringField` from `wtforms`, and `DataRequired` and `Length` from `wtforms.validators`, i.e.:

```
from wtforms import StringField
from wtforms.validators import DataRequired, Length
```

(c) Similarly, create the definitions of the fields for: `email`, `password`, `confirm_password`, and `submit` button in the class `RegistrationForm(FlaskForm)`:

```
...
email = StringField('Email', validators=[DataRequired(), Email()])
password = PasswordField('Password', validators=[DataRequired()])
confirm_password = PasswordField('Confirm Password',
    ↪ validators=[DataRequired(), EqualTo('password')])
submit = SubmitField('Register')
```

(d) add the necessary imports from `wtforms` and `wtforms.validators`, i.e. `PasswordField`, etc.:

```
...
from wtforms import PasswordField, SubmitField
from wtforms.validators import Email, EqualTo, ValidationError,
    ↪ Regexp
...
```


11. The next step is to add the form to the registration page - `register.html`.

(a) Create `register.html` in `blog/templates` dir.

(b) In `{% block content %}` section, specify that we want to add the form and its field `username` <sup>7</sup>:

```
...
<form method="POST" action="">
    {{ form.csrf_token }}
    {{ form.username.label }} {{ form.username }}
    <input type="submit" value="Register">
</form>
...
```

---

<sup>7</sup>If you are curious about what "form.csrf\_token" is about, see here: <https://wtforms.readthedocs.io/en/2.3.x/csrf/> 

(c) Using the same principle, add all other form fields (`email`, etc.).

12. To process the form, we need to modify `routes.py` to tell the server how to handle the form:

(a) Import `RegistrationForm` class from `forms.py`:

```
...
from blog import db
from flask import request, redirect
from blog.forms import RegistrationForm
...
```

(b) Add the `@app.route` decorator for `register`:

```
...
@app.route("/register", methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if request.method == 'POST':
        user = User(username=form.username.data,
                    ↪ email=form.email.data, password=form.password.data)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('home'))
    return render_template('register.html', title='Register',
                    ↪ form=form)
...
```

13. Test the registration process works as intended, by going to `http://127.0.0.1:5000/register` and create a few ‘users’. Check these users’ details are now listed in your db, in `user` table.

```
mysql> select * from user;
```

```
+----+-----+-----+-----+
| id | username | email           | password_hash |
+----+-----+-----+-----+
| 1  | johnsmith | john@smith.com | pbkdf2:sha256:50000$tlv0TvWz$c57... |
+----+-----+-----+-----+
```

**NB:** For the moment, we are assuming the user only provides valid input. Handling user input’s validity and errors is the subject of the next lab.

14. Modify the model by adding additional attributes, such as the user’s first and last name, and then update your db. Add these fields to the form and display them on the `register` page.
15. Currently, when a user registers successfully, they are taken to the home page. Change the redirection, so that after they have registered, instead, they are taken to a ‘*Thank you for registering*’ page.

## User Login

**User Login** functionality is implemented, using similar principles to those followed when implementing **User Registration**, i.e. you need to create `LoginForm` class in `forms.py`, create `login.html` template, and add logic of how to handle login to `routes.py`.

16. Update `forms.py`, by creating `LoginForm`:

```
class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(),
    ↪ Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Login')
```

17. Create a new `login.html`, and add the form to display the user's email and password fields (similar to 11).

18. Modify `routes.py`:

- (a) Add `@app.route("/login")`:

```
...
@app.route("/login", methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if request.method == 'POST':
        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)
            return redirect(url_for('home'))
    return render_template('login.html', title='Login', form=form)
...
```

- (b) Add all the necessary *imports*, including import of `LoginForm` to `routes.py`.

```
from blog.forms import LoginForm
from flask_login import login_user
```

19. Test the login process works as intended, by going to `http://127.0.0.1:5000/login` and check you can log in as one of the 'user' you added to the database in Task 13. On successful login the website will return to `home` page.

**NB:** As before, for the moment, we are assuming the user only provides valid input. Handling user input validity and errors is covered in the next lab.

## User Logout

20. You might be pleased to know that '**user logout**' functionality is, perhaps, the easiest of all to implement! We don't need to have a special form for this, just modification of `routes.py`, to which we need to add the `@app.route("/logout")` decorator:

```
...
@app.route("/logout")
def logout():
```

```
logout_user()
return redirect(url_for('home'))
```

**NB:** make sure you to add all the necessary import(s) to ‘from flask\_login import logout\_user’ statement at the top of the file.

21. Similar to our implementation of `login` functionality, on successful logout we are returning the user to the home page. To test whether the logout functionality works, we can: either create a ‘successful logout’ page and modify `routes.py` to redirect to it, or modify the website navigation to include the menu items for registration, logging in and logging out, and then deploy some logic so that the home page will reflect the current state of the user – see the next section.

## NAVIGATION ENHANCEMENT

22. Modify `layout.html` to enhance the website navigation by providing your website visitors with the links to `register`, `login` and `logout`. The website should display the links appropriate to each user, e.g. a ‘*guest*’ user should see `register` and `login`, while the logged-in user should see `logout`. You could also add a `Hello, <USER NAME>` greeting to the navigation bar to personalise the website for the logged-in user. Whilst, the default greeting would be `Hello, Guest!`.

*Hint:* `current_user` proxy allows you to access the logged-in user - see: <https://flask-login.readthedocs.io/en/latest/><sup>8</sup>. Also, check out the examples provided here: <https://flask.palletsprojects.com/en/1.1.x/tutorial/templates/><sup>9</sup>.

## DEPLOYMENT on OpenShift

To deploy our Flask website we will be using our School’s OpenShift server (<https://openshift.cs.cf.ac.uk/>)<sup>8</sup>.

23. Make sure you have followed the instructions for both labs, ‘Flask 1’ and this one, so that you already have all the project files in the appropriate directories.
24. Check you have `requirements.txt` file in the root dir of the project; and all the necessary libraries are listed in this file. The file is available in the snapshot of the current state of the project - see ‘Week 10 - Practicals - SUPPORTING FILES’ folder on Learning Central.
25. We will be using Gunicorn<sup>9</sup> to deploy our flask website on OpenShift, so you need to make sure a new line with the word `gunicorn` is added to the `requirements.txt`.
26. Commit changes locally and push to the remote repo on GitLab.
27. Go to GitLab, and check that all the files from the local directory have been successfully pushed.

---

<sup>8</sup>The video demonstrating deployment of our website both on localhost and OpenShift is available on Learning Central.

<sup>9</sup>A Python Web Server Gateway Interface HTTP server - <https://gunicorn.org>



28. Go to our OpenShift server, <https://openshift.cs.cf.ac.uk/><sup>10</sup>, and log in with your University network credentials.
29. **Create a project**, as follows:
  - (a) Click on **Create Project**.
  - (b) Fill in the form, e.g. *Name*: my-blog, *Display Name*: Blogging Website, and anything you wish to use to describe your project.
  - (c) Click on **Create** button.
  - (d) Click on the project name you have just created.
30. **Adding a Python 'Cartridge'**:
  - (a) Click on **Browse Catalog**.
  - (b) Search catalog for **Python**;
  - (c) On the pop-up dialog, click **Next**;
  - (d) On **Configuration** page:
    - Click on the project name you have just created;
    - for **Application Name** type the name of the project you have just created (i.e. 'my-blog');
    - and for **Git Repository** - the URL of your git repository. You can find this in and copy from GitLab (E.g. go to the repository's 'home' page -> click on Clone dropdown button -> **Clone with SSH**.)
    - Click on **Create** and **Close**.

### 31. **Project Build:**

Go back to your project, and then **Builds/Builds**. You will see that fetching of the source fails. If you inspect the **Build's log**, you will see the reason why this has happened, i.e. you have not yet given OpenShift permission to connect to your repo. This is done by creating a **Secret**.

### 32. **Creating a Secret:**

- (a) To create a Secret:
  - Go to **Resources** -> **Secrets** -> **Create Secret** button;
  - Select **Source Secret** for **Secret Type**; type \***Secret Name** (e.g. my-blog-secret);
  - Select **SSH Key** for **Authentication Type**;
  - and then either select the file with your SSH private key, or paste your private key into the text box;
  - Click on **Create**.
  - Whilst still on **Resources** -> **Secrets** page, select the secret you have just created, click **Add to Application**, then select your application from the dropdown menu, and then **Save**.

---

<sup>10</sup>If you are on a Mac, Safari might not work with our OpenShift. Use another browser, e.g. Chrome or Firefox.

(b) The `secret` also needs to be added to the builds:

- Builds -> Builds;
- then select your application;
- then **Actions** button -> **Edit** -> under **Git Repository URL** select 'advanced options' link (in blue) and from -> **Source Secret** -> dropdown menu select the secret you have just created, scroll to the bottom of the page and click on -> **Save**.

### 33. Project Re-Build and Deployment:






- (a) Go back to **Builds**, and click on **Start Build**. The second build should now succeed.
- (b) After the build has completed successfully, OpenShift will deploy your application (this might take some time), and you can then access your website from URL specified on the **Overview** page. The URL will be something like: `http://<PROJECTNAME>.apps.cs.cf.ac.uk`
- (c) **NB**: Deployment might not start automatically. To start it manually: go to **Applications** -> **Deployments** -> select your project -> click on **Deploy** button.



**If you modify your code locally, you need to push your modified code to GitLab, and then re-build and re-deploy your website on OpenShift.**



**It is strongly advised to work on your code locally and not make any code modifications directly on GitLab, otherwise you might end up with having to deal with rectifying conflicts.**

Useful resources	
Flask Website:	<a href="https://flask.palletsprojects.com/en/1.1.x/">https://flask.palletsprojects.com/en/1.1.x/</a> 
Flask Tutorial:	<a href="https://flask.palletsprojects.com/en/1.1.x/tutorial/">https://flask.palletsprojects.com/en/1.1.x/tutorial/</a> 
Flask-WTF home page:	<a href="https://flask-wtf.readthedocs.io/en/stable/">https://flask-wtf.readthedocs.io/en/stable/</a> 
Flask-WTF Quickstart:	<a href="https://flask-wtf.readthedocs.io/en/stable/quickstart.html">https://flask-wtf.readthedocs.io/en/stable/quickstart.html</a> 
Book on Flask:	M. Grinberg's (2018). "Flask web development : developing web applications with Python", O'Reilly.
M. Grinberg's tutorial on 'User Logins' ( <i>more or less the same as in the book above</i> ):	<a href="https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-v-user-logins">https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-v-user-logins</a> 
Plenty of video tutorials on YouTube ( <i>in no particular order!</i> ), e.g.:	<a href="#">[1]</a> , <a href="#">[2]</a> (these links are to the first lessons in a series; other episodes cover the flask forms).
A number of cheat sheets could be found on the web.	