

Coursework Report

Viktor Kanev
40282142@napier.ac.uk
Edinburgh Napier University - Web Technologies (SET09103)

1 Title - newsroom

2 Introduction

The aim of this coursework was to create a well-rounded web-app which offers an excellent level of functionality, both in terms of the number of features and their quality of implementation. For this reason I have chosen to do a blog-style app which can be designed to suit any particular topic. For this assignment I have chosen to be 'news' focused, hence the title of the web-app - 'newsroom'. The app utilizes routing, requests, redirects, responses, custom error code handling, static files, templates, sessions, logging, data storage, encryption and sending emails to reset password. The core features of the app are: adding new post, editing and deleting it, adding comments under the posts, there is a full-text search via which you can search through the posts providing a keyword, there is also a profile page where you can edit your username, email and profile picture. You can login/register and if you forgot your password there is an option to reset it by sending an reset-password email to the email linked to the account. The app can deal with multiple users, posts, comments, static files which is achieved by storing the data to a database file.

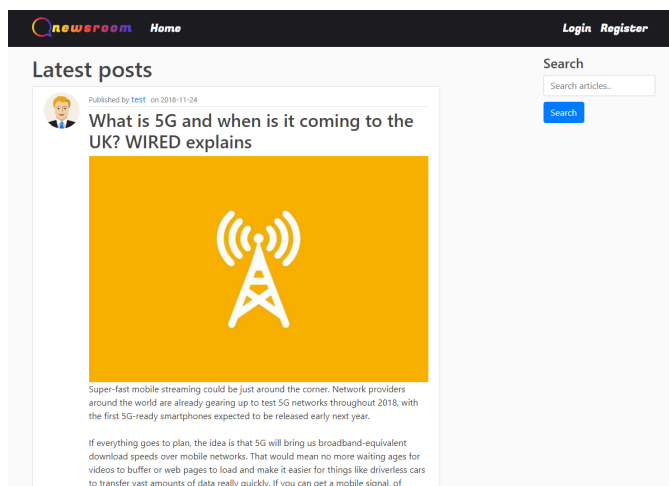


Figure 1: Main page

On the main page all the posts from the database are rendered in a single container per post which includes the author of the article including their profile picture, the title of the article and when it was posted. From there you can either login or register if you don't already have an account.

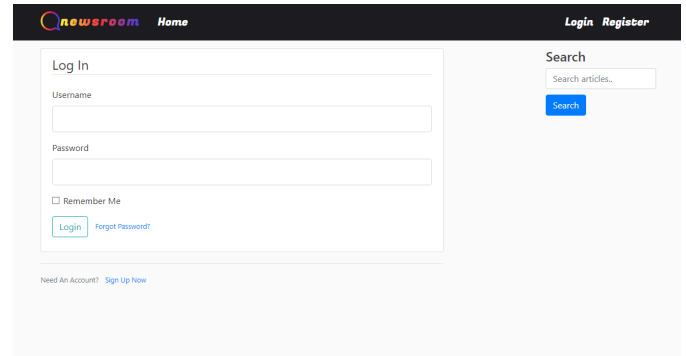


Figure 2: Login page

The login page is pretty straight-forward there are two fields for username and password, also if you do not have an account there is suggestion of registering one. Also there is a link for resetting your password which takes you to a separate page.

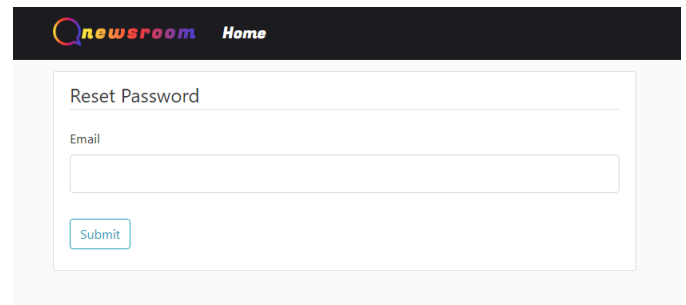


Figure 3: Reset password page

After you submit the form and follow the link that was sent to your email you can reset it.

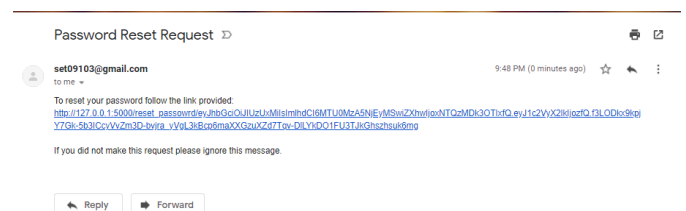
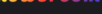


Figure 4: Email with reset link

 [Home](#)

Reset Password

Password


Confirm Password

[Reset Password](#)

Figure 5: **Reset page**


2.1 Logged in

Once you are logged in the Login and Register routes are hidden and also if you try to access their url you are redirected to the homepage. They are replaced by Profile, Add post and logout routes.

 Home

Profile Add post Logout

Latest posts

 Published by [test](#) on 2018-11-24


What is 5G and when is it coming to the UK? WIRED explains

Search


Search

Figure 6: **Logged in page**

On the profile page you can edit your username, email and profile picture which is assigned by default to a picture from the static files.


[Home](#)

[Profile](#)
[Add post](#)
[Logout](#)



test

test@test.com

Profile Info

Username

Email

Update profile picture

No file chosen

Search


Figure 7: **Profile page**

You can comment under the posts only if you are logged in. Otherwise the add comment form is not shown. You are only able to edit your own posts and the routes for the Update and Delete button are protected and can be only accessed by the author of the article.

Directive on Copyright.

Add comment

Comments:

 **test** 2018-11-24 @21:12

Nice!

Figure 8: **Commenting**

2.2 Add post

You can add post either with picture or just plain text. The picture is stored in the static folder.

2.3 Search

The input inserted into the search box will render on submit all the posts that contain the input in the title or the body of the article.

3 Design

3.1 Hierarchy

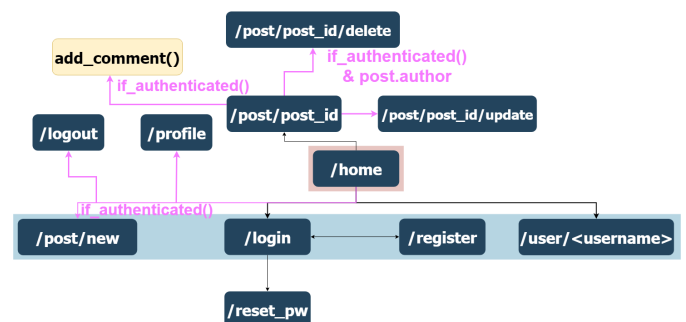


Figure 9: **URL Hierarchy**

3.2 Navigation

This is the URL hierarchy for the website. The website uses a bootstrap template with an additional **custom.css** file, which customizes the page to my preference. There is a top navigation with custom made logo. The use of the bootstrap template allows for the website to be responsive and viewed on smaller screens by collapsing the navigation.

3.3 Structure

On the main page you can see all the articles, which is the main goal of the web-app. All the data for the User, Post and Comments is stored to a single database file. There is a model for each of them and they all have relationship with each other so the posts and comments can be attached to

the author. Pictures for the avatars of the users are re-sized in order to save space on the server, they are also renamed to a random hex so that if two users upload image with the same name the image files won't collide. The same function is used for the pictures for the posts, however the images here are not re-sized. They are dynamically re-sized via the use of the **img-fluid** class in bootstrap so they can be properly displayed on smaller screens.

3.4 Security

The passwords for the users are also stored in the database, however before they are saved they are hashed using the **bcrypt** module in python.

3.5 Forms

For the forms of the web-app I am using **flask_wtf** which is a flask module which allows me easily to create forms and add custom validators, so that it is impossible for the users to enter wrong input. And if the users try to do so a **ValidationError** is raised, showing them what they have input wrong. For example when registering there are two fields for password and if they don't match, the users are pointed to why they are not being registered.

3.6 Custom error handling and flashed messages

If a user tries to go to a url which does not exist a custom error page is being showed. There are custom pages for codes 404 - not found, 403 - forbidden(if they try to access a route that requires authentication) and 500 - server error.

I am making use of flashed messages whenever a user is registered, password is changed, article is added/updated/deleted, comments is added, account is updated and there is also a **#deleteModal** on the delete button and a message is popped up whenever a user tries to delete a post asking them if they really want to do so, so that they don't delete a post by misclicking on the Delete button.

4 Enhancements

To start with I could make different level of access for the users, for example I can have moderators who can edit and delete posts(if a post is inappropriate for example). Another must-do is the ability for the users to edit or delete their comments under the posts. In order for the post to be more aesthetically pleasing would be implementing an editor to the **Add Post** form so that the authors can stylize the text. Another feature that is a must for a blog-like forum is the ability to send and receive messages, thus allowing more interaction between users, not only commenting under the posts, making it more personal.

5 Critical evaluation

All the features that are included in the web-app work quite well in my opinion. However for the time being the username and the email used to send emails to the users are currently

hard coded, this is due to the fact that the web-app will be run on multiple machines, otherwise a good idea would be to store them in environment variables.

Some of the lines of code that are written will not run on older versions of python, however they can be easily refractored so the app runs on older versions of python as well.

Improvement can be made on the add post form so the user can stylize the text.

I think there might be a problem with the search, which is currently filtering through all the posts in the database, and if the posts become thousands I am not sure how well the search will perform, it might become unusable.

6 Personal evaluation

I think I have performed better than my first assignment. The most difficult part was trying to display the correct information from the database. I have learned a lot about python and flask and the numerous possibilities that it provides. I have also learned a lot about databases and how to work with them after watching numerous tutorials and reading a lot of articles. Using bootstrap set a good foundation that I was able to work with and making it more personal and aesthetically pleasing by using my own custom.css file. The knowledge I have gained on the previous coursework was greatly improved by doing this one and overcoming the challenges that came with it.

7 Appendix

To use any of the external modules simply install them using the python environment(`pip install 'module name'`). For example I have use several external modules that are not in the learning environment.

7.1 SQLAlchemy

To install use:

pip install flask-sqlalchemy

Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application. It requires SQLAlchemy 0.8 or higher. The reason I chose to use it is because it aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks. I have chosen to use that instead of making SQL queries as they can be quite messy if they get too long and the risk of typos increase.

For the common case of having one Flask application all you have to do is to create your Flask application, load the configuration of choice and then create the SQLAlchemy object by passing it the application.

Once created, that object then contains all the functions and helpers from both `sqlalchemy` and `sqlalchemy.orm`. Furthermore it provides a class called `Model` that is a declarative base which can be used to declare models:

Listing 1: SQLAlchemy

```
1 from flask import Flask
2 from flask.sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp←
  /test.db'
6 db = SQLAlchemy(app)
7
8
9 class User(db.Model):
10     id = db.Column(db.Integer, primary_key=True)
11     username = db.Column(db.String(80), unique=True, nullable←
  =False)
12     email = db.Column(db.String(120), unique=True, nullable=←
  False)
13
14     def __repr__(self):
15         return '<User %r>' % self.username
```

To create the initial database, just import the db object from an interactive Python shell and run the `SQLAlchemy.create_all()` method to create the tables and database:

Listing 2: SQLAlchemy

```
1 >>> from yourapplication import db
2 >>> db.create_all()
```

Creating users:

Listing 3: SQLAlchemy

```
1 >>> from yourapplication import User
2 >>> admin = User(username='admin', email='admin@example.←
  com')
3 >>> guest = User(username='guest', email='guest@example.←
  com')
```

Adding data to the db:

Listing 4: SQLAlchemy

```
1 >>> db.session.add(admin)
2 >>> db.session.add(guest)
3 >>> db.session.commit()
```

Accessing the data:

Listing 5: SQLAlchemy

```
1 >>> User.query.all()
2 [<User u'admin'>, <User u'guest'>]
3 >>> User.query.filter_by(username='admin').first()
4 <User u'admin'>
```

7.2 Flask-WTF

To install use:

pip install flask-wtf

Forms provide the highest level API in WTForms. They contain your field definitions, delegate validation, take input, aggregate errors, and in general function as the glue holding everything together. I have chosen to use Flask-WTF forms as they have validators of the fields so I don't have to create my own validators, which save a lot of time.

Flask-WTF provides your Flask application integration with WTForms. For example:

Listing 6: Flask-WTF Form

```
1 from flask_wtf import FlaskForm
2 from wtforms import StringField
3 from wtforms.validators import DataRequired
4
5 class MyForm(FlaskForm):
6     name = StringField('name', validators=[DataRequired()])
```

In addition, a CSRF token hidden field is created automatically. You can render this in your template:

Listing 7: Flask-WTF Form

```
1 <form method="POST" action="/">
2     {{ form.csrf_token }}
3     {{ form.name.label }} {{ form.name(size=20) }}
4     <input type="submit" value="Go">
5 </form>
```

If your form has multiple hidden fields, you can render them in one block using `hidden_tag()`.

Listing 8: Flask-WTF Form `hidden_tag()`

```
1 <form method="POST" action="/">
2     {{ form.hidden_tag() }}
3     {{ form.name.label }} {{ form.name(size=20) }}
4     <input type="submit" value="Go">
5 </form>
```

Validating forms in your routes.py file:

Listing 9: Flask-WTF Validation

```
1 @app.route('/submit', methods=('GET', 'POST'))
2 def submit():
3     form = MyForm()
4     if form.validate_on_submit():
5         return redirect('/success')
6     return render_template('submit.html', form=form)
```

7.3 Flask-Login

To install use:

pip install flask-login

I have chosen to use flask-login as it handles the logins and saves you a lot of time, as well as protects your users' sessions from being stolen.

Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.

It will:

Store the active user's ID in the session, and let you log them in and out easily. Let you restrict views to logged-in (or logged-out) users. Handle the normally-tricky "remember me" functionality. Help protect your users' sessions from being stolen by cookie thieves. Possibly integrate with Flask-Principal or other authorization extensions later on.

The most important part of an application that uses Flask-Login is the LoginManager class. You should create one for your application somewhere in your code, like this:

Listing 10: Flask-Login

```
1 login_manager = LoginManager()
2 //Once the actual application object has been created, you can ←
  configure it for login
3 login_manager.init_app(app)
```

By default, Flask-Login uses sessions for authentication. This means you must set the secret key on your application, otherwise Flask will give you an error message telling you to do so.

How it works: You will need to provide a user_loader callback. This callback is used to reload the user object from the user ID stored in the session. It should take the unicode ID of a user, and return the corresponding user object. For example:

```
1 @login_manager.user_loader
2 def load_user(user_id):
3     return User.get(user_id)
```

Views that require your users to be logged in can be decorated with the login_required decorator:

```
1 @app.route("/logout")
2 @login_required
3 def logout():
4     logout_user()
5     return redirect(somewhere)
```

7.4 Flask-mail

To install use:

pip install flask-mail

One of the most basic functions in a web application is the ability to send emails to your users.

The Flask-Mail extension provides a simple interface to set up SMTP with your Flask application and to send messages from your views and scripts.

Listing 11: Flask-Mail

```
1 from flask import Flask
2 from flask_mail import Mail
3
4 app = Flask(__name__)
5 app.config['MAIL_SERVER'] = default 'localhost'
6 app.config['MAIL_PORT'] = default 25
7 app.config['MAIL_USE_TLS'] = default False
8 app.config['MAIL_USERNAME'] = 'default None'
9 app.config['MAIL_PASSWORD'] = 'default None'
10
11 mail = Mail(app)
```

Listing 12: Flask-Mail - Sending messages

```
1 from flask_mail import Message
2
3 @app.route("/")
4 def index():
5
6     msg = Message("Hello",
7                   sender="from@example.com",
8                   recipients=["to@example.com"])
```

7.5 Pillow

To install use:

pip install Pillow

Pillow is a fork of the Python Imaging Library (PIL). PIL is a library that offers several standard procedures for manipulating images. It's a powerful library, but hasn't been updated since 2011 and doesn't support Python 3. Pillow builds on this, adding more features and support for Python 3. It supports a range of image file formats such as PNG, JPEG, PPM, GIF, TIFF and BMP. We'll see how to perform various operations on images such as cropping, re-sizing, adding text to images, rotating, greyscaling, e.t.c using this library.

A crucial class in the Python Imaging Library is the Image class. It is defined in the Image module and provides a PIL image on which manipulation operations can be carried out. An instance of this class can be created in several ways: by loading images from a file, creating images from scratch or as a result of processing other images.

I have chosen to use Pillow for my users' avatars to resize them in order to save space.

Listing 13: Pillow - image re-sizing and saving to static

```
1 def save_picture(form_picture):
2     //generates random hex using secrets module, for older ←
   versions of python replace with os.urandom(8).hex()
3     random_hex = secrets.token_hex(8)
4     _, f_ext = os.path.splitext(form_picture.filename)
5     picture_fn = random_hex + f_ext
6     picture_path = os.path.join(app.root_path, 'static/avatars', ←
   picture_fn)
7
8     output_size = (125,125)
9     i = Image.open(form_picture)
10    i.thumbnail(output_size)
11    i.save(picture_path)
12
13    return picture_fn
```

7.6 itsdangerous

To install use:

pip install itsdangerous

You can serialize and sign a user ID for unsubscribing of newsletters into URLs. This way you don't need to generate one-time tokens and store them in the database. Same thing with any kind of activation link for accounts and similar things.

Signed objects can be stored in cookies or other untrusted sources which means you don't need to have sessions stored on the server, which reduces the number of necessary database queries.

Signed information can safely do a roundtrip between server and client in general which makes them useful for passing server-side state to a client and then back.

I have used this module to serialise the tokens used for sending the reset-password emails, so they can only be used by the user the email was sent to.

Because strings are hard to handle this module also provides a serialization interface similar to json/pickle and others. (Internally it uses simplejson by default, however this can be changed by subclassing.) The Serializer class implements that:

Listing 14: itsdangerous

```
1 from itsdangerous import TimedJSONWebSignatureSerializer as ↵
   Serializer
2
3 //in the User model adding functions
4 class User(db.Model, UserMixin):
5     id = db.Column(db.Integer, primary_key = True)
6     .
7     .
8     .
9
10    def get_reset_token(self, expire_sec=1800):
11        s = Serializer(app.config['SECRET_KEY'], expire_sec)
12        return s.dumps({'user_id': self.id}).decode('utf-8')
13
14    @staticmethod
15    def verify_reset_token(token):
16        s = Serializer(app.config['SECRET_KEY'])
17        try:
18            user_id = s.loads(token)['user_id']
19        except:
20            return None
21        return User.query.get(user_id)
```

8 References

Materials used: **workbook.pdf**

Flash messages: <http://flask.pocoo.org/docs/1.0/patterns/flashin>

Flask: <http://flask.pocoo.org/docs/1.0/>

SQLAlchemy: <http://flask-sqlalchemy.pocoo.org/2.3/> ,

<https://docs.sqlalchemy.org/en/latest/>

Flask-WTF: [https://flask-](https://flask-wtf.readthedocs.io/en/stable/)

[wtf.readthedocs.io/en/stable/](https://flask-wtf.readthedocs.io/en/stable/)

Flask-login: [https://flask-](https://flask-login.readthedocs.io/en/latest/)

[login.readthedocs.io/en/latest/](https://flask-login.readthedocs.io/en/latest/)

Flask-mail: <https://pythonhosted.org/Flask-Mail/>

YouTube series: <https://www.youtube.com/user/schafer5>,

[https://www.youtube.com/channel/UC-](https://www.youtube.com/channel/UC-QDfvrRiDB6F0bIO4IHkQ)

[QDfvrRiDB6F0bIO4IHkQ](https://www.youtube.com/channel/UC-QDfvrRiDB6F0bIO4IHkQ),

<https://www.youtube.com/user/CodingEntrepreneurs>

Flask-pillow: [https://auth0.com/blog/image-](https://auth0.com/blog/image-processing-in-python-with-pillow/)

[processing-in-python-with-pillow/](https://auth0.com/blog/image-processing-in-python-with-pillow/)

itsdangerous: <https://pythonhosted.org/itsdangerous/>