**Deploy:** [**https://sss-portfolio.herokuapp.com/**](https://sss-portfolio.herokuapp.com/)

**Agreed Requirements**

Agreed Requirements

For the agreed requirements, this report will list the requirements in two aspects which are

client requirements and sprint requirements, these requirements were discussed and agreed

during the project. We work through the project by meeting these requirements.

We have some client requirements, the first one is a view counter. We figured view count is

a good indicator of how successful the portfolio is designed, so the user can reflect based on

the frequency of visitors and do changes accordingly. We also think it is very cool if the

website allows visitors to comment and like, so there are some interactions between the two

parties. We have carefully assessed these requirements and have different priorities for

each of them, so we can fit them in our schedule throughout the semester. As of today, we

managed to implement a mostly functional view counter and the function to export the profile

page as a single pdf, so users may download their ePortfolio and send an offline copy or

print a hardcopy if they wish. Some other general things include communication. It is just to

make sure us and the client team are all on the same page throughout the development

process. The last one is the overall expectation from the clients, the main goal is to make

sure our final product is reliable, and easy to use for users creating the portfolio, and of

course, delivering the project on time.

We also have some agreed requirements in each sprint. For sprint 1, the goals are Trello

board setup, complete documentation, basic text upload portfolio, continue working on

database features, continue working on database features, ability to upload multimedia files:

pictures and videos etc., ability to categorize different types/fields of portfolio items, e.g.

music, art, code, videos, link Flask backend to SQLite database. For sprint 2, our agreed

requirements for the project are motivational Model, ability to upload multimedia files:

pictures and videos etc., ability to categorize different types/fields of portfolio items, e.g.

music, art, code, videos, link Flask backend to SQLite database, profile and landing page -

CSS to look like the Figma diagrams, decide on a handover time of the fully completed

product to clients, continue working on database features and deploy on Heroku and

Firebase. For sprint 3, our agreed requirements are working login/signup feature,

decide on a handover time of the fully completed product to clients, material UI for each

page, migrate from SQL to Firebase, put firebase private key in. env and include. gitignore

and continue working on database features. For the sprint 4, we also have agreed

requirements which include deploying on Heroku/Firebase, migrate from SQL to

PostgreSQL, material UI for each page, continue working on database features, view count,

export as pdf, look into AWS storage and PostgreSQL for hosting on Heroku in case

Firebase doesn’t work out.

The things that we all agreed to improve in the future are keep meetings short and precise,

work on frontend display, clean up backend code and fix bugs in contents that are viewable

when logged in/not logged in.

**Implemented Features**

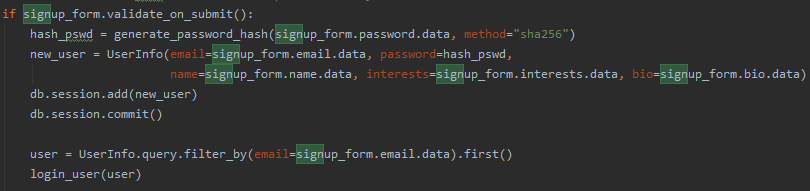
|  |  |  |
| --- | --- | --- |
| Feature Location | Feature Sets | Description |
| Landing page | Basic feature | Create a new account |
| Landing page | Basic feature | Login |
| Every page | Basic feature | logout |
| Profile page | Basic feature | Edit the Profile |
| Profile page | Extended feature | Upload the profile picture |
| Profile page | Extended feature | Export the profile as PDF |
| Portfolio page | Basic feature | Upload the files in the Portfolio |
| Portfolio page | Basic feature | Add comments to the Portfolio |
| Portfolio page | Extended feature | Display the image in the page |
| Portfolio page | Extended feature | Download the image |
| Portfolio page | Extended feature | delete the image |
| Portfolio page | Extended feature | Display the Videos in the video table |
| Portfolio page | Extended feature | Download videos |
| Portfolio page | Extended feature | Delete videos |
| Portfolio page | Extended feature | Display the Word document in the word document table |
| Portfolio page | Extended feature | Download one word document |
| Portfolio page | Extended feature | Delete one word document |
| Portfolio page | Extended feature | Display the PDF in the PDF table |
| Portfolio page | Extended feature | Download a PDF |
| Portfolio page | Extended feature | Delete a PDF |
| Portfolio page | Extended feature | Display the Miscellaneous in the Miscellaneous table |
| Portfolio page | Extended feature | Download a Miscellaneous |
| Portfolio page | Extended feature | Delete a Miscellaneous |

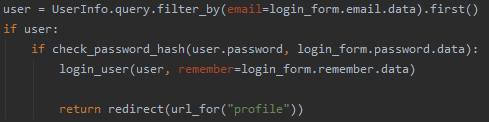
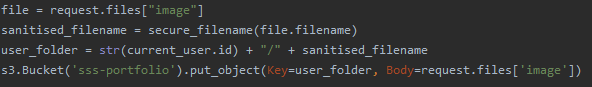
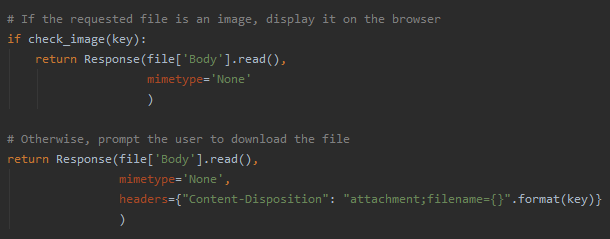
The main features of the system are all on the Portfolio page. All these features in the table are implemented.

• **Description of key algorithms**

The most important algorithms in our project is to upload, retrieve and display items from the S3 storage in the portfolio page, as well as account management such as user creation and authentication. Below will discuss the algorithms of each of the aforementioned features.

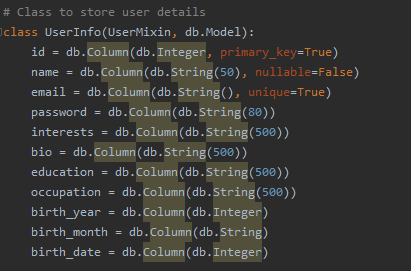
* **Account Creation**: The process of creating an account is handled in the **signup** route. The route uses the SignupForm class and asks a user to enter an email, password, name, interests, and bio. Once the form is submitted and validated, the form data is added PostgreSQL and a new user is registered. Note that rather than storing the user’s password as a plaintext, the database will store it as a hash using the generate\_password\_hash function from Werkzeug. Once the user has successfully made an account, they will automatically be signed in via the Flask login\_manager by querying the newly created user’s email, and redirected to their profile page.

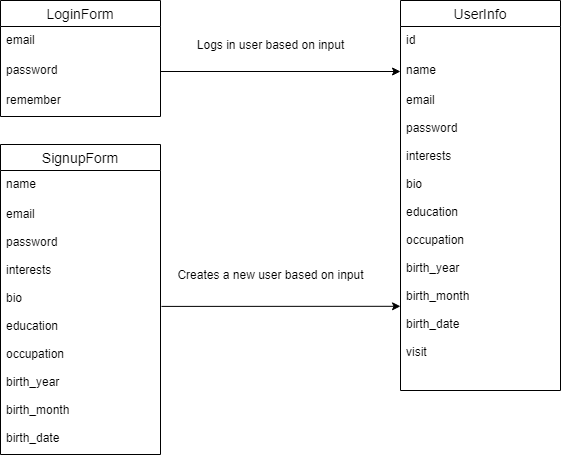


* **Account Authentication**: The process of logging in is handled in the **login** route. Similarly, with the signup procedure, the user fills out a form that requests their email and password. Once the form has been submitted, the app will query the email from the database and if the email exists, the check\_password\_hash will be called to hash the submitted password. If it matches with the hash stored in the database, the user will be logged in via Flask login\_manager and redirected to their profile page. 
* **File Upload**: This function is handled in the **s3\_upload** route, which is called when a user selects a file and clicks the upload button in the portfolio page. The filename is then sanitised, and its destination folder will be determined by querying the current user id. For example, if the current user’s id is 10, then the file will be stored in “10/filename” in S3. 
* **File Download**: This function is handled in the **s3\_download** route, called when a user clicks the download button in the portfolio. The file is retrieved using its key, then the function first checks whether the requested file is an image, which will be displayed in the browser. Otherwise, the function will prompt the user to download the file. 
* **File Deletion**: This function is handled by the **s3\_delete** route, which is called when a user clicks the delete button in the portfolio. The function simply deletes an item given its key. 
* **Portfolio Display**: When a user navigates to their portfolio page, it will display a list of all items uploaded by the user. To determine which items to display from S3 and how the items are categorised, several filter functions are used. In the **art\_portfolio** route, a bucket variable is created to access S3, then a list of all stored files are passed to the **art\_portfolio** html page. From there, a for loop used in tandem with the **show\_own\_items** filter function to show only items uploaded by the user. The **show\_own\_items** function checks whether an item is in the user’s directory based on their id. If an item passes the filter, it will be displayed in the portfolio. In addition, each item will also be subjected to a filter function that checks their type and be categorised accordingly. For example, if the **filter\_image** function returns true for an item, then it will be displayed in the images section.
* **Avatar Display/Upload**: By default, the user’s avatar will be a pre-set stick-figure stored in the Avatar folder in S3. If a user wants to upload a new avatar, they can do so in their profile page. The route that handles avatar upload is **s3\_upload\_avatar**, which takes an image submitted by the user and renames it to the user id. For example, if the user id is 12, then the image will be renamed 12.png. The image will then be uploaded into the Avatar folder. The **get\_avatar** function handles avatar display, it first looks into the Avatar folder for an image named after the user id, displaying it if it exists. Else, it will display the default avatar.
* **View another user’s portfolio**: The owner of a portfolio can generate a link to share their portfolio for others to view. In the public version of the portfolio, viewers are able to see the portfolio items as well as download them. This is achieved in a similar way to displaying the user’s own portfolio, the only difference is that the options to upload and delete files are disabled. The link is created by appending the user’s id to the view\_user\_portfolio page.

**• Description of key classes and the application’s layers and database relations**

**UserInfo**: This class is used to store user information in PostgreSQL. It contains a user’s email, hashed password, name, bio, interests, occupation, education, date of birth, as well as their id used as a primary key. Below is a screenshot of the schema in Python, as well as a class diagram.





**Signup and login forms**: Below is the class schema for the signup and login forms.



**Testing:**

Testing procedures are handled by the test.py file, and tests are run with the following terminal command:

Python test\_flask.py -v

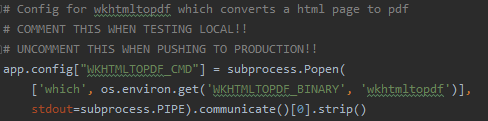
The system will then run all the test cases and return a report of which tests passed, as well as time taken for the process. The tests will see whether the app itself loads, and whether a user can access certain pages without being logged in, and whether accessing login-restricted pages will redirect them to the login page. In addition, the testing script also tests the site’s response to a successful login attempt (correct credentials, redirect to profile), a failed login attempt (incorrect credentials) as well as logging out when a user is logged in.

**Deployment guidelines**

The following instructions describe the process to test and deploy the code.

**GitHub repo**: <https://github.com/xye3/COMP30022_IT_Project.git>

**To run the test locally:** In the terminal, type “Python app.py” while the current directory is in the project’s root folder. The app will then be started in localhost port 9999. Open up a browser and navigate to <http://localhost:9999/>. In addition, comment out the following code block.



**To open the Heroku page**: Navigate to <https://sss-portfolio.herokuapp.com/>.

**To deploy code to Heroku**: Ensure that the Heroku CLI is installed, then log in to Heroku in the terminal. To add uploaded code into the Heroku repository, first add the remote into the local repo with the following line: heroku git:remote -a sss-portfolio. Then stage updated code with the following lines:

* git add .
* git commit -m “message”
* git push heroku master

The new code will be pushed to Heroku, and changes will be reflected.

**S3 Credentials**:

* AWS Access Key ID: AKIAIVUAFHRHT4F6FHEA
* Secret Access Key: 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
* Bucket Name: sss-portfolio

**Flask Secret Key:**

* OYddWvaBMZcZC0I\_s8vG6A

These credentials are stored in an .env file which is git ignored in the repository. For the production version, they are set in Heroku’s configurations.

**Boasting Page**

These are some aspects of our product the team is most proud of.

First of all, we have implemented the extra functionalities that our clients requested for, including a page view counter on the profile page and an export as pdf function.

The page view counter records the number of visits on the profile page for each user separately. It helps the owner keep track of how many people have visited their profile and use it to reflect on how successful their portfolio is designed based on the frequency of visitors. Then the user can do some adjustments to his/her portfolio accordingly.

Secondly, we implemented an export as pdf function for our profile page. It comes in handy for the creator, as the creator can have the option to download an offline copy of the profile page and submit it as an attachment to employers instead. In addition, it allows the user to print a hard copy so he/she can choose to hand it to employers in person.

Our product is also user-friendly. We aim to design a clean and neat portfolio system, so a new user can pick up how to use our website quickly and able to make changes easily. When browsing the website, the user will first be directed to our landing page, where it clearly identifies the purpose of the project. Then the user will be given a log in or sign up option depending if the user has registered before. Upon successful log in, the user will be directed to the profile page, which is the main page for displaying information, and can be simply edited with the “edit profile” button. In addition, there is a portfolio page where the user can upload different kinds of file. For example, posting a video showcasing one of his/her achievement. Therefore, the whole process of creating a profile on the website is intuitive and can be completed in a short amount of time.

Our product also maintains a clean database structure. All information is stored in two parts. The first part is PostgreSQL, which stores different kind of user information. It creates a unique id for each user and records the bio, email, password, name and any other text information. The second part is AWS S3 storage, which stores files for all other kinds of formats, such as images, videos and documents. This includes the profile picture in the profile page and the files uploaded in the portfolio page. The structure of the database ensures all the information uploaded to the server is saved in an organized manner, and can be easily extended if new fields are required for user information.