**Criterion E: Product development**

Advanced Techniques used in the production:

1. Flask Website framework coded in Python3 and HTML5
2. Implementation of relational database with SQLLite and Flask-SQLAlchemy
3. Package structuring of files.

**Blueprint of Website**

A close up of text on a white background

Description automatically generated

This is the layout of the website, which consists of a register page; a login page; a home page; an about page for the IB and AP program; a news page for my client to post all the news and updates onto, from which you can access each individual posts at their separate pages; a resource page which displays all the resources posted by the client; and an account page for individual users’ information.

**Advanced Technique: Building with Flask framework using Python 3 and HTML5**

My website is built using Flask, a micro web framework, and some of its extensions. The whole structure is relatively easy compared to other frameworks on the market. Essentially, it only involves three separate pieces of code shown here:

A screenshot of a cell phone

Description automatically generated

First of all, we import the Flask class, and create an instance of the class called app. This is how the Flask framework is installed and initialized in our code. Then we create a variety of routes, represented by the middle portion of the code, which we’ll get into further details later. Lastly, we run the instance declared earlier, and therefore the whole web application, through the last portion of the code. The condition if \_\_name == ‘\_\_main\_\_’ is basically saying that the application will run if python is directly running the code. Now, how exactly do we create different webpages through the framework is through the routes that we were discussing earlier. Every route is a separate page on our website. You’ll notice how a route is basically a function (def hello\_world() in the case above). The @app.route(‘/’) code on top is called a decorator, which binds that function with an URL, so whenever we visit the URL the website runs the function. However, most routes in my website aren’t as simple as an HTML header. In fact most routes will have to do with four separate files: routes.py, forms.py, models.py, and an HTML page. Here’s how each looks like:

A screenshot of a cell phone

Description automatically generated

Figure 1 The register route

A screenshot of a cell phone

Description automatically generated

Figure 2 The registration form

A screenshot of a cell phone

Description automatically generated

Figure 3 The User database table

A screenshot of a cell phone

Description automatically generated

Figure 4 List of all the HTML files

To demonstrate how a route, and therefore website, functions, we will do a case study on the register route. Looking at line 75 to 88 on picture 2, that entire function is what’s passed when the register page is visited. How the page works is that the page takes inputs from the end-user. Upon the input being submitted, the information gets saved into the database, and therefore an account is created for the user. Let’s break down this process. How exactly the website accepts input is through the “forms” written in forms.py file in picture 3, which uses the extension of Flask wtforms. We can see the registration form from line 8 to 23. The variables, like username and password, are what the forms takes in. The fields (i.e. StringField and PasswordField) are the types of input that the forms allow. Lastly, every form must have a SubmitField, which allows the user the submit the input. The forms are individually imported to route.py and called upon (i.e. line 79 in picture 2) when necessary. When the forms have been submitted, how we append the information into the database is through the help of SQLAlchemy, another Flask extension. In models.py in picture 4, we create each individual data table in the database. There’s one which stores all the user information (line 10 – 33). Each variable here, like username, password, image\_file, and more, are columns in the corresponding table. They are attributes that each item in the database will have.

A screenshot of a social media post

Description automatically generated

Figure 5 A clearer look at the user database table

This is a view of the User table that we’ll be using for the register page, provided by an app called Datum-Lite. Each row represents a user, with an item id, a username, an email address, a profile image, an encrypted password, and a level attribute. The table from models.py are also imported and called in routes.py when needed (line 82 in routes.py). We see that line 83 and 84 are commands that write the data from the form into the database. Lastly, we would render an HTML page for the users to have access to the form, and therefore registering their account. Picture one is the folder that contains all the html pages for different routes. Here’s the one for the register route:

A screenshot of a cell phone

Description automatically generated

Figure 6 The HTML page that's to be rendered by the register route

A screenshot of a cell phone

Description automatically generated

Figure 7 How the actual register page looks like

This is just very simple HTML code needing no further explanation, I’m just building a form can take in all the specified fields of input from the registration form. However, one thing worth noting is that all the content in the double curly-braces are variables that are defined in the corresponding route while the HTML page is rendered (in this case on line 88, where “form” in the HTML code represents the form in the route page that’s defined as the Registration form. Therefore, in register.html, for example, when {{form.email}} is called, it’s referring to the user input of the email field in the registration from that’s imported into route.py from form.py. This is roughly how a route, amongst many others, works. The repetition of the routes makes up the whole website.

**Advanced Technique: Constructing Relational Databases**

Sometimes when working with databases, the matter isn’t as simple as inputting/calling information from one and processing the information. For an instance, when creating the comments system for my individual post’s page, I ran into the problem where when the user inputs data into the database by submitting the comment, different posts’ pages won’t know which comments to display, since there was no way of identifying where each comment should go to.

A screenshot of a cell phone

Description automatically generated

Figure 8 A look at how captions are displayed

The problem was resolved through the implementation of relational databases. I created a one-to-many relationship between the table for the posts, and the one for the comments.

A screenshot of a cell phone

Description automatically generated

Figure 9 How the one-to-many relationship is established

Essentially what a one-to-many relationship is is a relationship where one post can have many comments belonged to it. This relationship is established by the db.relationship code on line 42 in the posts table and the db.ForeignKey code on line 52 for the comments table. When the relationship is established, we can see a column in the comment table named “post\_id,” which is specified by the foreignkey command on line 52. Here’s a look at it:

A screenshot of a social media post

Description automatically generated

Figure 10 How the relationship is shown on a table

From the first data table, we can clearly see the column “post\_id” at the very right, which is set to be “2.” This is the same ID as the one for the post which it belongs to, as we can see on the second table. Therefore, even if I were to make a comment on another post, the comments displayed on post 2 will still only be the three whose post\_id attribute = 2.

**Advanced Technique: Package Structuring of Files**

When coding the website, there were originally many instances of circular import, which is basically when you import items from another file, so the other file is ran when the import code in the first file is reached. Then on the other file, items from the first file is imported, but since the first file was terminated midway through running, the item was never defined yet, so the second file couldn’t call upon it and an error occurs. The most efficient way of solving the problem is through restructuring the whole project. Here’s how everything looked in the end:

A picture containing text

Description automatically generated

Figure 11 How the files are organized like a package

How the structuring of this works is that I put all the files and code into a single folder called “mainpage” and added an “\_\_init\_\_.py” file into the folder. Then I had a separate file called run.py to run the whole package of mainpage. Here’s what init.py and run.py look like:

A screenshot of a cell phone

Description automatically generated

Figure 12 Where all the applications are initialized

A screenshot of a cell phone

Description automatically generated

Figure 13 Where the package is ran

When flask detects an \_\_init\_\_.py it automatically recognizes the folder as a package. The \_\_init\_\_.py file is where all the classes are imported and initialized (i.e. line 7 and 9 where the flask app and the database app are initialized). The uniformed initialization of all the classes is the basis of how the other files in the package are able to function. Then in order to run the whole package, we use the run.py file outside of it, which simply imports the flask application initialized in \_\_init\_\_.py and run the application using the code that we discussed previously. Structuring the project in such a manner is a lot more clean, efficient, and organized.

Word Count: 1304