Contents

[First web-based approach – Python Flask and Bootstrap: 2](#_Toc99121381)

[Why I decided to change from a GUI to a Web Application: 2](#_Toc99121382)

[Creating a virtual environment using pip: 2](#_Toc99121383)

[Installing all modules and dependencies for the web application: 2](#_Toc99121384)

[Creating the directories and files for the program: 3](#_Toc99121385)

[Setting up the website directory: 3](#_Toc99121386)

[Creating the files that will be needed for the application: 3](#_Toc99121387)

[Setting up the application: 4](#_Toc99121388)

[Creating the views: 4](#_Toc99121389)

[Creating the base template: 4](#_Toc99121390)

[Base.html: 4](#_Toc99121391)

[Home.html: 5](#_Toc99121392)

[Registering the blueprints to the application: 5](#_Toc99121393)

[Updated version of “\_\_init\_\_.py”: 5](#_Toc99121394)

[Running main.py to check everything is working and initialised properly: 5](#_Toc99121395)

[Creating the login page (front-end): 5](#_Toc99121396)

[Adding some extra pieces to “base.html”: 5](#_Toc99121397)

[Second web-based approach – Python Flask and TailwindCSS: 6](#_Toc99121398)

[Why I decided to change from bootstrap to tailwind: 6](#_Toc99121399)

[Preparing the workspace: 6](#_Toc99121400)

[Creating the virtual environment: 6](#_Toc99121401)

[Entering the virtual environment: 6](#_Toc99121402)

[Installing the python packages needed: 6](#_Toc99121403)

[Creating the files and directories I need to begin this application: 6](#_Toc99121404)

[Installation and Setup of TailwindCSS: 7](#_Toc99121405)

[Setting up the main CSS file to inherit Tailwind’s classes: 8](#_Toc99121406)

[Building the application: 8](#_Toc99121407)

[Initialising the Flask Application: 8](#_Toc99121408)

[Creating a test blueprint: 9](#_Toc99121409)

[Building the home page: 10](#_Toc99121410)

[Setting up the route: 10](#_Toc99121411)

[Start of development based on designs: 10](#_Toc99121412)

# First web-based approach – Python Flask and Bootstrap:

This section documents my first initial approach to a web-based application using python’s flask and the CSS framework called bootstrap. This method didn’t make it past the initialisation stage, the reasoning behind this will be explained later on in the document.

## Why I decided to change from a GUI to a Web Application:

I took the decision to change my project from a GUI-based application to a Web-based application because with a web-based app you are offered a lot more freedom in terms of design and features in my opinion. For example, the HTML and CSS aspects alone of the web application allow for great design opportunities compared to that of a pythonic GUI. Furthermore having the ability to merge HTML, CSS and JS with Python allows for many more front-end features for the application which benefits the overall user experience whilst using the application and also provides support for all devices, since it will be accessible via a browser rather than a python script and/or executable.

## Creating a virtual environment using pip:

Text

Description automatically generated with medium confidenceFirst of all, I create a virtual environment by opening command prompt in the directory I’m going to be making the web application in and write the command “pipenv –python 3.10.1” to create a virtual environment in python version 3.10.1.

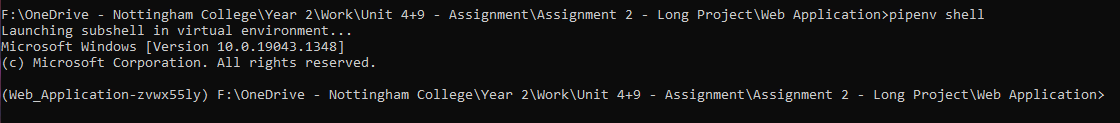
## Installing all modules and dependencies for the web application:

Text

Description automatically generatedText

Description automatically generatedA picture containing text

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidenceUp next I need to install all of the modules I will be using for this application. To do this, I start a shell within the virtual environment and install everything as seen below:

Text

Description automatically generated

## Text Description automatically generatedCreating the directories and files for the program:

To begin with, there are 2 things that need to be created before doing anything else. I need to create the “main.py” file which is the file that will be run and starts the entire web application up, and the “website” directory which will contain the main bulk of the code for the application itself. Also after creating the virtual environment and installing the dependencies, there are 2 files already in the main directory called “Pipfile” and “Pipfile.lock” which contain details on the environment such as packages and version requirements etc.

## Text Description automatically generated with medium confidenceSetting up the website directory:

For the website directory, there are a few sub-directories that I need to make which will all contain the front-end elements of the application (HTML, CSS and JS). These sub-directories will be called “templates” and “static”. Furthermore, inside the “static” sub-directory there will be 3 more sub-directories each called “styles” for CSS files, “js” for JavaScript files and “img” for any images. This sub-directory is called “static” because they contain static files, these files should not be changed at all once deployed. Additionally, the sub-directory “templates” speaks for itself in terms of its naming, you create templates which you later render in the python files which will be displayed later on in development.

## Text Description automatically generated with medium confidenceCreating the files that will be needed for the application:

This is where I create all of the python files which make the web application function, as well as all of the front-end files that the python files will render and utilise once the python functions have been written. The “\_\_init\_\_.py” file contains all of the main configuration and the setup of the application. The “auth.py” file will contain all of the code that makes the login function work, and handles requests from the login form that will be written in the front-end files, and again rendered by the python functions. The “database.db” file is as said in the file name, the database for this web application, now because this database is local it may pose as a security threat, however since the vital data such as a user password will be encrypted it won’t be an issue. The “models.py” file will contain the database classes that will be called in “auth.py” which allows us to write to the database. The “views.py” is essentially the same as “auth.py” however instead of containing the database functions and handling requests, its simply there to handle the rendering of the templates found in the front-end files. Finally, the front-end files that will be used and explained later on are “base.html”, “home.html”, “login.html”, “styles.css” and “main.js”.

## Setting up the application:

A screenshot of a computer

Description automatically generated with medium confidenceText

Description automatically generatedText

Description automatically generatedThis is where the actual back-end coding begins. The first file that will be written to is the “\_\_init\_\_.py” file, we need to initialise the application and some other things later on (because they aren’t need right now). Most of what is going on will be explained through comments in the screenshots provided below. The terminal screenshot displays the password generator I used for making the secret key for the application.

## Creating the views:

Graphical user interface, text

Description automatically generatedWhen it comes to Flask and web applications in particular, I need to create views that perform functions, their main use is to handle requests on specific pages and return a render template function that displays the front-end of the web page.

## Creating the base template:

When using python alongside html for a web application, you have the ability to write python-like code within the html files, this is specifically useful for creating blocks and using variables from the route functions. The blocks are best useful for creating base html files which act similar to a template, as shown below.

## Text Description automatically generatedBase.html:

## Text Description automatically generatedHome.html:

## Registering the blueprints to the application:

In order to utilise these blueprints (views), we need to register them in the “\_\_init\_\_.py” file, by doing this I import the “views.py” file and insert it into the “register\_blueprint” function alongside its “url\_prefix”. How that is done can be found in the screenshot below.

## Text Description automatically generatedUpdated version of “\_\_init\_\_.py”:

## Graphical user interface, application Description automatically generatedRunning main.py to check everything is working and initialised properly:

Up until now, after the first time running the main.py file, what we should find is the page title is “Home · College Tournament” and not “ · College Tournament”, and we should see a h1 header saying “Hello World!”. This is the result after running main.py and viewing the website on localhost.

## Creating the login page (front-end):

Since this application will only be available for the teachers to update, we need to have some kind of restriction in place to ensure students can’t get access to the website and alter scores and entities’ (teams or individuals) data. This will be in the form of a login page, with no signup option available (the login details will be pre-defined in the database) that only teachers will know and have access to. This will be a stylish but simple login form that any user will be prompted with, and the home page route and any other route we create within the application will use a special decorator that only allows logged in users to view. The screenshots below are showcases of the front-end code alongside some explanations of what certain pieces actually do and why I have used it.

## Adding some extra pieces to “base.html”:

The first element that I will be adding to the base is a navigation bar, this will later have some alterations to make the navigation and names cleaner and more concise but that will only be available to us once the login page, both front and back-end are complete.

# Second web-based approach – Python Flask and TailwindCSS:

## Why I decided to change from bootstrap to tailwind:

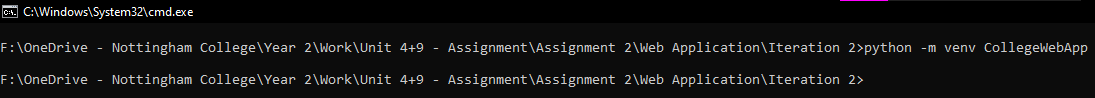
Graphical user interface, text

Description automatically generatedJust before I was about to fully commit to the front-end of this application, I actually discovered a better CSS framework that I could use that would benefit me more for this project in particular, called TailwindCSS. Despite this framework being more compatible with JS-based frameworks such as Vue, it is still applicable to a pythonic applications. The main feature of Tailwind that appeals to me and this project more than Bootstrap does, is the fact that Tailwind provides much better customisation and also has a few other features such as a VSC extension providing “intelli-sense”, making development easier, and better utility classes.

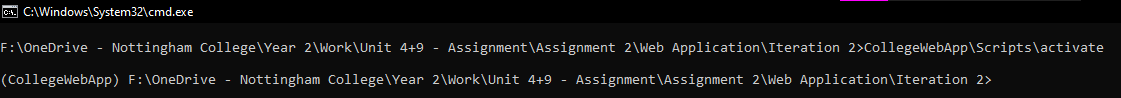
## Preparing the workspace:

Before creating this, I am keeping the first and current iteration of this project in the same directory, just separating them into 2 sub-directories to keep them in the same place. The previous iteration being named “Iteration 1”, and the second (current) iteration being named “iteration 2”.

### Creating the virtual environment:

For this iteration, I have changed from using “pipenv” for the virtual environment to “venv”. This serves the same purpose as Pipenv however it is easier to use.

### Entering the virtual environment:

To enter the virtual environment, it is extremely simple. I just need to run the command “CollegeWebApp\Scripts\activate” and I will be inside the virtual environment, completely ignoring any other packages I already have installed on my machine. You can tell that you have successfully entered the virtual environment when the name of the environment is in brackets to the left of the directory location in command prompt.

### Text Description automatically generatedInstalling the python packages needed:

Graphical user interface, text

Description automatically generatedInstalling the required python packages is extremely simple. The command is simply “pip install <package\_name>” and this will fetch the package from something called PyPi and install it to the virtual environment for me to use in my application. The packages I need to install for this project are the following:

* Flask
* Flask-login
* Flask-sqlalchemy

Note how I only install flask-login and flask-sqlalchemy. Since Flask itself is a dependency of both of these packages, they will automatically install Flask, and a few other dependencies if they are not present in the environment they are being installed to, removing the need for me to install Flask separately.

### Creating the files and directories I need to begin this application:

To begin with, there are only a few files and directories I need to make to have the structure of the application initialised. To begin with, to separate the application from the virtual environment that I created called “CollegeWebApp”, I will create a directory which will act as the application’s “root” directory called “app”. This will contain all of the files relating to this application. Text

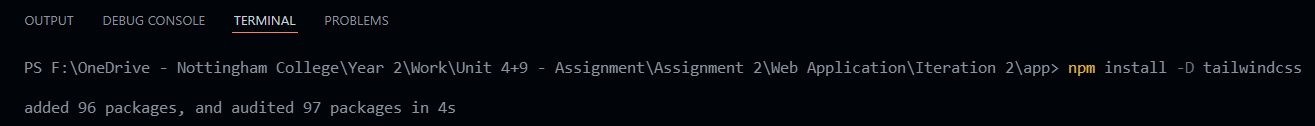
Description automatically generated

Using the windows command “dir /S” will display every file and directory inside the directory you run the command in. The screenshot above displays the root directory’s contents at the top, as well as the contents of the “website” directory at the bottom, all of which are accompanied with an explanation on their purpose for this application.

### Installation and Setup of TailwindCSS:

**IMPORTANT NOTE:** Node.js is required for the installation of TailwindCSS.

From this point onwards, any terminal usage I’ll be doing inside the built-in terminal on my IDE, Visual Studio Code (VSC).

First of all, once node.js is installed I need to install the CSS framework I’ll be using for this project, TailwindCSS. I can do this by running the following in the terminal.

“Added 96 packages” shows us that tailwindcss has been successfully installed, and to double check, I can check the newly created “node\_modules” directory in my workspace and check for the “tailwindcss” sub-directory, which is shown to the right.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generatedNext I need to generate a configuration file for tailwind. Features like this in particular are what made Tailwind a more favourable option over bootstrap, its flexible configuration and vast amount of utility classes. Running “npx tailwindcss init” creates a file called “tailwind.config.js” which is where any configuration for tailwind will take place.

Graphical user interface, text

Description automatically generated

To finish off initial configuration, I just need to insert the file location of all of my template files, these being the HTML files. The screenshot below shows this.

### Graphical user interface, text, chat or text message Description automatically generatedSetting up the main CSS file to inherit Tailwind’s classes:

First of all, I need to create the CSS file that will be holding all of tailwind’s classes. I will be creating a file called “input.css” in the “website/static/styles” directory and adding 3 short lines of code that when they are run, will import all of Tailwind’s classes.

Finally, in order to have these classes imported, I need to run the following command in the terminal: “npx tailwindcss -i ./website/static/styles/input.css -o ./website/static/styles/styles.css --watch”

Text

Description automatically generatedAs you can see below, the above command has been run and we now have a new file called “styles.css” which now contains all of the imported TailwindCSS classes. “input.css” is no longer needed so that file will be deleted and now the setup of both Tailwind and the workspace environment is complete.

## Building the application:

### Initialising the Flask Application:

The file structure will be slightly different compared to the previous iteration of this web application. Upon doing further research into Flask applications I have discovered better ways to utilise this package and this will be documented throughout this project.

Text

Description automatically generatedTo begin with, I need to head into the “website” directory and enter the file called “app.py” which will contain the class that manages the entire application.

This snippet here is the very bare bones, basic initialisation of the Flask application. At the minute this application will do entirely nothing, but can still be run and will create a local development server as seen below.

Text

Description automatically generated

### Creating a test blueprint:

In Flask terminologies, a blueprint is essentially something we register to our application that contains a collection of routes and other app-related functions, a blueprint is simply an object that allows you to define application functions, without requiring the application object (the class I defined earlier inside “app.py”) at the time of definition.

First of all, I need to create a new file called “views.py”, this file will contain a singular blueprint that will handle all web pages that are not related to logging in to the application (log-in page). They will be put in a separate file later on in development of the application. The purpose of each element in the screenshot below is explained with a corresponding comment.

A screenshot of a computer

Description automatically generated with medium confidence

#### Assigning the test blueprint to the application:

Text

Description automatically generated

Graphical user interface, application

Description automatically generatedNow if I run the application, and go to <http://localhost:5000/> in my browser, it should render the HTML that I wrote in the test function inside “views.py”, as seen below. (Screenshot on the right is from inspect element, to check correct HTML elements have been pushed)

Graphical user interface, text

Description automatically generated with medium confidence

At this point, everything is working as intended, so now it is time to start building the home page, this will also be where we can verify that the installation of Tailwind is properly successful by attempting to use the classes that have been imported into “styles.css”.

## Building the home page:

*\*A full list of tools, and links to them where necessary, can be found near to the end of this document.*

### Setting up the route:

The home page is going to have a number of features, as shown in the designs however like any other project, they are bound to have reworks or changes made to them mid-development. Any changes that are made from the initial designs will be documented and justified towards the end of this document. Before the development of the home page begins though, a “home.html” file needs to be created inside of the “templates” directory, and it needs to be linked to a route inside the newly created “views” blueprint.

Text

Description automatically generated

*\*Note: the test route has now been changed to “/test” and home takes “/”.*

To ensure the routes have been switched properly and the new “home.html” template is being rendered, I will add some markdown to the HTML file, differentiating from the html defined in the test function to make sure everything is updated.

Graphical user interface, application, chat or text message

Description automatically generatedA helpful feature from Flask is that I don’t need to stop, and restart the application after every change, since the application is in development mode, it automatically updates the contents of the app after every save of the code. The updated “/” route can be seen in the screenshot on the left, and the usage of “/test” can be seen on the right.

Graphical user interface, application

Description automatically generated

The test function is no longer needed so I’m going to remove that. From now is where the actual HTML based on the designs produced begins.

### Start of development based on designs:

#### Preparing the home.html file:

Before the elements are created, the html file needs some tags and external files to be linked as seen below. You’ll notice that I referenced the CSS file slightly differently than a normal CSS file would be. This has nothing to do with it being Tailwind, it’s actually due to the template tool that Flask uses to render the HTML, called Jinja2. It is also what allows me to use python in the HTML file later. I have also added a stylesheet link to flowbite’s framework as well, since some elements in this project will require some classes that flowbite also has to offer.

A screenshot of a computer

Description automatically generated

#### Navigation Bar:

This is a preview of the navigation bar in the designs I had produced:

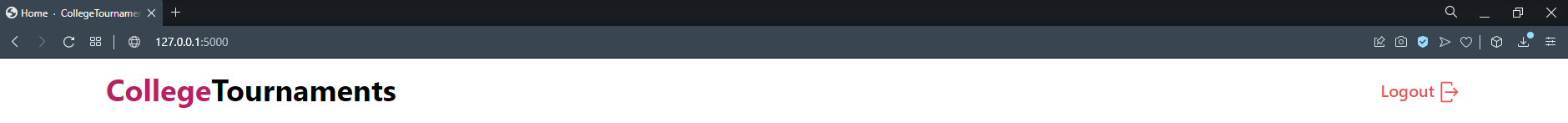


##### Code:

A screenshot of a computer

Description automatically generated with medium confidence

##### Output on browser:



Compared to navigation bars that you would find on most websites, the navigation bar for this project in particular is quite abnormal, since there is only a logout button and no navigation buttons other than that. The logout button currently has no functionality other than linking to a currently non-existent route, the login and logout routes will be created once the main page is complete.

#### Background:

Now that the navigation bar is in place, I can set the background for the div/section underneath it, which in my opinion is a key factor for how clean and appealing the design is. I’ll implement this simply by setting the background-image of the next div to be the image and specifying a pre-defined height that the image will be placed on the website as.

##### Code:



##### Output on browser:

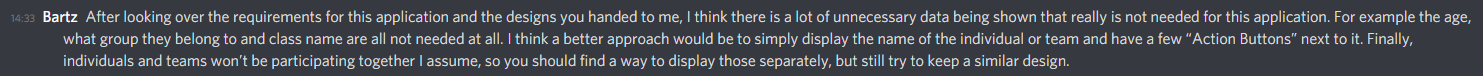
Background pattern

Description automatically generated with medium confidence

The classes that come with tailwind have made it extremely simple for me to set specific widths for the image such as “w-full” and “h-vh-full”, these 2 classes force the image width to be the full width of the screen, as well as the full height of the screen as well. Furthermore, I have used the “flex”, “items-center” and “justify-center” classes to later allow me to flexibly place the elements onto the page, and have them centred.

#### Dashboard:

The dashboard for this project is the elements in the center of the designs. It consists of 4 cards, and is placed on a white background. Now upon reviewing the design initially, I have decided that the settings panel is no longer needed, since there would be no configuration needed for this web-application since the user can freely zoom in and out themselves with their browser and no other configuration settings would be needed for an application like this. Furthermore, I also decided that there should be a more obvious way to add participants, as well as adding a score to their player data, so after consulting with a peer of mine, they gave me a few recommendations based on the designs I have created, and I will do my best to implement these.

 A few ideas that initially come to mind are the following:

* Move the total participants (currently displayed next to participants title) underneath the title and have it separately show the total of Individuals and Teams, but also have an icon accompanying both pieces of text that allows them to add the corresponding entity (Individual/Team).
* Add a new icon next to the “delete” icon on the participants display tab that allows the user to add a score to that member from an event they just played, rather than having the “Adjust Scores” button hidden in the bottom right where it doesn’t even correlate to the participants.
* The only elements needing to be displayed for the participants tab is the name of the individual/team and an “actions” tab as my peer advised me to do in the feedback he gave.
* Add a dropdown select menu of sorts to differentiate between displaying individuals and teams. Implementing it like this will keep the layout and design the same, and still allow for some kind of clean separation between the 2 participant types. This select menu element would also be implemented on the Leader-board and Events tab.
* Finally, now that the need for the settings tab has been eliminated, I will extend the events tab’s height to be equal to the tabs beside it, and completely remove the settings tab.

##### Getting the background card:

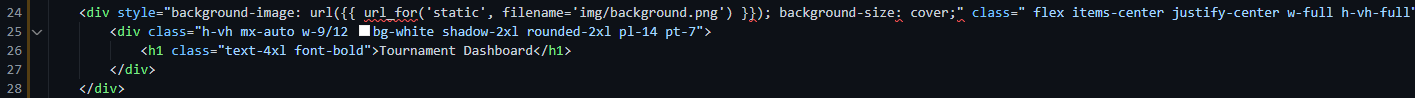
###### Design:

Graphical user interface

Description automatically generated

This is the initial design for the dashboard. First of all I need to get the main white background element in place to then put the (now 3) tabs on top of it.

###### Code:



To create the white background, I simply just need to create another div inside the container which has the background image inside of it and add a few classes to it to shape and size the element. For example, “bg-white shadow-2xl rounded-2xl” are what makes this a rounded rectangle with a shadow, “bg-white” is of course what makes it white, and “pl-14 pt-7” adds something called a padding which is what puts the rectangle in the position it’s in. I also went ahead and added the “Tournament Dashboard” title on the top left of the element as well. The classes assigned to the h1 element are simply text sizing, and making the text bold.

###### Output on Browser:

Text

Description automatically generated

##### Adding the 3 cards on the main card:

First of all, the 3 cards need to have a container that hold all 3 cards in, this container in particular needs to be a flex container and there will be 1 div per card inside this container div.

The only difference in layout between the initial design and what I have produced below is that the settings tab is now gone, and the events tab is full height alongside the 2 tabs to the left of it.

###### Design:

Graphical user interface

Description automatically generated

###### Code:

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generatedI have added some short comments explaining what each div does for readability purposes. I have made it so the divs have a specified width and height, a background colour and a shadow to add a nice visual effect to the dashboard. Finally, the “overflow-scroll” is a custom class I have made myself, which will display a scroll bar, and hide any text that goes outside of the div it is specified in. A real example of this will be shown once I start displaying data on the page.

###### Output on Browser:

Text

Description automatically generated

##### Participants Header:

The participants tab is going to be the lengthiest section of code in the entire application. This is because upon taking feedback from my peer and evaluating how I could best implement it, it involves implementing “modals” that pop up when you click an icon, for both entering data as well as confirming a delete. The development of that will be shown later once I get to it. To begin with though, I need to download the icon and add the title of the tab.

###### Design:

Table

Description automatically generated

###### Code:

Text

Description automatically generated

This snippet contains the icon, title and the select dropdown menu I mentioned previously. I have given the options a value of “pa\_...” because I will have multiple select menus that relate to individuals and teams, so I have given them a prefix of “pa\_” for participants in this scenario. The values also help to identify which option was selected, and allows me to display data based on what value is selected.

###### Output on Browser:

Graphical user interface, text, application, email

Description automatically generated

##### Participants table:

Before I begin the front-end for this section I need to carry out a lot of back-end code. This includes the creation of and linking to a database, as well as creating models that can be called to then insert data into the database.

###### Creating the Database and Database Models:

First of all, I need to create a file called “models.py” in the website directory. Then I need to import some flask modules that allow me to create these special database models, and then create said models. Evidencing of this is in the screenshot below.

I will be creating the database inside the “models.py” file, however I will import it into my “app.py” file and initialise it on start-up there. The reason I decided to do it like this is because if I define the database in “app.py”, I will get “circular import” errors which means I import 2 files to each other, so I will simply define the database in the models file, and import it into the app file, meaning models does not need to import app into the file. This will be a local “.db” file and I will use Flask\_SQLAlchemy to allow me to interact with this database. The code on how to do this is below.

Text

Description automatically generated

There will be more added to this file later on to help with the login feature.

###### Creating the database file and connecting to it:

In order to interact with the database, I need to initialise a connection with it, but database creation also needs to be automated with the models created in models.py due to the various data types and column constraints listed in the screenshot above.

Text

Description automatically generated

###### Adding data to the database:

The data that I’m about to add to the database is only for testing purposes, the database will be cleared when it is sent off to the client. There are also a few statements I need to write in order to add this test data to the database, I will write this code under the home function in the “views.py” file so that all I have to do is load the home page, and it will execute the statements I write.

Text

Description automatically generated

Here I have just defined 3 arrays (lists) containing data that I want to commit to different tables on the database. I simply loop through every item in the list and create a new instance of each of the models and then add it to the database. I randomly generate a number from 1 to 20 as a placeholder just so that each entry has a number of points defined. Giving each entry a different number helps to evaluate whether future sorting statements work when creating a leaderboard from the data.

After reloading the home page, the statements are run and as seen below, the data was entered to the database.

Table

Description automatically generatedTable

Description automatically generatedTable

Description automatically generatedText

Description automatically generated

To allow me to view this data, I am using an application called “DB Browser for SQLite”, it allows me to open my database file and view the tables and the data they contain visually, rather than having to run SQL statements to fetch the data from the database to view it.

###### Handling the data and displaying it:

This section is the first instance of where I will be using python code (Possible by using Jinja2 with Flask) inside my HTML file. I will be using it to loop through the lists that contain the data from the database, and displaying it in a styled table in the participants tab. I will also have to add some JavaScript code to make the select menu element I implemented previously functional, and display data based on what option they choose.

First of all, I added a notice to the user to select an option so they don’t miss it, or in case they don’t know what to do:

A screenshot of a computer

Description automatically generated with medium confidence

Next I need to actually fetch the data from the database inside of “views.py”, and feed it into the HTML file via a kwarg in the render\_template function

Text

Description automatically generated with medium confidence

Text

Description automatically generated

Here I create a function called “fetch\_db” that will query the database and fetch every entry from the individuals, teams and events tables and returns a tuple containing this data. I chose to return it as a tuple so that the data cannot be edited, even by mistake, as the program continues. I pass the individuals and teams into the HTML file via a kwarg as I mentioned previously, and they can now be used and manipulated inside of the HTML file.

Graphical user interface, text, application, chat or text message, website

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidenceBelow is the first instance of me using python/Jinja2 inside my HTML file. What this snippet does is it takes both of the individual and teams lists I defined in the screenshot above, and I use “individual|length” or “teams|length” to display the total amount of individuals and teams entered into the tournament. I also added an extra, where it would only display a button to add a new individual or team as long as the total of individuals or teams is below their respective maximum entries (20 individuals, 4 teams). Finally, I have added the “data-modal-toggle” attributes to the buttons because that will be what links them to the modals I create later on in the application, that allow the user to add a new individual or team, and since they will display different modals, they have different IDs attached to them.

The 2 text elements displaying the totals won’t always be showing, I will add a JavaScript function later on alongside the table displaying all of the data to only display a certain text element based on the select option that is chosen from the menu above it. But for now, both elements will be showing.

Next up is the tables displaying all of the participants. Initially, 2 tables will be showing because I will have separate tables defined for the individuals and the teams, but I will be implementing some JavaScript that manipulates the relevant CSS classes to display or hide certain elements based on the selected option.

Text

Description automatically generatedText

Description automatically generated with medium confidence

The screenshot above not only shows the overflow CSS class I created in action, but also that the data has been correctly fetched and imported from the database into the HTML file.

###### Graphical user interface, text, application Description automatically generatedHiding data on default and giving select menu functionality:

Text

Description automatically generated with medium confidenceText

Description automatically generatedText

Description automatically generated

The screenshot above displays the JavaScript that controls what data is displayed based on the select menu value. However for this to be functional, I need to reference it in the HTML file, which imports the JavaScript in this new .js file located in the static/js directory. I also need to reference a jQuery CDN to allow for me to run jQuery (a JavaScript library) in my imported JavaScript.

The results of this can be seen on the right of the JavaScript screenshot. I have defaulted nothing to be displayed inside of this tab if nothing is selected, and the notice for the user that was previously there will be the only thing that is displayed, as seen in the screenshot below.