

Development

For the sake of conciseness and clarity, when the document outlines certain HTML and CSS files, such as `/templates/login.html`, it will not show all of the different iterations of the page. Major changes, such as reorganisations of the page, would be displayed in later sections, but the actual process of originally designing the website is not documented to make the document easier to read. A lot of the design process involved tweaking classes, re-ordering HTML elements, and adding new or modifying existing CSS blocks.

Stage 1 - Setting up the website

Before creating the database system, I decided to get the website backend up and running, and design the login and signup pages, to make it easier to test certain elements of the database. This includes setting up the template design for the frontend, creating a form system with verification in JavaScript, and performing validation server-side. The first thing I did was get the flask backend running. For this, I modified some code that I have used before when using Flask as a backend.

`__init__.py`

```
from flask import Flask

# The main class of the application
class Kraken():
    def __init__(self, host: str, port: int) -> None:
        # Create the Flask application and set a secret key
        self.app = Flask("Kraken")
        self.app.config["SECRET_KEY"] = "secret-key-goes-here"

        # Initialise the website pages
        self.initPages()

        # Run the Flask application
        self.app.run(host=host, port=port)

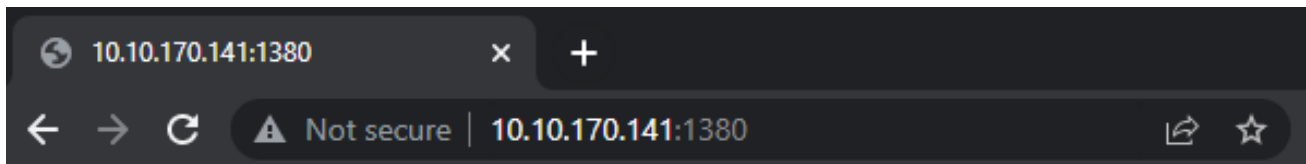
    def initPages(self) -> None:
        # Home page route
        @self.app.route("/")
        def main_index() -> str:
            # Display the returned text
            return "This is the homepage!"

if __name__ == "__main__":
    # Initialise the application on port 1380
    Kraken("0.0.0.0", 1380)
```

When run, it would output this to the console:

```
* Serving Flask app 'Kraken' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://10.10.170.141:1380/ (Press CTRL+C to quit)
```

And look like this in the website:



This is the homepage!

Having got the framework for the backend in place, I then created a basic template for the HTML (in Jinja) and some CSS code to work with it, so that I could test some of the functions that I wanted to use. These files were created at `/templates/test.html` and `/static/css/test.css`, and were deleted afterwards when I knew that the system was functioning as intended. To make sure that some of the modules of Flask were working as intended, I used the `render_template` and `flash` functions in Python, ran a loop in the HTML file using Jinja, and used `url_for` to import the CSS file.

changes to `__init__.py`

```
from flask import Flask, render_template, redirect, flash

# Flask is the application object
# render_template converts a Jinja file to html
# redirect redirects the website to another root function
# flash sends messages to the client
```

```
def initPages(self) -> None:

    # Home page route
    @self.app.route("/")
    def main_index() -> str:
        # flash sends a message to the next site that flask renders
        flash(["Apples", "Oranges", "Pears", 1, 2, 3])
```

```
# render_template takes the Jinja template file given in the templates folder
# and turns it into true HTML
return render_template("test.html")
```

/templates/test.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Kraken Test :)</title>

    <link href="{{url_for('static', filename='css/test.css')}}" rel="stylesheet"
    type="text/css" />
  </head>

  <body>
    <ol>

      {# Iterate through the list in the first flashed message, and create a
        list element for each one #}

      {% for element in get_flashed_messages()[0] %}
        <li>{{element}}</li>
      {% endfor %}

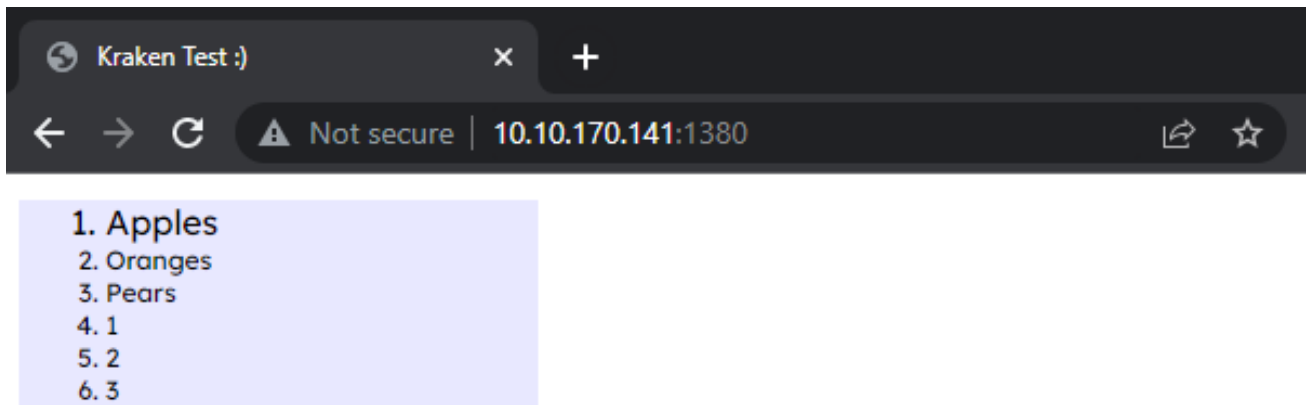
    </ol>
  </body>
</html>
```

/static/css/test.css

```
ol{
  background-color:#e8e8ff;
  font-size:12px;
  font-family:Lexend,sans-serif;
  width:200px;
}

li:first-of-type{
  font-size:16px;
}
```

When run, the website looked like this:



As you can see by the image, using the `flash` function, flask successfully sends the list `["Apples", "Oranges", "Pears", 1, 2, 3]` to the webpage for it to be recieved by the `get_flashed_messages` function. It also successfully managed to iterate through the list using `{% for element in get_flashed_messages()[0] %}`, and used the `url_for` function to import the stylesheet.

Before creating the database structure, I decided to create the frontend for the login and signup pages, so that it would be easier to test. Due to the above code working successfully, I started off by making a `base.html` template, that all of the other Jinja files would build on top of. I then created the `login.html` and `signup.html` files as well. This was made easier by my previous experience in web design, as I could use a library of CSS code that I have created from previous projects to speed up the design process. To be able to view the templates, I added some `app.route` functions to `__init__.py` using the `render_template` function.

changes to `__init__.py`

```
def initPages(self) -> None:

    # Home page route
    @self.app.route("/")
    def main_home() -> str:
        return "Hi o/"

    # Login page route
    @self.app.route("/login/")
    def auth_login() -> str:
        return render_template("login.html")

    # Signup page route
    @self.app.route("/signup/")
    def auth_signup() -> str:
        return render_template("signup.html")
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta http-equiv="Content-type" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Kraken</title>

    <!-- Site Meta -->
    <meta name="title" content="Kraken">
    <meta name="robots" content="index, follow">
    <meta name="language" content="English">

    <!-- Site Icons -->
    <link rel="apple-touch-icon" sizes="512x512" href="{{url_for('static',
    filename='img/icon/apple-touch-icon/apple-touch-icon-512-512.png')}}">
    <link rel="icon" type="image/png" sizes="512x512" href="{{url_for('static',
    filename='img/icon/tab-icon/tab-icon-512-512.png')}}">
    <link rel="mask-icon" href="{{url_for('static', filename='
    img/icon/mask-icon/mask-icon.svg')}}">

    <link rel="canonical" href="CanonicalUrl">

    <!-- Font Awesome Imports -->
    <!-- fa is a large icon database which you can import into a site -->
    <script src="https://kit.fontawesome.com/73a2cc1270.js"></script>
    <link rel="stylesheet" href="
    https://pro.fontawesome.com/releases/v6.0.0-beta3/css/all.css">

    <!-- Internal Stylesheet Imports -->
    <link href="{{url_for('static', filename='css/main.css')}}"
    rel="stylesheet" type="text/css" />
    <link href="{{url_for('static', filename='css/build.css')}}"
    rel="stylesheet" type="text/css" />

  </head>
  <body>

    <div class="page">
      <div class="application-container">

        <!-- Navigation bar, docked on the left hand side -->
        <!-- Contains the logo as a link to the homepage at the top, and a
        hamburger at the bottom -->

        <nav class="globalnav globalnav-vertical">
          <div class="globalnav-content">
            <div class="globalnav-list">
              <div class="globalnav-logo">
```

```

    <a class="globalnav-link globalnav-link-home link unformatted"
      href="{{ url_for('main_home') }}">
      
      <span class="globalnav-link-hidden-text visibly-hidden">
        Kraken
      </span>
    </a>
  </div>
  <ul class="globalnav-list">
    <li class="globalnav-item one fake" role="button"></li>
    <li class="globalnav-item two" role="button">
      <div class="hamburger hamburger--collapse js-hamburger"
        id="globalnav-hamburger">
        <div class="hamburger-box">
          <div class="hamburger-inner"></div>
        </div>
      </div>
    </li>
  </ul>
</div>
</div>
</nav>

```

<!-- Floating option modal for the navbar, which is opened and closed via the hamburger in the navigation bar -->

```

<div class="globalnav-floating-options">
  <!-- URL links are left blank for now as the pages have not yet
  been created -->

  <a class="globalnav-floating-option one" href="">
    <span class="globalnav-floating-option-content text header small">
      My Sites</span>
    </a>

  <a class="globalnav-floating-option two" href="">
    <span class="globalnav-floating-option-content text header small">
      Settings</span>
    </a>

  <a class="globalnav-floating-option three" href="">
    <span class="globalnav-floating-option-content text header small">
      Logout</span>
    </a>
</div>

```

<!-- Backdrop behind nav bar modal to apply a darkness filter behind the modal -->

```

<div class="globalnav-floating-options-backdrop"></div>

```

```

<!-- External Script Imports -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/
jquery.min.js"></script>
<script src="https://code.jquery.com/jquery-3.5.1.min.js"
crossorigin="anonymous"></script>

<!-- Internal Script Imports -->
<script src="{{url_for('static', filename='js/main.js')}}"></script>
<script src="{{url_for('static', filename='js/globalnav-floating-options.
js')}}"></script>

{% block content %}
{% endblock %}

</div>
</div>

</body>
</html>

```

For clarity, I have removed some unnecessary elements from the head of `base.html` , such as the open-graph protocol and some of the meta elements.

`/templates/login.html`

```

{% extends "base.html" %}

{% block content %}

<link href="{{url_for('static', filename='css/auth.css')}}" rel="stylesheet"
type="text/css" />

<div class="application-content">
  <div class="text-header-container">
    <h2 class="text header xl dark one">Kraken - Login</h2>
    <ul class="header-options">
      <li class="header-option header-option-login active notextselect">
        <h4 class="text header bold">Login</h4>
      </li>
      <li class="header-option header-option-signup notextselect"
        onclick="window.location.href=`{{ url_for('auth_signup') }}`">
        <h4 class="text header bold">Signup</h4>
      </li>
    </ul>
  </div>
  <div class="field-container active">
    <!-- Warning area for the form that uses the flashed warning message -->
    <span class="field-warning text italic">
      <!-- TODO: add code for flashed warning msg -->
    </span>
  </div>

```

```

<form class="field-options" method="post" action="/login/">

    {% set formItems = [
        ["Username", "Username", "text", "username", false, messages[2]],
        ["Password", "Password", "password", "password", true, ""]
    ]
    %}

    {% for item in formItems %}
    <div class="field-option field-option-name">
        <h4 class="text italic">{{item[0]}}</h4>
        <div class="field-input-container">
            <input class="field-input" placeholder="{{item[1]}}" type="{{item[2]}}"
            name="{{item[3]}}" value="{{item[5]}}">
            {% if item[4] %}
            <span class="eye-reveal">
                <i class="fa-solid fa-eye"></i>
            </span>
            {% else %}
            <span class="eye-spacer"></span>
            {% endif %}
        </div>
    </div>
    {% endfor %}

    <div class="field-option field-option-remember">
        <h4 class="text italic">Remember Me</h4>
        <div class="field-input-container">
            <input class="field-input" type="checkbox" name="remember"
            value={{messages[3]}}>
            <span class="eye-spacer"></span>
        </div>
    </div>

    <button class="field-submit btn secondary rounded slide" type="submit">
        <span class="btn-content text uppercase secondary">Submit</span>
    </button>

</form>

</div>
</div>

{% endblock %}

```



```
{% extends "base.html" %}

{% block content %}

<link href="{{url_for('static', filename='css/auth.css')}}" rel="stylesheet"
type="text/css" />
<div class="application-content">
  <div class="text-header-container">
    <h2 class="text header xl dark one">Kraken - Signup</h2>
    <ul class="header-options">
      <div class="header-option header-option-login notextselect"
onclick="window.location.href='{{ url_for('auth_login') }}'">
        <h4 class="text header bold">Login</h4>
      </div>
      <div class="header-option header-option-signup active notextselect">
        <h4 class="text header bold">Signup</h4>
      </div>
    </ul>
  </div>

  <div class="field-container active">

    <!-- Warning area for the form that uses the flashed warning message -->

    <span class="field-warning text italic">
      <!-- TODO: add code for flashed warning msg -->
    </span>

    <form class="field-options" method="post" action="/signup/">

      {% set formItems = [
        ["Name", "Name", "text", "name", false,
        ["Email", "name@domain.com", "email", "email", false,
        ["Username", "Username", "text", "username", false,
        ["Password", "Password", "password", "password", true],
        ["Repeat Password", "Again :/", "password", "password", "password-repeat"]
        ]
      ]
      {%}

      {% for item in formItems %}
      <div class="field-option field-option-name">
        <h4 class="text italic">{{item[0]}}</h4>
        <div class="field-input-container">
          <input class="field-input" placeholder="{{item[1]}}"
            type="{{item[2]}}" name="{{item[3]}}">

          {% if item[4] %}
          <span class="eye-reveal">
            <i class="fa-solid fa-eye"></i>
          </span>
          {% endif %}
        </div>
      </div>
    </div>
  </div>
</div>
```

```

        {% else %}
        <span class="eye-spacer"></span>
        {% endif %}
    </div>
</div>
{% endfor %}

<button class="field-submit btn secondary rounded slide" type="submit">
    <span class="btn-content text uppercase secondary">Submit</span>
</button>

</form>

</div>
</div>

{% endblock %}

```

The files make use of `url_for` to fetch many different URLs, including page icons, stylesheets, images, links to other pages, and scripts.

For example, the URL for the home button image in `base.html` is defined by `{{url_for('static', filename='img/icon/512-512/kraken-icon-png-'+navbarLogoColor+'-512-512.png')}}`. The variable `navbarLogoColor` is declared in child templates to define which colour should be used.

I also added some icons to be used as the logo for the website. They are located in `/static/img/icon/<resolution>/` where there are different folders for each resolution of the icon. These were generated by me using a bit of Python code that I have written previously. To add some variation to the site, there are three colours of logo: primary (blue), secondary (magenta) and gradient (a gradient of the two going from top left to bottom right). These colours match up with the primary and secondary colours of the website, defined in the CSS code.

The inheriting system is displayed here in these files, where you can see `{% block content %} {% endblock %}` in `base.html`, to define where the code block `content` will be inserted into the file, and then `{% extends "base.html" %}` and `{% block content %} {% endblock %}` in `login.html` and `signup.html`, which define what file will be extended, and which block to insert into.

The CSS and JavaScript for both pages is imported from the `/static/css/` and `/static/js/` directories respectively. `main.css` is my library of CSS code that I have collected (the syntax for classes is shown below), and `build.css` contains the CSS for the `base.html` template. `build.css` contains the code for the floating navigation options, that I tested by appending classes in devtools. This means that the only thing I need to do for it to work is to program the event listeners. The login and signup pages have a CSS file called `auth.css`, that defines the page-specific styling for the login form.

For JavaScript, I added in some abstract imports to be filled in later: `main.js` for global code, and `globalnav-floating-options.js` for when I code the navigation bar.

Syntax for `/templates/main.css`

Global classes:

- `.notextselect`
- `.nopointerevents`
- `.visibly-hidden`
- `.fake`
- `.box`
- `.no-inversion`

Positional Classes:

- `.relative`
- `.sticky`
- `.fixed`
- `.abs`
- `.static`

Text classes:

`.text` `<light|dark|primary|secondary|accent|grey-100 => grey-800>` `[italic]`
`[bold|thin]` `[ellipsis]` `[xl|large|default-size|small]` `[header|jumbo]`
`[lowercase|uppercase]` `[notextselect]` `[left|center|right|justify]`

Link classes:

`.text` `.link` `[classes for text]` `[disabled]` `[link-slide]` `[notformatted]`
(link-slide class requires the css variable `--link-slide-width`)

Btn classes:

`.btn` `<light|dark|primary|secondary|accent>` `<square|rounded|pill>` `[slide]`
`.btn.slide` `[from-left|from-right]`

If slide is set, the btn must contain a span element with syntax:

`span .btn-content .text` (all text classes apply here)

which contains the text of the button

A span like this is recommended even if the `.slide` class is not present so you can format the text inside separately

em classes:

`[classes for text]`

```
/* .section-content.fixed-width will set the width to 1440px, and will set the width to 100% when the viewport width is less than 1440px */
```

/static/css/build.css

```
.globalnav {
  background: var(--colors-grey-200);
  position:fixed;
}

.globalnav-floating-options {
  transform:scale(0);
  transform-origin:bottom left;
  opacity:0;
  margin:0;
  transition:transform,opacity,margin,visibility;
  transition-delay:130ms;
  transition-duration: 260ms;
  transition-timing-function: ease-in-out;
  background-color:var(--colors-grey-100);
  position:fixed;
  bottom:0;
  left:96px;
  z-index:8;
  padding:16px 8px;
  display:flex;
  flex-direction: column;
  align-items: center;
  border-radius: 10px;
  visibility:hidden;
}

.globalnav-floating-options.is-active {
  margin-bottom:16px;
  margin-left:16px;
  transform:scale(1);
  opacity:1;
  visibility:visible;
}

.globalnav-floating-option:not(:first-of-type) {
  margin-top:16px;
}

.globalnav-floating-options-backdrop {
  z-index:7;
  position:fixed;
  width:calc(100vw - 96px);
  height:100vh;
  background-color: #000;
```

```
    top:0;
    right:0;
    opacity:0;
    transition: opacity 260ms 130ms ease-in-out, visibility 260ms 130ms
    ease-in-out;
    visibility: hidden;
}

.globalnav-floating-options-backdrop.is-active {
    opacity:0.4;
    visibility: visible;
}

.application-container {
    width:calc(100vw - 96px);
    height:100vh;
    display:flex;
    flex-direction:row;
    margin-left:96px;
}

.application-content {
    width:100%;
    height:100%;
    padding: 16px;
}

.application-content .text-header-container {
    margin-bottom:64px;
}

.main {
    /*height:calc(100% - 79px - 64px);*/
    width:100%;
    display:flex;
    justify-content: center;
}

.main-content {
    width:100%;
    height:100%;
}

.main-content.thin {
    width:60%
}
```

```
.application-container {
  width:100vw;
  height:100vh;
  display:flex;
  flex-direction:row;
}

.application-content {
  width:100%;
  height:100%;
  padding: 16px;
}

.application-content .text-header-container {
  margin-bottom:64px;
  display:flex;
  flex-direction:column;
  align-items: center;
}

.application-content .text-header-container .text.one {
  margin-bottom:16px;
}

.application-content .header-option {
  position:relative;
  overflow:visible;
}

.application-content .header-option::after {
  content: "";
  background-color: var(--colors-grey-500);
  width: 70%;
  height: 4px;
  border-radius: 5px;
  position: absolute;
  bottom: -5px;
  right: -8px;
  opacity:1;
  transition-duration:200ms;
  transition-timing-function:ease-in-out;
  transition-property: background-color,width,height,bottom,rigt,opacity;
}

.application-content .header-option.active::after {
  background-color: var(--colors-secondary-dark);
}
```

```
.application-content .header-option:hover::after {
  background-color: var(--colors-grey-300);
  width: 100%;
  height: 100%;
  bottom: 0;
  right: 0;
  opacity: 0.2;
}

.application-content .header-option.active:hover::after {
  background-color: var(--colors-secondary);
}

.application-content .section-header-item:not(:last-child) {
  margin-bottom: 16px
}

.application-content .header-options {
  width: 50%;
  display: flex;
  justify-content: space-evenly
}

.application-content .header-option {
  cursor: pointer;
}

.application-content .header-option:active {
  opacity: .8;
}

.application-content .header-option.active {
  color: var(--colors-secondary);
}

.application-content .field-container {
  display: flex;
  flex-direction: column;
  align-items: center;
}

.application-content .field-container .field-submit {
  margin-top: 32px;
  align-self: center;
}

.application-content .field-container .field-options {
  width: 360px;
  display: flex;
  flex-direction: column;
}
```

```
.application-content .field-container .field-option:not(:last-child) {
  margin-bottom:8px
}

.application-content .field-container .field-option {
  display:flex;
  flex-direction: row;
  justify-content: space-between;
}

.application-content .field-container .field-option .field-input-container {
  display:flex;
  flex-direction: row;
}

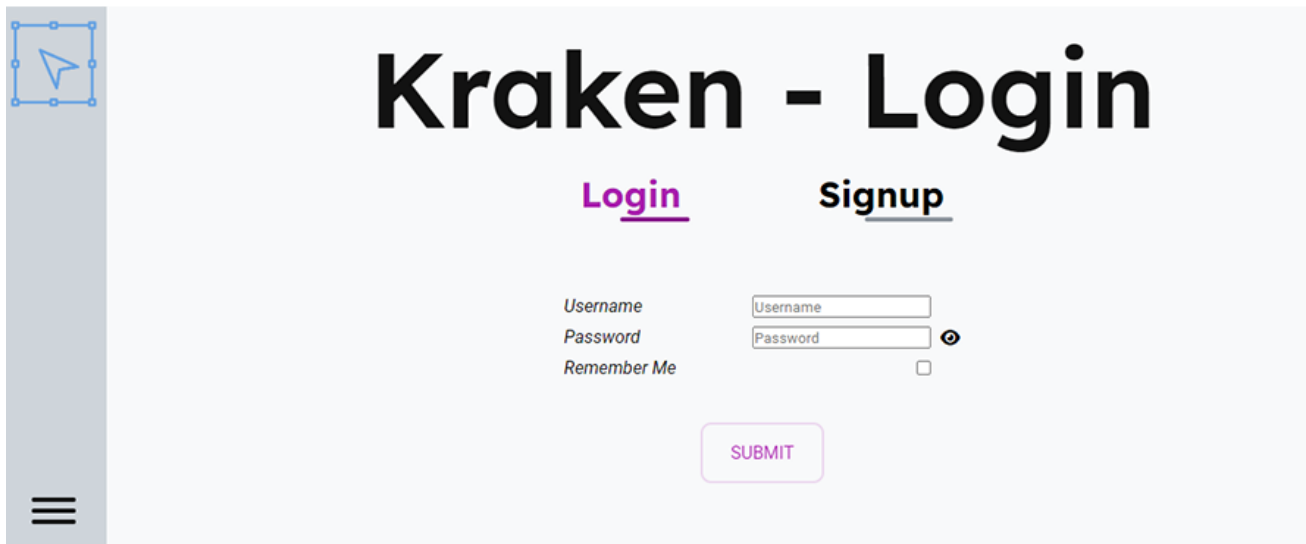
.application-content .field-container .field-option .field-input {
  font-family: var(--font-body);
}

.application-content .field-container .field-option .field-input-container
.eye-reveal,
.application-content .field-container .field-option .field-input-container
.eye-spacer {
  width:19px;
  height:19px;
  display:flex;
  justify-content: center;
  align-items: center;
  margin-left:8px;
}

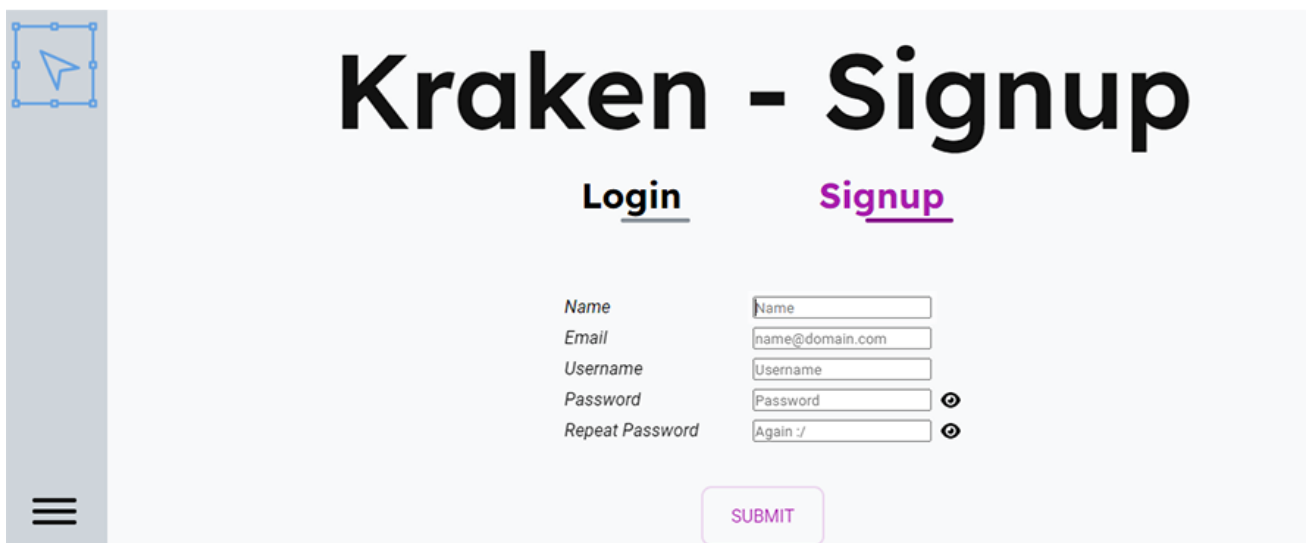
.application-content .field-container .field-option .field-input-container
.eye-reveal:active {
  color:var(--colors-secondary);
}

.application-content .field-container .field-warning {
  color:#e63832;
  margin-bottom:8px;
  max-width: 360px;
  text-align: center;
}
```


After running the website, the login and signup pages looked like this:



The login page features a light gray background. On the left is a vertical sidebar with a blue square icon containing a white cursor arrow and a hamburger menu icon at the bottom. The main content area has the title "Kraken - Login" in large black font. Below the title are two links: "Login" in purple and "Signup" in black. The form includes labels for "Username", "Password", and "Remember Me" on the left, and corresponding input fields on the right. The "Password" field has a toggle icon. A purple "SUBMIT" button is centered below the form.



The signup page has a similar layout to the login page. The title is "Kraken - Signup". The links are "Login" in black and "Signup" in purple. The form labels on the left are "Name", "Email", "Username", "Password", and "Repeat Password". The input fields on the right include "Name", "Email" (with a placeholder "name@domain.com"), "Username", "Password", and "Again :)", with toggle icons for the password fields. A purple "SUBMIT" button is at the bottom center.

Currently, when you click the submit button, it redirects to a page saying "Method not allowed", as the post functions have not been added to `__init__.py` yet.

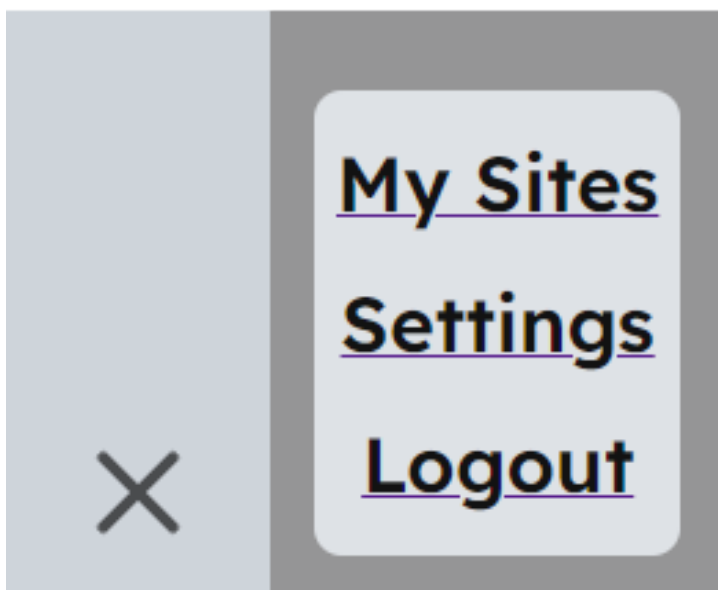
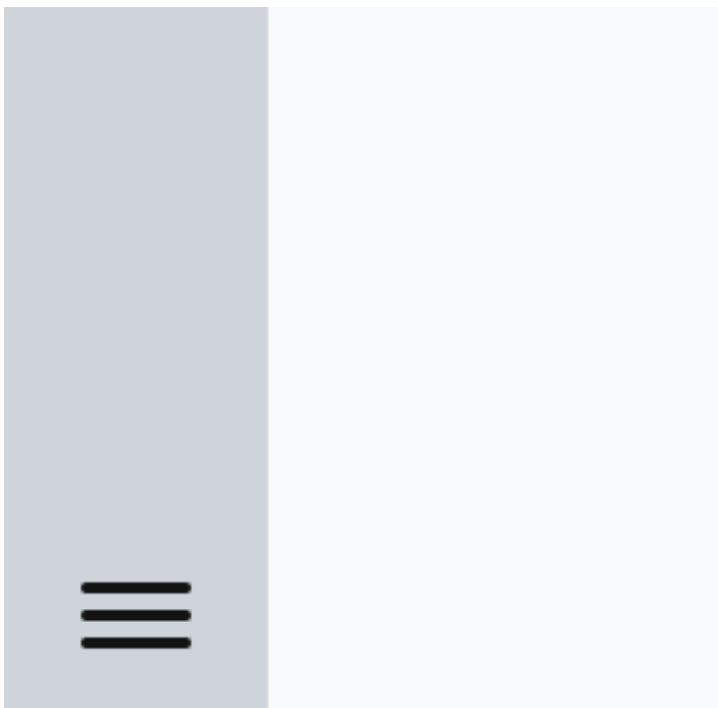
The next step was to set up the frontend programming for the login and signup pages, and the navigation bar. Starting with the navigation bar, I needed to set event listeners for the hamburger and background darkener `div`. These listeners would add or remove the `is-active` class to the hamburger, options list, and background, so that the website renders correctly. The file is already imported into `base.html` via `<script src="{{url_for('static', filename='js/globalnav-floating-options.js')}}">`.

`/static/js/globalnav-floating-options.js`

```
document.getElementById("globalnav-hamburger").addEventListener("click",()=>{
    document.getElementById("globalnav-hamburger").classList.toggle('is-active');
    document.querySelectorAll(".globalnav-floating-options").forEach((e)=>{
        e.classList.toggle("is-active");
    });
    document.querySelectorAll(".globalnav-floating-options-backdrop").forEach((e)=>{
        e.classList.toggle("is-active");
    });
});
```

```
document.querySelectorAll(".globalnav-floating-options-backdrop").forEach((e)=>{
  e.addEventListener("click",()=>{
    document.getElementById("globalnav-hamburger").classList.remove(
      'is-active');
    document.querySelectorAll(".globalnav-floating-options").forEach(
      (e)=>{e.classList.remove("is-active")});
    document.querySelectorAll(".globalnav-floating-options-backdrop").forEach(
      (e)=>{e.classList.remove("is-active")});
  })
});
```

After running the website, the floating navigation options looked like this:



There is validation of the inputs both client-side and server-side, so I needed to implement that into the JavaScript for the login and signup pages. I also need to add code so that when the all-seeing eye is pressed, the password field is miraculously revealed. For the validation, I will use the `verifyField` function that I have written in pseudocode (which will also be used in the Python backend). Because of this, there will be three files, `auth.js`, `login.js` and `signup.js` so that there is less duplicated code. There will also be a function `isEmail`, which uses a regex validation check to make sure that the email given is in a valid format.

`/static/js/auth.js`

```
// Function called for each field to make sure it is in the correct format, takes
// a few arguments as flags for what makes it valid
function verifyField(field,fieldName,mustHaveChar=true,minLen=3,
    canHaveSpace=false,canHaveSpecialChar=true,isPassword=false) {
    // List of special characters for the canHaveSpecialChar flag
    specialChar="%&{}\\<>?*?/$!'\"':@+`|="

    // Make sure that the input given is a string
    if (typeof field !== "string") {throw new Error("HEY! that's not a string?")}

    // Check through all the flags given and throw an appropriate error message
    // if input is invalid
    if (field.length==0 && mustHaveChar) {return `${fieldName} is not filled out.`}
    if (field.length<minLen) {
        return `${fieldName} must be greater than ${minLen-1} characters.`
    }
    if (!canHaveSpace && field.includes(" "))
        {return `${fieldName} cannot contain spaces.`}
    if (!canHaveSpecialChar) {
        // Iterate through each character in specialChar to see if its in the input
        // I didn't use regex for this as I wanted to be able to tell the user which
        // character wasn't allowed
        var char;
        for (var i=0;i<specialChar.length;i++) {
            char=specialChar[i]
            if (field.includes(char)) {
                return `${fieldName} cannot contain '${char}'`
            }
        }
    }
}

// If the given input is a password
if (isPassword) {
    // If it doesnt match the given regular expression for password checks
    if (!field.match(/(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])/
        /(?=.*?[#?!@$%^&*~_%&{}\\<>?*?/$!'\"':@+`|=]).{8,}/)/) {
        return `${fieldName} must contain at least 1 of each: uppercase character,
        lowercase character, number, and special character`
    }
}
```

```

// Regex pattern breakdown
// (?=.*?[A-Z]) = contains an uppercase character
// (?=.*?[a-z]) = contains a lowercase character
// (?=.*?[0-9]) = contains a digit
// (?=.*?[\#?!@$%^&*-_&{}\\<>*\$!\'\"':@+`|=]) = contains a special character
// .{8,} = has a minimum length of 8 and no upper limit

return ""
}

// Initialise the code for the all seeing eyes to enable viewing the password
function initAllSeeingEye(element, reveal) {
  // Add onclick event to given element (the eye element)
  element.addEventListener("click", e=> {
    // toggle input type of given input between password and text
    reveal.setAttribute('type', reveal.getAttribute('type') === 'password' ?
      'text' : 'password');
    // toggle the fa-eye-slash class for the eye (this sets the icon displayed)
    element.classList.toggle('fa-eye-slash');
  })
}

// Function takes a string and returns a boolean determining whether it is in a
// valid email format, using regex
function isEmail(email) {
  return email.match(/^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*(\\.\\w{2,3})+$/);
}

// fetch warning element and disable submit bottom
warningSpan = document.querySelector(".field-container .field-warning")
document.querySelector(".field-submit").disabled = true

```

/static/js/login.js

```

// function called when an input is changed, to check whether all inputs are valid
function verifyAllFields() {
  if (fields["Username"].value.length < 1) {
    warningSpan.innerText = "Username is not filled out"
    document.querySelector(".field-submit").disabled = true
    return
  }

  if (fields["Password"].value.length < 1) {
    warningSpan.innerText = "Password is not filled out"
    document.querySelector(".field-submit").disabled = true
    return
  }
}

```

```

warningSpan.innerText = ""
document.querySelector(".field-submit").disabled = false
}

// Dictionary of all fields in the form
fields={
  "Username":document.querySelector(".field-option-username .field-input"),
  "Password":document.querySelector(".field-option-password .field-input")
}

initAllSeeingEye(document.querySelector(".field-option-password .eye-reveal i"),
document.querySelector(".field-option-password .field-input"))
document.querySelectorAll(".field-input").forEach(field=>
  {field.addEventListener("change",verifyAllFields)})

```

/static/js/signup.js

```

// function called when an input is changed, to check whether all inputs are valid
function verifyAllFields() {
  verifyOutput=verifyField(fields["Name"].value,"Name",true,3,true,false)

  if (verifyOutput.length > 0) {
    warningSpan.innerText = verifyOutput
    document.querySelector(".field-submit").disabled = true
    return
  }

  verifyOutput=verifyField(fields["Email"].value,"Email",true,0)

  if (verifyOutput.length > 0) {
    warningSpan.innerText = verifyOutput
    document.querySelector(".field-submit").disabled = true
    return
  }

  // check for email in the correct format
  if (!isEmail(fields["Email"].value)) {
    warningSpan.innerText = "Email is not a valid email address"
    document.querySelector(".field-submit").disabled = true
    return
  }

  verifyOutput=verifyField(fields["Username"].value,"Username",true,3,false,false)

  if (verifyOutput.length > 0) {
    warningSpan.innerText = verifyOutput
    document.querySelector(".field-submit").disabled = true
    return
  }
}

```

```

verifyOutput=verifyField(fields["Password"].value,"Password",true,8,false,true,
true)

if (verifyOutput.length > 0) {
    warningSpan.innerText = verifyOutput
    document.querySelector(".field-submit").disabled = true
    return
}

// Make sure passwords match
if (fields["Password"].value!=fields["Repeat Password"].value) {
    warningSpan.innerText = "Passwords do not match"
    document.querySelector(".field-submit").disabled = true
    return
}

// If no errors are called, then enable the button and clear the warning message
warningSpan.innerText = ""
document.querySelector(".field-submit").disabled = false
}

// Dictionary of all fields in the form
fields={
    "Name":document.querySelector(".field-option-name .field-input"),
    "Email":document.querySelector(".field-option-email .field-input"),
    "Username":document.querySelector(".field-option-username .field-input"),
    "Password":document.querySelector(".field-option-password .field-input"),
    "Repeat Password":document.querySelector(
        ".field-option-password-repeat .field-input")
}

initAllSeeingEye(
    document.querySelector(".field-option-password .eye-reveal i"),
    document.querySelector(".field-option-password .field-input"))

initAllSeeingEye(
    document.querySelector(".field-option-password-repeat .eye-reveal i"),
    document.querySelector(".field-option-password-repeat .field-input"))

document.querySelectorAll(".field-input").forEach(field=>
    {field.addEventListener("change",verifyAllFields)})

```

This is an image of the all-seeing eye in action:

Username

Password

Remember Me ☒

Username

Password

Remember Me ☒

After implementing the client-side validation, I then tested each field with a variety of different inputs to make sure the validation code was functioning properly

Field	Test Data	Reason	Expected Outcome	Actual Outcome	Pass/Fail
Name	Aaaaa	Check it says valid	Valid	Valid	Pass
Name	Aaa Bbb	Check it allows spaces	Valid	Valid	Pass
Name	Aaa%bbb	Check that it doesn't allow special chars	Invalid	Valid	Fail
Name	null	Check that it requires name	Invalid	Invalid	Pass
Email	a@b.cc	Check it says valid	Valid	Valid	Pass
Email	ab12@f42.x7	Check it says valid	Valid	Valid	Pass
Email	@b.cc	Check it recognises the area before @	Invalid	Invalid	Pass
Email	a@.cc	Check it recognises the area after @	Invalid	Invalid	Pass

Field	Test Data	Reason	Expected Outcome	Actual Outcome	Pass/Fail
Email	a@b.c	Check it requires a top level domain longer than 1	Invalid	Invalid	Pass
Email	a@b.cdef	Check it requires a top level domain shorter than 4	Invalid	Invalid	Pass
Email	a b@ccc.uk	Check it doesn't allow spaces	Invalid	Invalid	Pass
Email	null	Check that it requires email	Invalid	Invalid	Pass
Username	Aaa	Check it says valid	Valid	Valid	Pass
Username	Aaa bbb	Check it doesn't allow spaces	Invalid	Invalid	Pass
Username	A-._+c	Check it allows special characters not given in the list	Valid	Valid	Pass
Username	%&{}}\<>*/\$!'\"':@+	Check it doesn't allow these special characters	Invalid	Valid	Fail
Username	null	Check it requires username	Invalid	Invalid	Pass

Field	Test Data	Reason	Expected Outcome	Actual Outcome	Pass/Fail
Password	aaa	Check it has a minimum length of 8	Invalid	Invalid	Pass
Password	Aaaaaa_1	Data to work off for next tests	Valid	Valid	Pass
Password	aaaaaa_1	Check it requires an uppercase character	Invalid	Invalid	Pass
Password	AAAAAA_1	Check it requires a lowercase character	Invalid	Invalid	Pass
Password	Aaaaaa_a	Check it requires a number	Invalid	Invalid	Pass
Password	Aaaaaaa1	Check it requires a special character	Invalid	Invalid	Pass
Password	null	Check it requires password	Invalid	Invalid	Pass
Passwords	Aaaaaa_1 in both fields	Check both fields have to match	Valid	Valid	Pass
Passwords	Aaaaaa_1 in one field, ABCDEF in the second	Check both fields have to match	Invalid	Invalid	Pass

As you can see from the table, two of the validation checks failed. These are listed here:

Field	Test Data	Reason	Expected Outcome	Actual Outcome	Pass/Fail
Name	Aaa%bbb	Check that it doesn't allow special chars	Invalid	Valid	Fail
Username	%&{}\\ <>*?/\$!'\"':@+	Check it doesn't allow these special characters	Invalid	Valid	Fail

These were both to do with verifying special characters, and, when I revisited the code, I saw that the error was when I tried to iterate through the characters in a string like you do can do in python. Hence, I modified the line `for (var char in specialChar)` to function properly, and both tests came out as invalid.

Field	Test Data	Reason	Expected Outcome	Actual Outcome	Pass/Fail
Name	Aaa%bbb	Check that it doesn't allow special chars	Invalid	Invalid	Pass
Username	%&{}\\ <>*?/\$!'\"':@+	Check it doesn't allow these special characters	Invalid	Invalid	Pass

changes to `/static/js/auth.js`

```
function verifyField(...)
```

```
    if (!canHaveSpecialChar) {
        var char;
        for (var i=0;i<specialChar.length;i++) {
            char=specialChar[i]
            if (field.includes(char)) {
                return `${fieldName} cannot contain '${char}'`
            }
        }
    }
}
```

I then implemented the server-side validation, which re-checks all of the validation performed client-side (using the same `verifyField` function), to ensure that the inputs are valid and weren't tampered with client-side. Flask retrieves the values of the form via the `requests` import. The database checking has not yet been implemented at this point, as I wanted to get the login and signup pages fully completed before creating the database. This also involved adding code to the templates to interpret the flashed error message.

changes to `__init__.py`

```
from flask import Flask, render_template, redirect, flash, request
```

```
# Flask is the application object
# render_template converts a Jinja file to html
# redirect redirects the website to another root function
# flash sends messages to the client
# request allows the code to handle form inputs
```

```
# Login post route
@self.app.route("/login/", methods=["post"])
def auth_login_post() -> Response:
    # Get the filled-in items from the login form
    username = request.form.get("username")
    password = request.form.get("password")
    remember = True if request.form.get('remember') else False

    # TODO: get the user from the database. if there's no user it returns none
    if False:
        # Flashes true to signify an error, the error message, the username given,
        # and the remember flag given
        flash([True, 'Please check your login details and try again.'])
        return redirect(url_for('auth_login'))

    # TODO: check for correct password
    if False:
        flash([True, 'Please check your login details and try again.'])
        return redirect(url_for('auth_login'))

    # TODO: login user
    return redirect(url_for("main_home"))
```

```
# Signup post route
@self.app.route("/signup/", methods=["post"])
def auth_signup_post() -> Response:
    # Get the filled-in items from the signup form
    name=request.form.get("name")
    email=request.form.get("email")
    username=request.form.get("username")
    password1=request.form.get("password")
```

```
password2=request.form.get("password-repeat")

# the verifyField function returns either an empty string if the field meets the
# requirements defined by the arguments, or an error message. So, if
# len(verifyOutput) > 0, that means that the field is invalid

# Verify the name input and return an error message if invalid
verifyOutput=self.verifyField(name,"Name",canHaveSpace=True,
canHaveSpecialChar=True)

if len(verifyOutput) > 0:
    # Flashes true to signify an error, and the error message
    flash([True,verifyOutput])
    return redirect(url_for("auth_signup"))

# Verify the email input and return an error message if invalid
verifyOutput=self.verifyField(email,"Email",minLen=0,canHaveSpace=False,
canHaveSpecialChar=True)

if len(verifyOutput) > 0:
    # Flash an error message
    flash([True,verifyOutput])
    return redirect(url_for("auth_signup"))

# Verify the username input and return an error message if invalid
verifyOutput=self.verifyField(username,"Username",canHaveSpecialChar=False)

if len(verifyOutput) > 0:
    # Flash an error message
    flash([True,verifyOutput])
    return redirect(url_for("auth_signup"))

# Verify the password input and return an error message if invalid
verifyOutput=self.verifyField(password1,"Password",minLen=8)

if len(verifyOutput) > 0:
    # Flash an error message
    flash([True,verifyOutput])
    return redirect(url_for("auth_signup"))

# Return an error message if the passwords do not match
if password1!=password2:
    # Flash an error message
    flash([True,"Passwords do not match"])
    return redirect(url_for("auth_signup"))

# TODO: check whether this email already has an account

if False:
    flash([True,"That email is already in use"])
    return redirect(url_for("auth_signup"))
```

```
# TODO: check whether this username already exists
```

```
if False:
    flash([True,"That username is already in use"])
    return redirect(url_for("auth_signup"))
```

```
# TODO: create a new user in the database
```

```
return redirect(url_for("auth_login"))
```

```
def verifyField(self,field:str,fieldName:str,mustHaveChar:bool=True,minLen:int=3,
    canHaveSpace:bool=False,canHaveSpecialChar:bool=True) -> str:
    # List of special characters for the canHaveSpecialChar flag
    specialChar="%&{}\\<>*?/$!'\"':@+`|="

    # Make sure that the input given is a string, raise an exception if its not
    if type(field) != str: Exception("HEY! that's not a string?")

    # Check through all the flags given and throw an appropriate error message if
    # input is invalid
    if len(field) == 0 and mustHaveChar:
        return f"{fieldName} is not filled out."
    if len(field) < minLen:
        return f"{fieldName} must be greater than {minLen-1} characters."
    if not canHaveSpace and " " in field:
        return f"{fieldName} cannot contain spaces."
    if not canHaveSpecialChar:
        for char in specialChar:
            if char in field:
                return f"{fieldName} cannot contain '{char}'"

    return "" # Return an empty string if the input is valid
```

changes to /templates/login.html and /templates/signup.html

```
{% set messages = get_flashed_messages()[0] %}
```

```
<span class="field-warning text italic">
    {% if messages[0] %}
        {{ messages[1] }}
    {% endif %}
</span>
```

At the suggestion of one of the stakeholders, I also added a feature so that when you submit the form, and it throws an error, the form values are carried over so that the user doesn't have to fill them out again. I implemented this using the `flash` function, flashing a list containing the inputs that they had given. To make sure that this didn't cause any issues when opening the page for the first time, the `auth_login` and `auth_signup` functions also flash a list (`[False, "", "", "", ""]`) to prevent any index errors. In the HTML files, an extra variable is added to the Jinja list of field items, to define what value the input should default to.

changes to `__init__.py`

```
# Login page route
def auth_login() -> str:
    # Flash an empty list of values to stop errors in the Jinja code
    flash([False, "", "", "", ""])
    return render_template("login.html")
```

```
# Signup page route
def auth_signup() -> str:
    # Flash an empty list of values to stop errors in the Jinja code
    flash([False, "", "", "", ""])
    return render_template("signup.html")
```

```
# Login post route
def auth_login_post() -> Response:
```

```
    # Flashes true to signify an error, the error message, the username given, and
    # the remember flag given
    flash([True, 'Please check your login details and try again.', username, remember])
    return redirect(url_for('auth_login'))
```

```
# Signup post route
def auth_signup_post() -> Response:
```

```
    if len(verifyOutput) > 0:
        # Flashes true to signify an error, the error message, the name given
        # (removed due to error), the email given, and the username given
        flash([True, verifyOutput, "", email, username])
        return redirect(url_for("auth_signup"))
```

```
# Flash an error message and the filled in values
flash([True,verifyOutput,name,"",username])
```

```
flash([True,verifyOutput,name,email,""])
```

```
flash([True,verifyOutput,name,email,username])
```

```
# Return an error message if the passwords do not match
if password1!=password2:
    # Flash an error message and the filled in values
    flash([True,"Passwords do not match",name,email,username])
    return redirect(url_for("auth_signup"))
```

changes to /templates/login.html

```
{% set formItems = [
["Username","Username","text","username",false,messages[2]],
["Password","Password","password","password",true,""]
]
%}
```

```
<input class="field-input" placeholder="{{item[1]}}" type="{{item[2]}}"
name="{{item[3]}}">
```

changes to /templates/signup.html

```
{% set formItems = [
["Name","Name","text","name",false,messages[2]],
["Email","name@domain.com","email","email",false,messages[3]],
["Username","Username","text","username",false,messages[4]],
["Password","Password","password","password",true,""],
["Repeat Password","Again :/", "password","password","password-repeat",""]
]
%}
```

```
<input class="field-input" placeholder="{{item[1]}}" type="{{item[2]}}"
name="{{item[3]}}" value="{{item[5]}}">
```

To finish the design of the login and signup pages, I added a jinja variable that defines the colour of the logo in the sidebar, and added another variable that defines whether or not the hamburger and subsequent option modal is visible or not. This is because, although it will be required for other sites (such as the homepage), the navigation bar is not necessary here as all of the links in the navigation bar will redirect to `/login` as the user is not signed in.

changes to `/templates/base.html`

```
<!-- navbarLogoColor is a jinja variable that is defined in files that extend
from this one. It defines what colour the logo should be - primary, secondary,
or gradient -->
<link rel="apple-touch-icon" sizes="512x512" href="{{url_for('static',
filename='img/icon/512-512/kraken-icon-png- '+navbarLogoColor+'-128-128.png')}}">
<link rel="icon" type="image/png" sizes="128x128" href="{{url_for('static',
filename='img/icon/128-128/kraken-icon-png- '+navbarLogoColor+'-128-128.png')}}">
```

```

```

```
{% if navbarOptionsEnabled %}
<ul class="globalnav-list">
  <li class="globalnav-item one fake" role="button"></li>
  <li class="globalnav-item two" role="button">
    <div class="hamburger hamburger--collapse js-hamburger"
      id="globalnav-hamburger">
      <div class="hamburger-box">
        <div class="hamburger-inner"></div>
      </div>
    </div>
  </li>
</ul>
{% endif %}
```

```
{% if navbarOptionsEnabled %}

<!-- Floating option modal for the navbar, which is opened and closed via the
hamburger in the navigation bar -->

<div class="globalnav-floating-options">
  <a class="globalnav-floating-option one" href="">
    <span class="globalnav-floating-option-content text header small dark">
      My Sites
    </span>
  </a>
```



```

<a class="globalnav-floating-option two" href="">
  <span class="globalnav-floating-option-content text header small dark">
    Settings
  </span>
</a>

<a class="globalnav-floating-option three" href="">
  <span class="globalnav-floating-option-content text header small dark">
    Logout
  </span>
</a>
</div>

<!-- Backdrop behind nav bar modal to apply a darkness filter behind the modal -->

<div class="globalnav-floating-options-backdrop"></div>

<script src="{{url_for('static', filename='js/globalnav-floating-options.js')}}">
</script>

{% endif %}

```

changes to `/templates/login.html` and `/templates/signup.html`

```

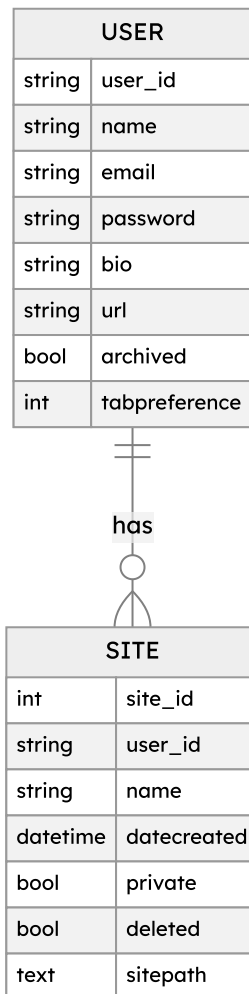
{% set navbarLogoColor = "secondary" %}
{% set navbarOptionsEnabled = False %}

```

All of the logo images now have the `navbarLogoColor` variable to define which image to fetch. The hamburger and navigation options are now surrounded in `{% if navbarOptionsEnabled %}`. Both of these variables will be defined in files that extend from this file, such as `login.html`. The script import for `/js/globalnav-floating-options.js` has also been moved into the if block to remove unnecessary imports.

Stage 2 - Creating and implementing the database

After completing the login and signup pages, I created the database, referring to the entity relationship diagram that I had outlined when planning. The two entities are `User` and `Site`, where `user_id` is a foreign key in `Site` to allow them to link together via a one to many relationship. The `User` entity contains some settings information, such as `bio`, `url`, and `tabpreference`, which will be able to be changed in the settings page, that are implemented now to make development down the line easier.



The database is managed by the `flask_sqlalchemy.SQLAlchemy` object. In `__init.py`, the object is created (with the variable name `databaseObject`) when the file is run so that `models.py`, the new file that I created which contains the entity classes, can import it. After adding the `databaseObject` object to the class, it imports the two classes from `models.py`, so that the database can interact with them. I also moved all of the flask setup into the function `initFlask` to make the code clearer.

changes to __init__.py

```
from flask_sqlalchemy import SQLAlchemy

# SQLAlchemy manages the SQL database

# Create the database object
databaseObject = SQLAlchemy()

# The main class of the application
class Kraken():
    # Global reference to database object
    global databaseObject

    def __init__(self, host: str, port: int) -> None:
        # Assign the database object to the local db reference
        self.db = databaseObject

        # Initialise the flask application
        self.initFlask()

        # Initialise the SQL database
        self.db.init_app(self.app)

        # Import the User and Site entities from models.py
        from models import User, Site
        self.User = User
        self.Site = Site
```

```
def initFlask(self) -> None:
    # Create the Flask application and set a secret key
    self.app = Flask(__name__)
    self.app.config["SECRET_KEY"] = "secret-key-goes-here"
    # Set the database file URL to /db.sqlite in the root directory
    self.app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///db.sqlite"
    self.app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    # Initialise the website pages
    self.initPages()
```

models.py

```
from flask_login import UserMixin

# Import the SQLAlchemy database object from the main class
from __init__ import databaseObject as db

# User class to store the user's information in the database
class User(UserMixin, db.Model):
```

```

# Set the name of the table in the database to "user"
__tablename__="user"

# Define the columns in the table
# Primary Key user_id as a string
user_id = db.Column( db.String, primary_key=True )
# Name as a string
name = db.Column( db.String )
# Email as a string, cannot be null and must be unique
email = db.Column( db.String, nullable=False, unique=True )
# Password as a string, cannot be null
password = db.Column( db.String, nullable=False )
# Bio as a text field
bio = db.Column( db.Text )
# URL as a text field
url = db.Column( db.Text )
# Archived flag as a boolean, cannot be null (default False)
archived = db.Column( db.Boolean, nullable=False )
# Tab preference as a number, cannot be null (default four)
tabpreference = db.Column( db.Float, nullable=False )

# Setup the foreign key relationship
sites = db.relationship("Site")

# Function to return the primary key
def get_id(self) -> str: return self.user_id

# Site class to store the User's sites in the database
class Site(db.Model):
    # Set the name of the table in the database to "site"
    __tablename__="site"

    # Define the columns of the table
    # Primary Key site_id as an integer
    site_id = db.Column( db.Integer, primary_key=True )
    # Foreign Key user_id as a string, referring to user_id in the User table
    user_id = db.Column( db.String, db.ForeignKey("user.user_id") )
    # Name as a string, cannot be null
    name = db.Column( db.String, nullable=False )
    # Datecreated as a datetime format
    datecreated = db.Column( db.DateTime )
    # Private flag as a boolean, cannot be null
    private = db.Column( db.Boolean, nullable=False )
    # Deleted flag as a boolean, cannot be null (default False)
    deleted = db.Column( db.Boolean, nullable=False )
    # Sitepath as a text field
    sitePath = db.Column( db.Text )

    # Function to return the primary key
    def get_id(self) -> int: return self.site_id

```

I then ran the following commands in an online SQL editor to create the database, and saved it as `db.sqlite` in the root directory, so that SQLAlchemy could use it. I also created `db.sqlite.bak`, so I could have an empty version of the database to use throughout development and testing.

db.sqlite commands

```
CREATE TABLE user (  
    user_id TEXT PRIMARY KEY,  
    name TEXT,  
    email TEXT NOT NULL,  
    password TEXT NOT NULL,  
    bio TEXT,  
    url TEXT,  
    archived BOOLEAN NOT NULL,  
    tabpreference INT NOT NULL  
)  
  
CREATE TABLE site (  
    site_id INTEGER PRIMARY KEY AUTOINCREMENT,  
    name TEXT NOT NULL,  
    datecreated DATE,  
    private BOOLEAN NOT NULL,  
    deleted BOOLEAN NOT NULL,  
    user_id TEXT,  
    sitePath TEXT,  
    CONSTRAINT fk_user_id,  
    FOREIGN KEY (user_id) REFERENCES user(user_id)  
)
```

I now added the authentication code to `auth_login_post` and `auth_signup_post` so that they could query the new database. This also included importing the `werkzeug.security` module to implement the hashing of passwords, and the `flask_login` module to implement the logging in system

changes to `__init__.py`

```
from flask_login import LoginManager, login_user  
  
# LoginManager is the object that manages signed in users  
# login_user logs in a give user  
  
from werkzeug.security import generate_password_hash, check_password_hash  
  
# generate_password_hash and check_password_hash are used when generating  
# and authenticating users
```

```
def __init__(self, host: str, port: int) -> None:
```

```
# Initialise the login manager
self.loginManager=LoginManager()
# set which function routes to the login page
self.loginManager.login_view="auth_login"
self.loginManager.init_app(self.app)

# Fetches a row from the User table in the database
@self.loginManager.user_loader
def loadUser(user_id): return self.User.query.get(user_id)
```

```
def auth_login_post() -> Response:
```

```
# Fetch the user from the database. if there's no user it returns none
user = self.User.query.filter_by(user_id=username).first()

if user is None:
    # Flashes true to signify an error, the error message, the username
    # given, and the remember flag given
    flash([True, 'Please check your login details and try again.',
            username, remember])
    return redirect(url_for('auth_login'))

# TODO: check for correct password
if not check_password_hash(user.password, password):
    flash([True, 'Please check your login details and try again.',
            username, remember])
    return redirect(url_for('auth_login'))

# Log in the user and redirect them to the homepage
login_user(user, remember=remember)
return redirect(url_for("main_home"))
```

```
def auth_signup_post() -> Response:
```

```
# Check whether this email already has an account
if self.User.query.filter_by(email=email).first():
    flash([True, "That email is already in use", name, "", username])
    return redirect(url_for("auth_signup"))

# Check whether this username already exists
if self.User.query.filter_by(user_id=username).first():
```

```
flash([True,"That username is already in use",name,email,""])
return redirect(url_for("auth_signup"))
```

Next, I created the function `createUser`, that would be called when all of the validation in `auth_signup_post` is complete. It takes the variables `username`, `email`, `name`, and `password`. The function creates a new entry in the database, and creates the users file structure in the server-side storage, making use of the `generateFolderStructure` function.

changes to `__init__.py`

```
def __init__(host:str,port:int) -> None:
    import os
    self.os=os
```

```
def auth_signup_post() -> Response:
```

```
# create a new user in the database and send to the login page
self.createUser(username,email,name,password1)
return redirect(url_for("auth_login"))
```

```
def createUser(self,u:str,e:str,n:str,p:str) -> None:

    # Create a new User object using the variables given
    newUser = self.User(
        user_id=u,
        name=n,
        email=e,
        password=generate_password_hash(p,method='sha256'),
        bio="",
        url="",
        archived=False,
        tabpreference=4,
    )

    # Server-side folder generation

    prefix="static/data/userData/"

    # List of all folders to create
    folderStructure=[
        self.os.path.abspath(f"{prefix}{u}"),
        self.os.path.abspath(f"{prefix}{u}/sites/")
    ]

    self.generateFolderStructure(folderStructure)
```

```

# Create new user and commit to database
self.db.session.add(newUser)
self.db.session.commit()

def generateFolderStructure(self, folders: list) -> None:
    # Iterate through given list of directories
    for folder in folders:
        # Ignore if directory already exists
        if self.os.path.isdir(folder): continue
        # Create the directory
        try: self.os.makedirs(folder)
        except OSError as e: raise e

```

I then modified the `auth_signup_post` function so that it logs you in as soon as the user creates their account.

changes to `__init__.py`

```
def auth_signup_post() -> Response:
```

```

# create a new user in the database and server-side storage
self.createUser(username, email, name, password1)

# Log in the new user and redirect them to the homepage
login_user(self.User.query.filter_by(user_id=username).first(),
remember=False)
return redirect(url_for("auth_login"))

```

Finally, I created the `auth_logout` function that logs out a user, and added it to the navigation bar URL for later iterations. It also has the `login_required` decorator, that will be used more later

changes to `__init__.py`

```
from flask_login import LoginManager, login_user, login_required, logout_user
```

```

@self.app.route("/logout/")
@self.app.route("/account/logout/")
@login_required
def auth_logout() -> Response:
    logout_user()
    return redirect(url_for("auth_login"))

```


Stage 3 - Homepage and Settings

This stage consisted of creating the frontend HTML for the user, including the settings pages. Although they lack much tangible functionality, they create the basis for future developers to implement. I decided not to complete all of the functionality due to time constraints, and lack of the features being required in the brief.

First, I created the home page. There are two template files `home-nosite.html` and `home-sites.html`, which are called by Flask based on whether the user has any sites stored in the database.

Due to there not yet being a way of creating sites, the line

```
flash([[x.user_id,x.name,x.private] for x in self.Site.query.filter_by(user_id=current_user.user_id).all()])
```

 would never work.

Instead, for testing and design purposes, I used the test data `flash(["user1", "Site 1", True], ["user1", "Epic Webpage", False])` to see how the code handles a user's sites, and I used `and False` in the if block to simulate the user having no sites.

changes to `__init__.py`

```
# Index route, redirects to auth_login, which will redirect to main_home if
# logged in
@self.app.route("/")
def main_index() -> Response: return redirect(url_for("auth_login"))

# Home Page Route
@self.app.route("/home/")
@login_required # User must be logged in to access this page
def main_home() -> str:
    # check to see if user has any sites
    if len(self.Site.query.filter_by(user_id=current_user.user_id).all()) > 0:
        # and False:

        # For each site, flash its userid, name, and privacy flag
        # This will not yet do anything as there are no sites in the server

        # flash([[x.user_id,x.name,x.private] for x in self.Site.query.filter_by(
        # user_id=current_user.user_id).all()])

        # For testing and design purposes, the flash command above was commented out
        # and replaced with this command
        flash(["user1", "Site 1", True], ["user1", "Epic Webpage", False])

    return render_template("home-sites.html")
return render_template("home-nosite.html")
```

/templates/home-nosite.html

This file uses `current_user.name` to display the username of the logged-in user. The object `current_user` is a Flask variable that is read from and inserted into the page during the `render_template` function. It is also used in `__init__.py` for commands such as `filter_by(user_id=current_user.user_id)`.

```
{% extends "base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% block content %}

<link href="{{url_for('static', filename='css/home.css')}}" rel="stylesheet"
type="text/css" />

<div class="application-content">

    <div class="text-header-container">
        <h2 class="text header large dark one">Welcome, {{current_user.name}}</h2>
    </div>

    <div class="empty-container">
        <div class="empty-image"></div>
        <div class="empty-text-container">

            <h4 class="text header dark one">Looks Pretty Empty Here...</h4>

            <a class="text header two link primary" href="{{url_for('site_create')}}">
                Maybe you should create a new site?
            </a>

        </div>
    </div>

</div>

{% endblock %}
```

/templates/home-sites.html

```
{% extends "base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set sites = get_flashed_messages()[0] %}

{% block content %}
```

```

<link href="{{url_for('static', filename='css/home.css')}}" rel="stylesheet"
type="text/css" />

<div class="application-content">

    <div class="text-header-container">
        <h2 class="text header large dark one">Welcome, {{current_user.name}}</h2>
    </div>

    <div class="site-divs-container" style="display:flex;flex-wrap:wrap">

        {% for site in sites %}

            <a class="site-div link notformatted" href="{{url_for('site_edit_home',
                name=site[0],site=site[1])}}" style="background-color:
                {%if site[3]%} {{site[3]+'bb'}} {%else%} var(--colors-primary) {%endif%}">

                {% if site[2] %}
                    <div class="site-div-private-watermark" style="position: absolute;
                        opacity: 0.5;font-size: 136px;right: 16px;top: 16px;">

                        <i class="faicon fa-regular fa-lock"></i>
                    </div>
                {% else %}
                    <div class="site-div-public-watermark" style="position: absolute;
                        opacity: 0.5;font-size: 136px;right: 16px;top: 16px;">

                        <i class="faicon fa-regular fa-book-bookmark"></i>
                    </div>
                {% endif %}

                <h5 class="site-div-title text large one">{{ site[1] }}</h5>
            </a>

            {% endfor %}

            <a class="site-div link notformatted" style="background-color:
                var(--colors-primary-light);display:flex;align-items:center;
                justify-content:center" href="{{url_for('site_create')}}">

                <h5 class="site-div-title text header jumbo small one center">
                    Create New Site
                </h5>
            </a>

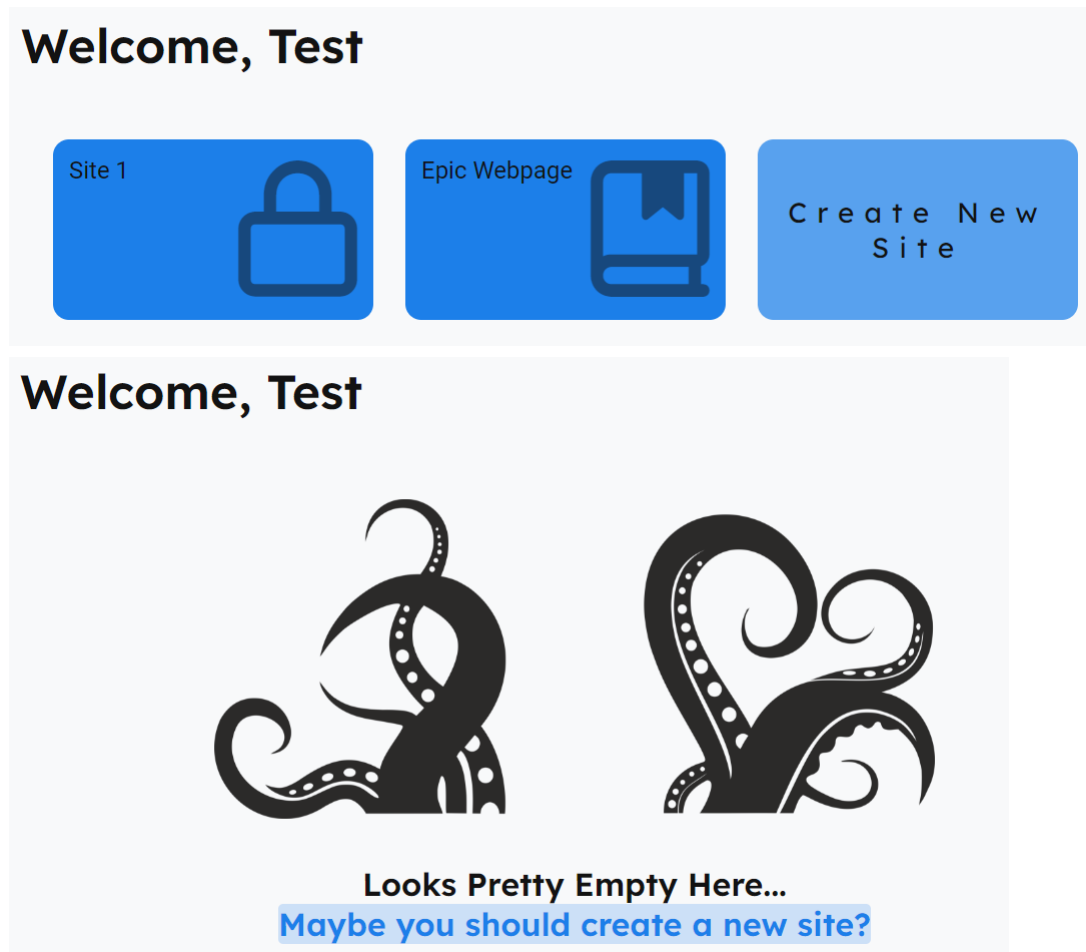
        </div>

    </div>

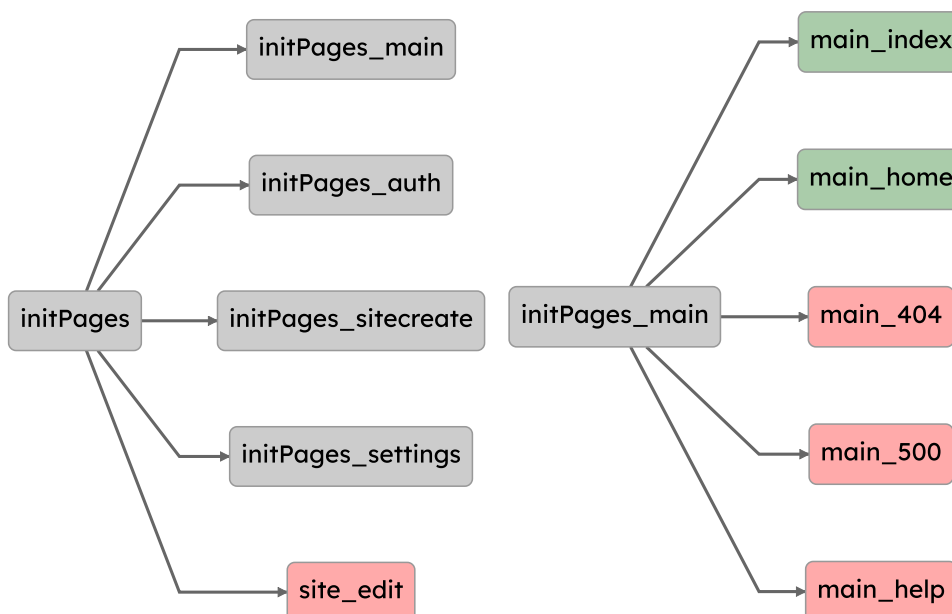
    {% endblock %}

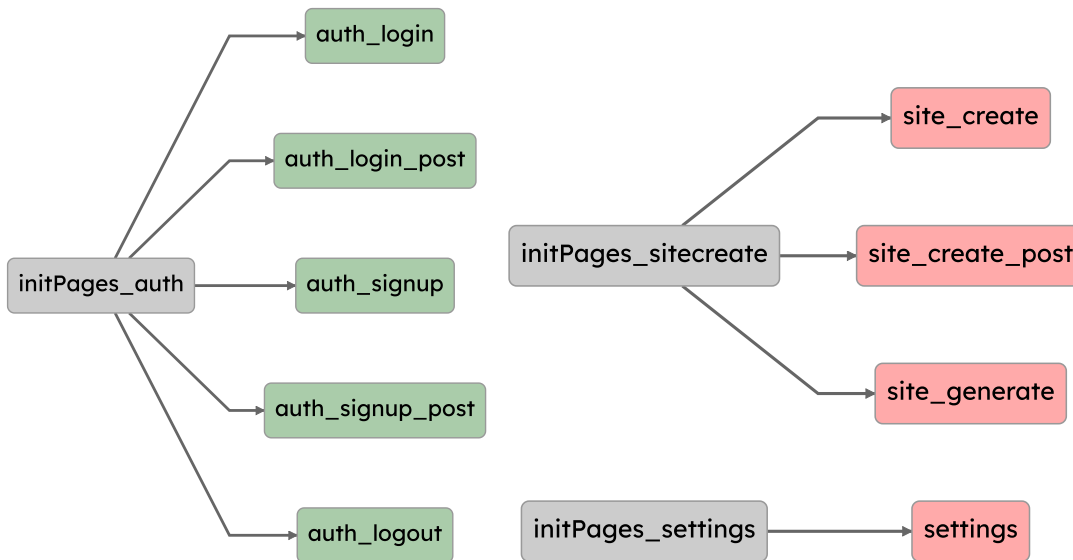
```

The homepage looked like this:



I also decided to reorganise the `app.route` functions in `__init__.py` into separate functions in the class, such as `initPages_auth` and `initPages_main`, to make the file easier to interpret. The diagram below shows the new organisation. Coloured boxes demonstrate routing functions: green means it has a use, and red means it is currently either abstract or simply outputs text on screen.



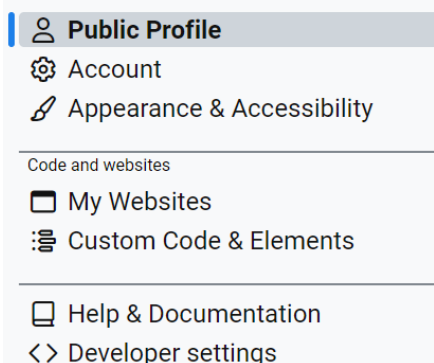


At this point in development, I was able to quickly build the frontend versions of the settings pages for the account system (they are listed in the desirable features section of the success criteria). I didn't add in database functionality to them yet as I knew it would take significantly longer.

The settings page, which is accessed from the navigation hamburger, consists of 7 sections, as outlined in the design section. 6 of them open in the same page, whereas the seventh, `Help & Documentation`, loads the help page. Similar to how the `base.html` Jinja architecture works, there is a `settings-base.html` template that the settings pages are built off of. I did not write the templates for `Custom Code & Elements` or `Developer Settings` as they are not yet implemented into the database. The template `My Websites` will need to be revisited when the website creation system is in place - for now, I used test data such as to simulate the user having sites.

To set the highlighted element in the local navigation bar, the Jinja variable `settingsSidebarActivated` is declared in child files so that, during rendering, the navigation bar knows which element to give the `is-active` class. It uses the if statement `{% if settingsSidebarActivated==<i> %} is-active{% endif %}` to assign the class, and the if statement `{% if not settingsSidebarActivated==<i> %} href="{{ url_for('main_help') }}"{% endif %}` to assign the link if it is not selected.

Settings



```
{% extends "base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set settingsSidebarActivated = get_flashed_messages()[0] %}

{% block content %}

<link href="{{url_for('static', filename='css/settings.css')}}" rel="stylesheet"
type="text/css" />

<div class="application-content">
  <div class="text-header-container">
    <h2 class="text header large dark one">Settings</h2>
  </div>

  <div class="settings-container">
    <div class="settings-sidebar">

      <a class="settings-sidebar-item one {% if settingsSidebarActivated==1 %}
is-active {% endif %} link notformatted"
{% if settingsSidebarActivated!=1 %} href="{{url_for('settings_profile')}}"
{% endif %}>

        <i class="settings-sidebar-item-icon fa-regular fa-user"></i>
        <span class="settings-sidebar-item-title text large">Public Profile</span>
      </a>

      <a class="settings-sidebar-item two {% if settingsSidebarActivated==2 %}
is-active {% endif %} link notformatted"
{% if settingsSidebarActivated!=2 %} href="{{ url_for('settings_admin') }}"
{% endif %}>

        <i class="settings-sidebar-item-icon fa-regular fa-gear"></i>
        <span class="settings-sidebar-item-title text large">Account</span>
      </a>

      <a class="settings-sidebar-item three {% if settingsSidebarActivated==3 %}
is-active{% endif %} link notformatted"
{% if settingsSidebarActivated!=3 %} href="{{ url_for('settings_looks') }}"
{% endif %}>

        <i class="settings-sidebar-item-icon fa-regular fa-paintbrush"></i>
        <span class="settings-sidebar-item-title text large">
          Appearance & Accessibility</span>
      </a>

      <div class="settings-sidebar-separator text one">Code and websites</div>

    </div>
  </div>
</div>
```

```

<a class="settings-sidebar-item four {% if settingsSidebarActivated==4 %}
is-active{% endif %} link notformatted"
{% if settingsSidebarActivated!=4 %} href="{{ url_for('settings_sites') }}"
{% endif %}>

    <i class="settings-sidebar-item-icon fa-regular fa-browser"></i>
    <span class="settings-sidebar-item-title text large">My Websites</span>
</a>

<a class="settings-sidebar-item five {% if settingsSidebarActivated==5 %}
is-active{% endif %} link notformatted"
{% if settingsSidebarActivated!=5 %} href="{{ url_for('settings_code') }}"
{% endif %}>

    <i class="settings-sidebar-item-icon fa-regular fa-list-timeline"></i>
    <span class="settings-sidebar-item-title text large">
        Custom Code & Elements</span>

</a>

<div class="settings-sidebar-separator text two"> </div>

<a class="settings-sidebar-item six {% if settingsSidebarActivated==6 %}
is-active{% endif %} link notformatted"
{% if settingsSidebarActivated!=6 %} href="{{ url_for('main_help') }}"
{% endif %}>

    <i class="settings-sidebar-item-icon fa-regular fa-book-blank"></i>
    <span class="settings-sidebar-item-title text large">
        Help & Documentation</span>

</a>

<a class="settings-sidebar-item seven {% if settingsSidebarActivated==7 %}
is-active{% endif %} link notformatted"
{% if settingsSidebarActivated!=7 %} href="{{ url_for('settings_dev') }}"
{% endif %}>

    <i class="settings-sidebar-item-icon fa-regular fa-code-simple"></i>
    <span class="settings-sidebar-item-title text large">
        Developer settings</span>

</a>

</div>

<div class="settings-content">

    {% block settings_content %}
    {% endblock %}

```

```

    </div>
  </div>
</div>

{% endblock %}

```

The `Public Profile` page contains inputs such as display name, bio, and URL, that will all appear on a user's profile page. As of yet, there is no post route function so the data does not get stored. after updating it.

Profile

Name

Your name probably isn't used much yet, but may appear in reference to websites that you have created. Your current name is set to "Test".

Profile Picture



The image must be a minimum of 200x200 pixels, and in a 1:1 ratio. This is not operational yet, and probably won't be for a while.

Bio

Url

All of the above fields are optional and can be left blank. By filling them out, you agree that this information can be displayed publicly and stored in our servers. We don't have a privacy statement, but we probably should.

/templates/settings-profile.html

```

{% extends "settings-base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set settingsSidebarActivated = 1 %}
{% set avatarImagePath = get_flashed_messages()[1] %}

{% block settings_content %}

<h3 class="settings-content-header text header dark">Profile</h3>

<form class="settings-content-options">

```



```
<div class="settings-content-option one">
  <h4 class="settings-content-option-title text dark bold">Name</h4>

  <input class="settings-content-option-input" type="text"
    name="settings_profile_name" placeholder="{{current_user.name}}">

  <p class="settings-content-option-caption text small dark">
    Your name probably isn't used much yet, but may appear in reference to
    websites that you have created. Your current name is set to
    "{{current_user.name}}".
  </p>
</div>

<div class="settings-content-option two">
  <h4 class="settings-content-option-title text dark bold">Profile Picture</h4>
  <div class="settings-content-option-image-upload">
    <figure class="settings-content-option-image-upload-figure"
      style="background-image:url( {{ url_for('static',filename='data/userIcons/'+
      current_user.user_id+'.png') }})"></figure>

  </div>

  <p class="settings-content-option-caption text small dark">
    The image must be a minimum of 200x200 pixels, and in a 1:1 ratio.
    This is not operational yet, and probably won't be for a while.
  </p>
</div>

<div class="settings-content-separator one"></div>

<div class="settings-content-option three">
  <h4 class="settings-content-option-title text dark bold">Bio</h4>

  <textarea class="settings-content-option-input" type="text"
    name="settings_profile_bio" maxlength=240
    placeholder="Tell us about yourself"></textarea>

</div>

<div class="settings-content-option four">
  <h4 class="settings-content-option-title text dark bold">Url</h4>
  <input class="settings-content-option-input" type="text"
    name="settings_profile_url">

</div>

<div class="settings-content-separator two"></div>

<div class="settings-content-option settings-content-update">
```

```

<p class="settings-content-option-caption text small dark">
  All of the above fields are optional and can be left blank. By filling them
  out, you agree that this information can be displayed publically and stored
  in our servers. We don't have a privacy statement, but we probably should.
</p>

<button class="field-submit btn primary thin rounded slide" type="submit">
  <span class="btn-content text uppercase primary">Update Profile</span>
</button>

</div>

</form>

{% endblock %}

```

The `Account` page contains functionalities such as deleting account and changing username. Like with the previous page, there is currently no functionality to these processes. There will be more information added to the change username prompt as, if a user does, it will change all of the URLs for their webpages, which could create issues for the user.

Account

Change Username

WARNING. Changing your username can create issues.

DEW IT

Change Email

New Email

Password

Your old and new email addresses will be sent confirmation codes in order to change them.

DEW IT

Export Account Data

Export all metadata, websites, and other stored information for your account. They will be available to download here.

EXPORT METADATA

EXPORT WEBSITES

DOWNLOAD ALL

Archive Account

This will disable your account until you wish to unlock it.

ARCHIVE YOUR ACCOUNT

Reset Account

This will remove all of your websites, custom code, and non-essential settings. The only remaining settings will be your username, name, email, and password. It is recommended that you export your account data before doing this.

RESET YOUR ACCOUNT

Delete Account

This will remove all trace of your account from our servers, and is an irreversible actions. It is recommended that you export your account data before doing this.

DELETE YOUR ACCOUNT

```
{% extends "settings-base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set settingsSidebarActivated = 2 %}

{% block settings_content %}

<h3 class="settings-content-header text header dark">Account</h3>

<form class="settings-content-options">

  <div class="settings-content-option one">
    <h4 class="settings-content-option-title text dark bold">Change Username</h4>

    <p class="settings-content-option-caption text small danger dark">
      <span class="text small danger bold">WARNING</span>. Changing your username
      can <a class="text small link danger bold">create issues</a>.
    </p>

    <div class="btn primary thin rounded slide m-xs-t">
      <span class="btn-content text uppercase primary">Dew it</span>
    </div>
  </div>

  <div class="settings-content-option one">
    <h4 class="settings-content-option-title text dark bold">Change Email</h4>

    <input class="settings-content-option-input" type="text"
      name="settings_admin_email" placeholder="New Email"><br>

    <input class="settings-content-option-input m-xs-t" type="password"
      name="settings_admin_email_password" placeholder="Password">

    <p class="settings-content-option-caption text small dark">
      Your old and new email addresses will be sent confirmation codes in order
      to change them.
    </p>

    <div class="btn primary thin rounded slide m-xs-t">
      <span class="btn-content text uppercase primary">Dew it</span>
    </div>
  </div>

  <div class="settings-content-separator one"></div>

  <div class="settings-content-option three">
    <h4 class="settings-content-option-title text dark bold">
      Export Account Data</h4>
```

```
<p class="settings-content-option-caption text small dark">
  Export all metadata, websites, and other stored information for your
  account. They will be available to download here.
</p>

<div class="btn primary thin rounded slide m-xs-t">
  <span class="btn-content text uppercase primary">Export Metadata</span>
</div>
<div class="btn primary thin rounded slide m-xs-t">
  <span class="btn-content text uppercase primary">Export Websites</span>
</div>
<div class="btn primary thin rounded slide m-xs-t">
  <span class="btn-content text uppercase primary">Download All</span>
</div>

</div>

<div class="settings-content-separator two"></div>

<div class="settings-content-option four">
  <h4 class="settings-content-option-title text dark bold danger">
    Archive Account</h4>

  <p class="settings-content-option-caption text small dark">
    This will disable your account until you wish to unlock it.
  </p>

  <div class="btn danger thin rounded slide m-xs-t">
    <span class="btn-content text uppercase danger">Archive your account</span>
  </div>
</div>

<div class="settings-content-option five">
  <h4 class="settings-content-option-title text dark bold danger">
    Reset Account</h4>

  <p class="settings-content-option-caption text small dark">
    This will remove all of your websites, custom code, and non-essential
    settings. The only remaining settings will be your username, name, email,
    and password. It is recommended that you export your account data before
    doing this.
  </p>

  <div class="btn danger thin rounded slide m-xs-t">
    <span class="btn-content text uppercase danger">Reset your account</span>
  </div>
</div>

<div class="settings-content-option six">
  <h4 class="settings-content-option-title text dark bold danger">
    Delete Account</h4>
```

```

<p class="settings-content-option-caption text small dark">
  This will remove all trace of your account from our servers, and is an
  irreversible action. It is recommended that you export your account data
  before doing this.
</p>

<div class="btn danger thin rounded slide m-xs-t">
  <span class="btn-content text uppercase danger">Delete your account</span>
</div>

</div>

</form>

{% endblock %}

```

The `Appearance and Accessibility` page doesn't do much at the moment, but will be added to as and when features are required.

Appearance and Accessibility

Tab Preference

4 (Default) ▼

When editing and rendering code, this determines how many spaces represent one tab. (Doesn't do anything yet)

/templates/settings-looks.html

```

{% extends "settings-base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set settingsSidebarActivated = 3 %}

{% block settings_content %}

<h3 class="settings-content-header text header dark">
  Appearance and Accessibility</h3>

<form class="settings-content-options">

  <div class="settings-content-option one">
    <h4 class="settings-content-option-title text dark bold">Tab Preference</h4>

    <select class="settings-content-option-input"
      name="settings_looks_tab_preference">
      <option value="1">1</option>
      <option value="2">2</option>
    </select>
  </div>
</form>

```

```

<option value="3">3</option>
<option value="4" selected="selected">4 (Default)</option>
<option value="5">5</option>
<option value="6">6</option>
<option value="8">8</option>
<option value="10">10</option>
<option value="12">12</option>
</select>

<p class="settings-content-option-caption text small dark">
  When editing and rendering code, this determines how many spaces represent
  one tab. (Doesn't do anything yet)
</p>
</div>

</form>

{% endblock %}

```

The `My Websites` page is the most useful of the settings pages, as of now. It contains a table of all of the user's current sites, with links to their respective pages. It also calculates and displays how large the pages are. As such, it requires a Python subroutine to work it out, displayed below.

My Websites

Websites	
@test	
 @test/Site 1 738B	Website Settings
 @test/Epic Webpage 685B	Website Settings

/templates/settings-sites.html

```

{% extends "settings-base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set settingsSidebarActivated = 4 %}
{% set flashedSiteNames = get_flashed_messages()[1] %}

{% block settings_content %}

<h3 class="settings-content-header text header dark">My Websites</h3>

```

```

<form class="settings-content-options">

  <div class="settings-content-table">
    <div class="settings-content-table-header">
      <div class="settings-content-table-row"><span class="text header dark">
        Websites</span></div>

      <div class="settings-content-table-row">
        <span class="text header small dark">@{{current_user.user_id}}</span>
      </div>
    </div>

    <div class="settings-content-table-content">

      {% for site in flashedSiteNames %}

      <div class="settings-content-table-row {% if site==flashedSiteNames[-1] %}
        last-row{% endif %}">

        <span class="settings-content-table-row-icon text dark">
          <i class="fa-regular {% if site[2] %}fa-lock{% else %}fa-book-bookmark
            {%endif%}"></i>
        </span>

        <span class="settings-content-table-row-title text dark">
          <a href='{{url_for("site_edit_home",name=site[0],site=site[1])}}'
            class="text link dark notformatted">@{{site[0]}}/{{site[1]}}</a>
        </span>

        <span class="settings-content-table-row-size text dark">
          {{site[3]}}
        </span>

        <span class="settings-content-table-row-settings text dark">
          <a href='{{url_for("site_edit_home",name=site[0],site=site[1])}}'
            class="text link primary notformatted">Website Settings</a>
        </span>

      </div>

      {% endfor %}

    </div>
  </div>

</form>

{% endblock %}

```

The changes to `__init__.py` are outlined below. This includes the `app.route` functions for each page. The number flashed is interpreted as `settingsSidebarActivated` in the Jinja template rendering.

The `My Websites` `app.route` function includes the line

```
self.convertByteSize(self.getFolderSize(self.os.path.abspath(site.sitePath))) .
```

This will:

- Fetch the file path of the current selected site (`site.sitePath`), which may look like `<username>\sites\<sitename>`
- Get the absolute path of the directory (`os.path.abspath()`)
- Call the `getFolderSize` subroutine and pass the absolute path - this will return a value, in bytes, of the directory
- Call the `convertByteSize` subroutine and pass the size of the directory, in bytes - this will return a human readable string showing how large the directory is.

The `getFolderSize` function is recursive, meaning it will call itself each time it finds another subfolder. It will go through each branch in a depth-first manner, before collapsing back up when there are no more subfolders.

changes to `__init__.py`

```
def getFolderSize(self,path:str) -> int:
    # Get the size of the base directory, should return 0
    size=self.os.path.getsize(path)

    # for all directories and files under the path
    for sub in self.os.listdir(path):
        subPath=self.os.path.join(path,sub) # get the path of the directory / file

        # get the size if it is a file
        if self.os.path.isfile(subPath): size+=self.os.path.getsize(subPath)

        # get the size if it is a directory by calling this function
        elif self.os.path.isdir(subPath): size+=self.getFolderSize(subPath)

    # return the size, in bytes
    return size
```

```
def convertByteSize(self,bytes:int) -> str:
    # Zero check
    if bytes==0: return "0B"

    # All possible sizes
    sizes=("B", "KB", "MB", "GB", "TB", "PB", "EB", "ZB", "YB")
```



```

i=int(math.floor(math.log(bytes,1024)))
p=math.pow(1024,i)
s=round(bytes/p,2)

if i==0: s=int(s)

return f"{s}{{sizes[i]}}"

```

```

def initPages_settings(self) -> None:

```

```

    @self.app.route("/account/settings/")
    @login_required
    def settings() -> Response:
        # redirect to the first settings page
        return redirect(url_for("settings_profile"))

    @self.app.route("/account/settings/profile")
    @login_required
    def settings_profile() -> str:
        # get the user icon for displaying
        flash(1,self.getUserImage(current_user.user_id))
        return render_template("settings-profile.html")

```

```

    @self.app.route("/account/settings/admin")
    @login_required
    def settings_admin() -> str:
        flash(2)
        return render_template("settings-admin.html")

```

```

    @self.app.route("/account/settings/looks")
    @login_required
    def settings_looks() -> str:
        flash(3)
        return render_template("settings-looks.html")

```

```

    @self.app.route("/account/settings/sites")
    @login_required
    def settings_sites() -> str:
        flash(4)

        # Flash a list of information about the user's sites
        # to be used in the site table.
        flash([
            [
                x.user_id,
                x.name,
                x.private,
                self.convertByteSize(self.getFolderSize(self.os.path.abspath(x.sitePath)))
            ] for x in self.Site.query.filter_by(user_id=current_user.user_id).all()])

```

```
        return render_template("settings-sites.html")

    @self.app.route("/account/settings/code")
    @login_required
    def settings_code() -> str:
        flash(5)
        return render_template("settings-code.html")

    @self.app.route("/account/settings/dev")
    @login_required
    def settings_dev() -> str:
        flash(7)
        return render_template("settings-dev.html")
```

Stage 4 - Creating a New Site

In this stage, I created the system for when a user creates a new site. This includes various subroutines outlined in the design section of the report, such as the website name formatting and storage of colour palettes. Some functionality has not yet been included yet, such as the ability to import a website. This is because there is not a way to export sites yet, and it would be quite time consuming to create a validation algorithm to make sure that the imported files are not malicious: I want to finish the essential success criteria before moving on to the desirable features.

Similar to the settings pages, I first created a base template to build the rest of the pages from.

/templates/site-create-base.html

```
{% extends "base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% block content %}

<link href="{{url_for('static', filename='css/site-create.css')}}"
rel="stylesheet" type="text/css" />

<div class="application-content">
  <div class="text-header-container">
    <h2 class="text header large dark one">Create a new site</h2>

  </div>

  <div class="main">
    <div class="main-content thin">
      {%block site_create_base%}
      {%endblock%}
    </div>
  </div>
</div>

{% endblock %}
```

The next step is to start creating the form for the first page. It consists of the website name, description, and whether the user wants the site to be public or private. Without any JavaScript, the page looked like this:

Create a new site

Want to import an exported site? [Import a website.](#)

Owner Website Name *

@test /

The name must be at least 4 characters long, and contain only lowercase alphanumeric characters, dashes, underscores and periods. Any illegal characters will be converted into dashes. It must also be unique! If you need inspiration for a name, you ain't gonna get any from me :)

Description (Optional)

☐ **Public**
Anyone online can see this website. Only you can edit it.

☐ **Private**
Only people who you give the link can view the website.

[CREATE SITE](#)

/templates/site-create.html

```
{% extends "site_create_base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% block site_create_base %}

<p class="text dark">
  Want to import an exported site?
  <a class="link text primary">Import a website.</a>
</p>

<div class="horizontal-separator one m-s-v"></div>

<form class="new-site-form one" method="post">

  <div class="form-input-container one">

    <div class="form-input-content-column">
      <span class="text large dark one">Owner</span>
      <span class="text large dark two">
        <span>@{{current_user.user_id}}</span><span class="m-m-1 m-s-r"></span>
      </span>
    </div>

    <div class="form-input-content-column">
      <span class="text large dark one">
        Website Name <sup class="text large danger">*</sup>
      </span>
    </div>
  </div>
</form>
```

```

<input id="new_site_name" class="new-site-input text dark two input
small-text-input" data-form-input-display="inactive" type="text"
name="new_site_name">

</div>
<div class="message-container m-s-t text small one visibly-hidden">
  Your site name will look like:
  <span class="message-container-jsedit"></span>
</div>

<p class="text dark m-s-t two">
  The name must be at least 4 characters long, and contain only lowercase
  alphanumeric characters, dashes, underscores and periods. Any illegal
  characters will be converted into dashes. It must also be unique!
  If you need inspiration for a name, you ain't gonna get any from me :)
</p>
</div>

<div class="horizontal-separator two m-s-v"></div>

<div class="form-input-container three">
  <span class="text large dark one">Description (Optional)</span>

  <input id="new_site_desc" class="new-site-input text dark two input
small-text-input" type="text" name="new_site_desc">
</div>

<div class="horizontal-separator three m-s-v"></div>

<div class="form-input-container two">

  <div class="input-checkbox">
    <input class="input-checkbox-input" name="new_site_privacy"
id="new_site_privacy_visible" type="radio" value="public">

    <span class="input-checkbox-icon">
      <i class="faicon fa-regular fa-book-bookmark"></i>
    </span>

    <div class="input-checkbox-text-container">
      <span class="input-checkbox-title text bold one">Public</span>
      <span class="input-checkbox-caption text small two">
        Anyone online can see this website. Only you can edit it.
      </span>
    </div>
  </div>

  <div class="input-checkbox">
    <input class="input-checkbox-input" name="new_site_privacy"
id="new_site_privacy_hidden" type="radio" value="private">

```

```

    <span class="input-checkbox-icon">
      <i class="faicon fa-regular fa-lock"
        style="color:var(--colors-warning)"></i>
    </span>

    <div class="input-checkbox-text-container">
      <span class="input-checkbox-title text bold one">Private</span>
      <span class="input-checkbox-caption text small two">
        Only people who you give the link can view the website.
      </span>
    </div>
  </div>

  <div class="horizontal-separator three m-s-v"></div>

  <button class="field-submit btn primary thin rounded slide" type="submit"
    disabled=true id="new_site_form_submit">

    <span class="btn-content text uppercase primary">Create Site</span>
  </button>
</div>

</form>

{% endblock %}

```

I had a basic outline of the JavaScript I needed to write, based on the algorithms in the Design section:

- Fetch all of this user's site names from the server
- Validation for the website name, including a function to remove all repeated dashes
- A way of styling the website name input to show whether it is valid
- A way of programatically enabling and disabling the submit button

To get all of the user's site names, I added a SQL query to the `app.route` function for this page, and flashed the results so that it could be used by the JavaScript.

changes to `__init__.py`

```

@self.app.route("/home/new/")
@login_required
def site_create() -> str:
    out=""

    # get all current site names for the logged in user, then flash (send) it to
    # the site, where it is processed by the javascript
    allusernames = [x.name for x in self.Site.query.filter_by(
        user_id=current_user.user_id).all()]

```

```

for name in allusernames:
    out+=name+", "

flash(out[:-1])

return render_template("site-create.html")

```

changes to /templates/site-create.html

```

<script> var flashedSiteNames="{{get_flashed_messages()[0]}}".split(",") </script>

```

/static/site-create.js

The `checkFormSubmitButton` function is called whenever an input is changed, to see whether all inputs are valid. If so, it sets the submit button (`formSubmit`) to enabled, and if not, sets it to disabled. This system can be hijacked via DevTools, so the same validation system will be implemented in Python as well.

```

function checkFormSubmitButton() {
    if (!(formName.getAttribute("data-form-input-display") == "success" ||
        formName.getAttribute("data-form-input-display") == "warning")) {
        formSubmit.setAttribute("disabled", "")
        return
    }

    if (!(formPrivacy1.checked) && !(formPrivacy2.checked)) {
        formSubmit.setAttribute("disabled", "")
        return
    }

    formSubmit.removeAttribute("disabled")
    return
}

```

The `verifyNameField` is called when the website name input (`formName`) is changed. It returns a data attribute that is used in the CSS to style the form.

```

function verifyNameField() {
    var nameInput = document.getElementById("new_site_name");
    var val = nameInput.value;

    hideFormMessage()

    if (val.length < 1) { return "inactive" }
    if (val.length < 4) { return "danger" }
}

```

```

var check=true
for (var letter of val) {
  if (requiredChars.includes(letter)) { check=false }
}

if (check) { return "danger" }
var sitenames = ["helloworld"]

if (flashedSiteNames.includes(val)) {
  editFormMessage("A site with this name already exists!")
  return "danger"
}

for (var letter of val) {
  if (!(allowedChars.includes(letter))) {
    editFormMessageSiteNameWarning(val)
    return "warning"
  }
}

if (hasRepeatedDashes(val)) {
  editFormMessageSiteNameWarning(replaceRepeatedDashes(val));
  return "warning"
}

hideFormMessage()
return "success"
}

```

This is the CSS that styles the website name input, to demonstrate how it interacts with the JavaScript.

```

.new-site-form.one .form-input-container.one
[data-form-input-display="success"].new-site-input {
  border-color:var(--colors-success);
  box-shadow: 0px 0px 22px -8px var(--colors-success);
}

.new-site-form.one .form-input-container.one
[data-form-input-display="warning"].new-site-input {
  border-color:var(--colors-warning);
  box-shadow: 0px 0px 22px -8px var(--colors-warning);
}

.new-site-form.one .form-input-container.one
[data-form-input-display="danger"].new-site-input {
  border-color:var(--colors-danger);
  box-shadow: 0px 0px 22px -8px var(--colors-danger);
}

```


This is what the input looks like when the `data-form-input-display` tag is changed. The four options are `inactive`, `danger`, `success`, `warning`.

Owner	Website Name *	Owner	Website Name *
@test	/ <input type="text"/>	@test	/ <input type="text"/>
Owner	Website Name *	Owner	Website Name *
@test	/ <input type="text"/>	@test	/ <input type="text"/>

The `verifyNameField` function makes use of 4 other functions, `hideFormMessage`, `editFormMessage`, `hasRepeatedDashes`, and `replaceRepeatedDashes`.

The first two are used to interact with the warning message that is displayed underneath the input.

```
function hideFormMessage() {
  messageContainer.classList.add("visibly-hidden")
  messageSpan.innerHTML=""
}
```

```
function editFormMessage(val) {
  messageContainer.classList.remove("visibly-hidden")
  newInner=val.toLowerCase()

  for (var i=0; i<newInner.length; i++) {
    var letter=newInner[i];
    if (!(allowedChars.includes(letter))) {
      newInner=newInner.replaceAt(i,"-")
    }
  }

  if (hasRepeatedDashes(newInner)) { newInner=replaceRepeatedDashes(newInner) }
  messageSpan.innerHTML=newInner
}
```

The second two handle the formatting of the input. When the validated name is being created, if it has any invalid characters in it, they will all be replaced with dashes (invalid characters are any characters not in `allowedChars`). This means that if you have a string such as `abc&& bcgf`, it will be turned into `abc---bcgf`. The function `hasRepeatedDashes` uses Regex to identify any adjacent dashes in the string, and the function `replaceRepeatedDashes` implements recursion to remove them all, changing `abc&& bcgf` to `abc-bcgf`.

These functions, in combination with `verifyField`, went through extensive testing to ensure that they worked properly. The data and results can be found in the appendix.

```
function hasRepeatedDashes(val) {
  for (var i=0;i<val.length;i++) {
    if (val[i] == "-" && val[i+1] == "-") {
      return true
    }
  }
  return false
}
```

```
function replaceRepeatedDashesRecursion(val) {
  for (var i=0;i<val.length;i++) {
    if (val[i] == "-" && val[i+1] == "-") {
      val.splice(i+1,1)
      val=replaceRepeatedDashesRecursion(val)
    }
  }
  return val
}

function replaceRepeatedDashes(val) {
  return listToStr(replaceRepeatedDashesRecursion(val.split("")))
}
```

The rest of the JavaScript contains variable allocation and event listeners:

```
var requiredChars = "qwertyuiopasdfghjklzxcvbnm1234567890"
var allowedChars = "qwertyuiopasdfghjklzxcvbnm-._1234567890";

var formSubmit = document.getElementById("new_site_form_submit");

var formName = document.getElementById("new_site_name");
var formDesc = document.getElementById("new_site_desc");

var formPrivacy1 = document.getElementById("new_site_privacy_visible");
var formPrivacy2 = document.getElementById("new_site_privacy_hidden");

var messageContainer = document.querySelector(
  ".new-site-form .form-input-container.one .message-container");

var messageSpan = document.querySelector(
  ".new-site-form .form-input-container.one .message-container
  .message-container-jsedit");
```

```

formName.addEventListener("keyup", (event) => {
    formName.setAttribute("data-form-input-display", verifyNameField())
    checkFormSubmitButton();
})

formPrivacy1.addEventListener("click", checkFormSubmitButton)
formPrivacy2.addEventListener("click", checkFormSubmitButton)

formSubmit.addEventListener("click", (event) => {
    if (!(formSubmit.getAttribute("disabled"))) {
        formSubmit.children[0].innerHTML = "CREATING SITE..."
    }
})

```

After completing the JavaScript validation, I then rewrote the code in Python and used it in the `app.route` post function, so that if the client-side validation was bypassed, it would still be validated server-side.

changes to `__init__.py`

```

@self.app.route("/home/new/", methods=["post"])
@login_required
def site_create_post() -> Response:

    def listToStr(var:list) -> str:
        out=""
        for char in var: out+=char
        return out

    def replaceToDash(var:str) -> str:
        # replaces any invalid characters in the string var with a dash, used to
        # format the site name correctly so that there arent any errors
        var=list(var)
        for i in range(len(var)):
            if var[i] not in "qwertyuiopasdfghjklzxcvbnm-._1234567890": var[i]="-"
        return listToStr(var)

    def replaceRepeatedDashes(var:str) -> str:
        # recursive function to remove adjacent dashes from a string
        var=list(var)
        for i in range(len(var)):

            # if this is the last character, return
            if i+1 >= len(var): return listToStr(var)

            # if this and the next character are dashes
            if var[i] == "-" and var[i+1] == "-":
                # remove this character
                del var[i]
                var = list(replaceRepeatedDashes(var))

```

```

        return listToStr(var)

# get the user inputs
sitename = request.form.get("new_site_name")
sitedesc = request.form.get("new_site_desc")
isPublic = request.form.get("new_site_privacy")=="public"

# remove adjacent dashes
sitename=replaceRepeatedDashes(replaceToDash(sitename.lower()))

# session can carry over variables between functions
session["new_site_sitename"]=sitename
session["new_site_sitedesc"]=sitedesc
session["new_site_isPublic"]=isPublic

return redirect(url_for("site_create_options_1"))

```

To ensure that all pages in the site creation process can only be accessed in order, all `app.route` functions have a piece of code referencing `flask.request.referrer` to ensure that the user is following the steps chronologically

changes to `__init__.py`

```

@self.app.route("/home/new/1")
@login_required
def site_create_options_1() -> str | Response:
    # stop people from starting halfway through the form i.e. if they didnt come
    # from the previous site_create page, send them to the start
    if not request.referrer == url_for("site_create",_external=True):
        return redirect(url_for("site_create"))

    return render_template("site-create-options-1.html")

```

The next page is the colour palette selection system. This involves functions for colour temperature adjustment, generation of lighter and darker variants, and storing the variables in a way that the backend can read them. The page would consist of a light or dark mode toggle, faders for certain parameters, and colour pickers for the primary, secondary, and accent colours. Without any JavaScript, the page looked like this:

Create a new site

Choose a color scheme!

Light or dark mode?

LIGHT DARK

Light & Dark Bounds

Monochromatic Temperature

#FFFFFF

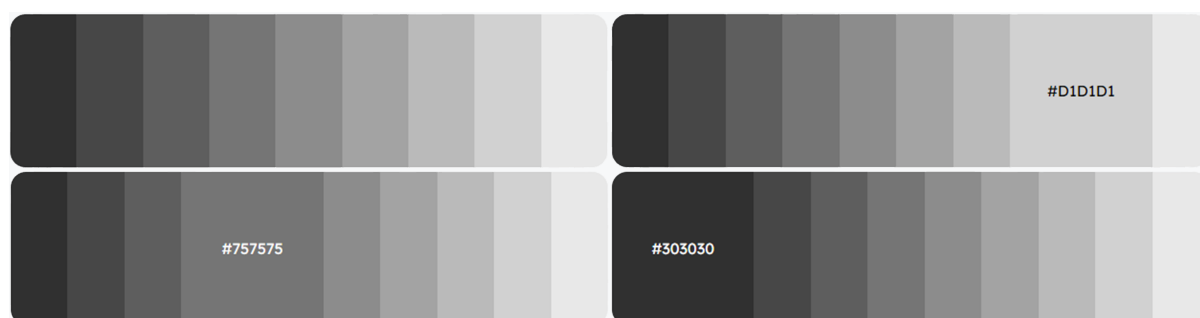
#000000

#E63946

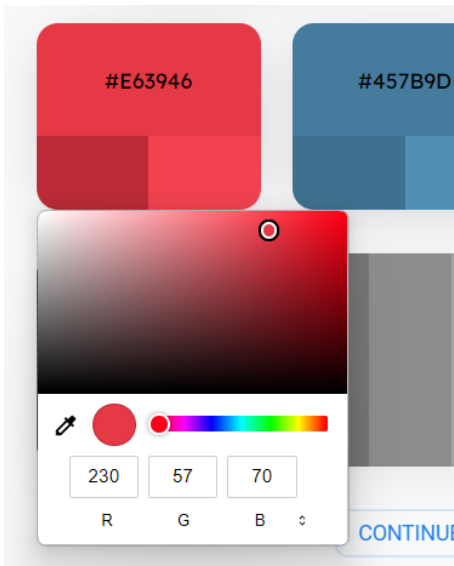
#457B9D

#A8DADC

The row of 9 mono-chromatic colours at the bottom will be set as the variables `--colors-grey-<100-900>`. To display them, it uses an "expanding column" system that I have used previously. When a column is hovered, it will "expand open" to reveal text inside. When a column is hovered, it looks like this:



For the colour pickers, I have used the native input colour picker option:



/templates/site-create-options-1.html

```
{% extends "site_create_base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% block site_create_base %}

<p class="text dark">Choose a color scheme!</p>
<div class="horizontal-separator one m-s-v"></div>

<form class="new-site-form two" method="post">

  <div class="light-dark-selector-container">
    <div class="light-dark-selector">
      <h2 class="text large center one">Light or dark mode?</h2>
      <div class="button-container">

        <div class="btn primary thin rounded slide from-left m-xs-t"
          id="new_site_lightModeToggle">
          <span class="btn-content text uppercase primary notextselect">
            Light
          </span>
        </div>

        <div class="btn secondary thin rounded slide from-right m-xs-t"
          id="new_site_darkModeToggle">

          <span class="btn-content text uppercase secondary notextselect">
            Dark
          </span>

        </div>

      </div>
    </div>
  </div>

</form>

</div>
```

```

    </div>
  </div>
</div>

<div class="horizontal-separator one m-s-v"></div>

<div class="color-options">
  <div class="input-slider-option one">

    <span class="text dark bold">Light & Dark Bounds</span>

    <div class="input-slider-and-number">

      <input class="input sliding-input" type="range" min="0" max="100"
        value="100" name="new_site_colors_light_dark_bounds"
        id="new_site_colors_light_dark_bounds_slider">

      <input class="input small-number-input" type="text" pattern="[-+]?d*"
        min="-100" max="100" id="new_site_colors_light_dark_bounds_number">

    </div>
  </div>

  <div class="input-slider-option two">

    <span class="text dark bold">Monochromatic Temperature</span>

    <div class="input-slider-and-number">

      <input class="input sliding-input" type="range" min="-100" max="100"
        value="0" name="new_site_colors_monochromatic_tint"
        id="new_site_colors_monochromatic_temperature_slider">

      <input class="input small-number-input" type="text" pattern="[-+]?d*"
        id="new_site_colors_monochromatic_temperature_number" min="-100"
        max="100">

    </div>

  </div>
</div>

<div class="horizontal-separator three m-s-v"></div>

<div class="color-display-container">

  <div class="color-display light-dark-display">

    <div class="color-single-card light-color">
      <span class="color-code text uppercase center">#ffffff</span>
    </div>
  </div>
</div>

```

```
<div class="color-single-card dark-color">
  <span class="color-code text uppercase center">#000000</span>
</div>

</div>

<div class="color-display main-color-display">

  <div class="color-triple-card primary-color">
    <input type="color" class="color-card-picker-input"
      id="new_site_colors_primary_picker" value="#e63946">

    <div class="color-triple-card-main">
      <span class="color-code text uppercase center">#e63946</span>
    </div>

    <div class="color-triple-card-sub-container">
      <div class="color-triple-card-sub one"></div>
      <div class="color-triple-card-sub two"></div>
    </div>
  </div>

  <div class="color-triple-card secondary-color">
    <input type="color" class="color-card-picker-input"
      id="new_site_colors_secondary_picker" value="#457b9d">

    <div class="color-triple-card-main">
      <span class="color-code text uppercase center">#457b9d</span>
    </div>

    <div class="color-triple-card-sub-container">
      <div class="color-triple-card-sub one"></div>
      <div class="color-triple-card-sub two"></div>
    </div>
  </div>

  <div class="color-triple-card accent-color">
    <input type="color" class="color-card-picker-input"
      id="new_site_colors_accent_picker" value="#a8dadc">

    <div class="color-triple-card-main">
      <span class="color-code text uppercase center">#a8dadc</span>
    </div>

    <div class="color-triple-card-sub-container">
      <div class="color-triple-card-sub one"></div>
      <div class="color-triple-card-sub two"></div>
    </div>
  </div>

</div>
```



```
<div class="color-display grey-display">
  <div class="color-columns grey-colors expanding-columns-container">

    <div class="color-column expanding-column g900">
      <span class="color-code expanding-text text uppercase center">
        #303030</span>
      </div>

    <div class="color-column expanding-column g800">
      <span class="color-code expanding-text text uppercase center">
        #474747</span>
      </div>

    <div class="color-column expanding-column g700">
      <span class="color-code expanding-text text uppercase center">
        #5e5e5e</span>
      </div>

    <div class="color-column expanding-column g600">
      <span class="color-code expanding-text text uppercase center">
        #757575</span>
      </div>

    <div class="color-column expanding-column g500">
      <span class="color-code expanding-text text uppercase center">
        #8c8c8c</span>
      </div>

    <div class="color-column expanding-column g400">
      <span class="color-code expanding-text text uppercase center">
        #a3a3a3</span>
      </div>

    <div class="color-column expanding-column g300">
      <span class="color-code expanding-text text uppercase center">
        #bababa</span>
      </div>

    <div class="color-column expanding-column g200">
      <span class="color-code expanding-text text uppercase center">
        #d1d1d1</span>
      </div>

    <div class="color-column expanding-column g100">
      <span class="color-code expanding-text text uppercase center">
        #e8e8e8</span>
      </div>

  </div>
</div>
```

```

</div>

<div class="submit-container">
  <button class="field-submit btn primary thin rounded slide" type="submit">
    <span class="btn-content text uppercase primary">Continue</span>
  </button>
</div>

</form>

{% endblock %}

```

/static/site-create-options-1.js

The first JavaScript I wrote was the definitions for the colour storage. For clarity, any long strings, such as queries, have been replaced with `...`. The `defaultColors` dictionary defines what the default colours are (the defaults are also defined in CSS for when the page loads) so that the page knows what to first render. The `colors` dictionary defines the user-selected colours.

```

var defaultColors = {
  "light": "#ffffff", "dark": "#000000",
  "primary": "#e63946", "primary-dark": "", "primary-light": "",
  "secondary": "#457b9d", "secondary-dark": "", "secondary-light": "",
  "accent": "#a8dadc", "accent-dark": "", "accent-light": "",
  "grey-100": "#303030", ... , "grey-900": "#e8e8e8"
}
var colors = defaultColors

var colorDisplay = {
  "light": [document.querySelector("..."), document.querySelector("...")],
  "dark": [document.querySelector("..."), document.querySelector("...")],

  "primary": [document.querySelector("..."), document.querySelector("...")],
  "primary-dark": [document.querySelector("...")],
  "primary-light": [document.querySelector("...")],

  "secondary": [document.querySelector("..."), document.querySelector("...")],
  "secondary-dark": [document.querySelector("...")],
  "secondary-light": [document.querySelector("...")],

  "accent": [document.querySelector("..."), document.querySelector("...")],
  "accent-dark": [document.querySelector("...")],
  "accent-light": [document.querySelector("...")],

  "grey-100": [document.querySelector("..."), document.querySelector("...")],
  ...
  "grey-900": [document.querySelector("..."), document.querySelector("...")],
}

```

The way the JavaScript manages to send the data to the server is by an input element in the form. This means it can be retrieved via the `flask.requests` module. I appended this element to the end of the form and then wrote the `updateStored` function, which sets the content of the input to the content of the `colors` dictionary. The definition for the element (referred to as `stored`), along with the other HTML elements are defined via the DOM.

```
<input id="color-output" class="visibly-hidden" type="text" value="."
name="new_site_color_options_dict">
```

```
function updateStored() {
  var out="";var keys=Object.keys(colors);
  for (var i=0; i<keys.length; i++) { out=out+keys[i]+":"+colors[keys[i]]+", " }
  out=out.slice(0,out.length-1)
  stored.value=out
}

var btnLight=document.getElementById("new_site_lightModeToggle");
var btnDark=document.getElementById("new_site_darkModeToggle");
var lightModeSelected=true;
var stored=document.getElementById("color-output")

var primaryColorPicker=
  document.getElementById("new_site_colors_primary_picker");

var secondaryColorPicker=
  document.getElementById("new_site_colors_secondary_picker");

var accentColorPicker=
  document.getElementById("new_site_colors_accent_picker");
```

I also added the utility function `setColor` to change a value in the dictionary, so that the code looked cleaner.

```
function setColor(k,v) { colors[k]=v }
```

Every time either the light or dark toggle is pressed, they call either the `setLightMode` or `setDarkMode` functions:

```
function setLightMode() {
  document.body.removeAttribute("data-kraken-darkmode")
  updateStored()
  updateLightDarkDisplay()
}
```

```
function setDarkMode() {
  document.body.setAttribute("data-kraken-darkmode", "")
  updateStored()
  updateLightDarkDisplay()
}
```

The `updateLightDarkDisplay` and `updateColorDisplays` functions are called to update the HTML element colours and texts whenever something is changed:

```
function updateColorDisplays() {
  colorDisplay["primary"][0].style.backgroundColor=colors["primary"]
  colorDisplay["primary"][1].innerHTML=colors["primary"]
  colorDisplay["primary-dark"][0].style.backgroundColor=colors["primary-dark"]
  colorDisplay["primary-light"][0].style.backgroundColor=colors["primary-light"]

  colorDisplay["secondary"][0].style.backgroundColor=colors["secondary"]
  colorDisplay["secondary"][1].innerHTML=colors["secondary"]
  colorDisplay["secondary-dark"][0].style.backgroundColor=
    colors["secondary-dark"]
  colorDisplay["secondary-light"][0].style.backgroundColor=
    colors["secondary-light"]

  colorDisplay["accent"][0].style.backgroundColor=colors["accent"]
  colorDisplay["accent"][1].innerHTML=colors["accent"]
  colorDisplay["accent-dark"][0].style.backgroundColor=colors["accent-dark"]
  colorDisplay["accent-light"][0].style.backgroundColor=colors["accent-light"]

  updateStored()
}

function updateLightDarkDisplay() {
  colorDisplay["light"][0].style.backgroundColor=colors["light"]
  colorDisplay["light"][1].innerHTML=colors["light"]
  colorDisplay["dark"][0].style.backgroundColor=colors["dark"]
  colorDisplay["dark"][1].innerHTML=colors["dark"]

  updateStored()
}
```

The `updateLightDarkVariables` is called whenever the "light & dark bounds" fader is changed:

```
function updateLightDarkVariables() {
  var val = 100-lightDarkBoundsSlider.value;

  var newColor = darken({h:0,s:0,l:100},val/12);
  newColor = hslToRgb(newColor.h,newColor.s,newColor.l);
  newColor = rgbToHex(newColor.r,newColor.b,newColor.g);
  setColor("light", "#"+newColor);
}
```

```

var newColor = lighten({h:0,s:0,l:0},val/8);
newColor = hslToRgb(newColor.h,newColor.s,newColor.l);
newColor = rgbToHex(newColor.r,newColor.b,newColor.g);
setColor("dark", "#"+newColor);

updateStored()
}

```

The `updateColorVariables` is called whenever a colour picker is changed, to generate light and dark variants of that colour. This meant that the user could select any of the colour inputs, and when they submitted it, it would immediately show a darker and lighter version underneath.



```

function updateColorVariables() {
  var changePercent = 20

  for (var color of ["primary","secondary","accent"]) {
    console.log(color)

    var newColor = hexToRgb(colors[color])

    // console.log("Color as RGB:")
    // console.log(newColor)

    newColor = rgbToHsl(newColor.r,newColor.g,newColor.g)

    // console.log("Color as HSL:")
    // console.log(newColor)

    newColor = darken(newColor,changePercent)

    // console.log("Darker Color as HSL:")
    // console.log(newColor)

    newColor = hslToRgb(newColor.h,newColor.s,newColor.l);

    // console.log("Darker Color as RGB:")
    // console.log(newColor)

    newColor = rgbToHex(newColor[0],newColor[1],newColor[2]);
  }
}

```

```

// console.log("Darker Color as Hex:")
// console.log(newColor)

setColor(color+"-dark", "#"+newColor)

newColor = hexToRgb(colors[color])
newColor = rgbToHsl(newColor.r,newColor.g,newColor.b)

// console.log("Color as HSL:")
// console.log(newColor)

newColor = lighten(newColor,changePercent)

// console.log("Lighter Color as HSL:")
// console.log(newColor)

newColor = hslToRgb(newColor.h,newColor.s,newColor.l);

// console.log("Lighter Color as RGB:")
// console.log(newColor)

newColor = rgbToHex(newColor[0]/255,newColor[1]/255,newColor[2]/255);

// console.log("Lighter Color as Hex:")
// console.log(newColor)

setColor(color+"-light", "#"+newColor)

console.log(colors[color+"-dark"])
console.log(colors[color+"-light"])

}

updateStored()

}

```

In the backend, the form retrieves the inputted colour variables via the form element `new_site_color_options_dict`. It then interprets the given input into a dictionary, and stores it in the `flask.session` object, meaning it can be accessed at a later date.

changes to `__init__.py`

```

@self.app.route("/home/new/1", methods=["post"])
@login_required
def site_create_options_1_post() -> Response:
    # new_site_color_options_dict is in the format
    # "key1:value,key2:value,key3:value"
    # so splitting by comma gives ["key1:value","key2:value","key3:value"]

```

```

# then iterate through the list, split by colon, and add parts to dictionary

# split into "k:v" list items
formOutput = request.form.get("new_site_color_options_dict").split(",")

# create output dictionary
colorOptions = {}
for pair in formOutput: # where pair is in the format "<key>:<value>"
    # split into ["<key>", "<value>"]
    colonsplit=pair.split(":")

    # append to dictionary - result = {... , "<key>":"<value>"}
    colorOptions[colonsplit[0]]=colonsplit[1]

# add color options dictionary to session
session["new_site_colorOptions"]=colorOptions

# proceed to next page
return redirect(url_for("site_create_options_2"))

```

The next page is the typeface selection system. This one is much simpler, and only requires event listeners for each typeface to set the active group. Inside each group, there is a hidden input element that will be assigned the `new_site_font_face_list_active` name if active, or the `new_site_font_face_list_inactive` name if inactive. This means that the backend only has to query the active name to find the selected group. The JavaScript ensures that only one input can have the active name.

Create a new site

Choose font family - individual elements can be customised

Lexend

Paragraph text - Roboto

Prata

Paragraph text - Lato

DM Sans

Paragraph text - Catamaran

Titillium Web

Paragraph text - Raleway

Caudex

Paragraph text - PT Mono

Noto Serif Display

Paragraph text - Lora

STAATLICHES

Paragraph text - Syne Mono

CONTINUE

The template makes use of a Jinja list of all fonts, where the two trailing booleans state whether the font has a googlefonts api link. The code iterates through each element, creating a new form entry for each, which has the header font, paragraph font, and google api imports for each font, if required (`<link href="https://fonts.googleapis.com/css2?family={{headerFont}}&display=swap" rel="stylesheet">`). It also sets the first entry as the active one.

/templates/site-create-options-2.html

```
{% extends "site_create_base.html" %}

{% set navbarLogoColor = "primary" %}
{% set navbarOptionsEnabled = True %}

{% set fontsList = [

    ["Lexend", "Roboto", True, True],
    ["Prata", "Lato", True, True],
    ["DM Sans", "Catamaran", True, True],
    ["Titillium Web", "Raleway", True, True],
    ["Caudex", "PT Mono", True, True],
    ["Noto Serif Display", "Lora", True, True],
    ["Staatliches", "Syne Mono", True, True],

]
%}

{% block site_create_base %}

    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>

    <p class="text dark">
        Choose font family - individual elements can be customised
    </p>

    <div class="horizontal-separator one m-s-v"></div>

    <form class="new-site-form three" method="post">

        <div class="text-options">
            {%for fontList in fontsList%}

                {%set counter=fontsList.index(fontList)+1%}

                {%set headerFont=fontList[0]%}
                {%set paraFont=fontList[1]%}

                <div class="text-option {{counter}}{%if counter==1% active{%endif%}">

                    {%if fontList[2]%}
```



```

        <link href="https://fonts.googleapis.com/css2?family={{headerFont}}
        &display=swap" rel="stylesheet">
    {%endif%}

    {%if fontList[3]%}
        <link href="https://fonts.googleapis.com/css2?family={{paraFont}}
        &display=swap" rel="stylesheet">
    {%endif%}

    <div class="text-option-header text header dark one"
    style="font-family: '{{headerFont}}'">{{headerFont}}</div>

    <div class="text-option-paragraph text two"
    style="font-family: '{{paraFont}}'">Paragraph text - {{paraFont}}</div>

    <input class="visibly-hidden text-option-list"
    value="{{headerFont}},{{paraFont}}" name="new_site_font_face_list_
    {%if counter==1%}active{%else%}inactive{%endif%}">

</div>

    {%endfor%}
</div>

<div class="submit-container">
    <button class="field-submit btn primary thin rounded slide" type="submit">
        <span class="btn-content text uppercase primary">Continue</span>
    </button>
</div>

</form>

<script src="{{url_for('static', filename='js/site-create-options-2.js')}}">

{% endblock %}

```

The JavaScript consists of event listeners for each typeface group, which will remove all other active tags and set the selected one to active.

/scripts/site-create-options-2.js

```

var textOptions = document.querySelectorAll(".new-site-form.three .text-option")

textOptions.forEach((e)=>{
    e.addEventListener("click", ()=>{

        textOptions.forEach((f)=>{
            f.classList.remove("active")
            f.querySelector(".text-option-list").name=
                "new_site_font_face_list_inactive"
        })
    })
})

```

```

        e.classList.add("active")
        e.querySelector(".text-option-list").name=
            "new_site_font_face_list_active"

    })
})

```

The back end queries `new_site_font_face_list_active` to get the selected font, gets the name of said fonts, and stores them as a list in the `flask.session`.

changes to `__init__.py`

```

@self.app.route("/home/new/2")
@login_required
def site_create_options_2() -> str | Response:
    # stop people from starting halfway through the form i.e. if they didnt come
    # from the previous site_create page, send them to the start
    if not request.referrer == url_for("site_create_options_1",_external=True):
        return redirect(url_for("site_create"))

    return render_template("site-create-options-2.html")

@self.app.route("/home/new/2", methods=["post"])
@login_required
def site_create_options_2_post() -> Response:
    # new_site_font_face_list_active is in the format
    # "<headerfontname>,<paragraphfontname>"
    # so split by comma to get ["<headerfontname>","<paragraphfontname>"]

    # get value from the active form element and split into list
    formOutput = request.form.get("new_site_font_face_list_active").split(",")

    # store font selection in session
    session["new_site_fontOptions"]=formOutput

    # proceed to next page
    return redirect(url_for("site_create_options_generate"))

```

After this, it redirects to `site_create_options_generate`. This will collect all of the `flask.session` data and store it in the dictionary `siteSettings` (and then clear the session data). It then calls the function `createSiteStructure` and passes said dictionary, and redirects the user to the site they just generated.

The `createSiteStructure` takes the `siteSettings` dictionary and converts it into a database object, along with generating the required server-side files and directories. The data was originally going to be stored in a JSON or XML file, however I opted to use config files (`.ini`) as the `ConfigParser` python library allows for easier editing and creation of data storage. The storage of site pages and code files is stored in a `.json` file. An example `site.ini` file may look like this:

```
[settings]
name = Epic-Webpage
user = test
desc = This is a description!

[color]
accent = #878acf
accent-dark = #696ec3
accent-light = #abaede
dark = #0f0f0f
grey-100 = #303030
grey-200 = #474747
grey-300 = #5e5e5e
grey-400 = #757575
grey-500 = #8c8c8c
grey-600 = #a3a3a3
grey-700 = #bababa
grey-800 = #d1d1d1
grey-900 = #e8e8e8
light = #f5f5f5
primary = #1cb566
primary-dark = #15894d
primary-light = #22d87a
secondary = #d9ca20
secondary-dark = #ada11a
secondary-light = #e6da56

[font]
header = DM Sans
body = Catamaran
```

The `generateFolderStructure` and `generateFileStructure` functions are called to create the files and directories. They take a list of paths and iterate through them, creating new files or directories if they don't already exist.

The `defaultHTMLPage` function returns a string of HTML content to set as the default content when a new site is created.

changes to __init__.py

```
@self.app.route("/home/new/generate")
@login_required
def site_create_generate() -> Response:
    # stop people from starting halfway through the form i.e. if they didnt come
    # from the previous site_create page, send them to the start
    if not request.referrer == url_for("site_create_options_2",_external=True):
        return redirect(url_for("site_create"))

    # Store session data into a dictionary, along with extra paramaters such as
    # when the site was created and the user id
    siteSettings={
        "name":session["new_site_sitename"],
        "user":str(current_user.user_id),
        "desc":session["new_site_sitedesc"] if session["new_site_sitedesc"]!="
            else "No Description Set",

        "created":self.datetime.datetime.utcnow(),
        "isPublic":session["new_site_isPublic"],
        "colorOptions":session["new_site_colorOptions"],
        "fontOptions":session["new_site_fontOptions"],
    }

    # Create the site database object and local storage
    self.createSiteStructure(siteSettings)

    # Clear any used session variables
    session["new_site_sitename"]=" "
    session["new_site_sitedesc"]=" "
    session["new_site_isPublic"]=" "
    session["new_site_colorOptions"]={}
    session["new_site_fontOptions"]=[]

    # Redirect to the site homepage
    return redirect(url_for("site_edit_home",
                            name=siteSettings["user"],
                            site=siteSettings["name"]))
```

```
def createSiteStructure(self,siteSettings:dict[str,str | any]) -> None:

    # get the prefix for the site path
    sitePath=self.os.path.abspath(
        f"static/data/userData/{siteSettings['user']}/sites/{siteSettings['name']}")

    # get the path for the site config file
    siteConfigFile=f"{sitePath}/site.ini"
```

```

# create a list of required directories
folderStructure = [
    f"{sitePath}",
    f"{sitePath}/output",
    f"{sitePath}/files"
]

# create a list of required files
fileStructure = [
    siteConfigFile,
    f"{sitePath}/siteDat.json",
    f"{sitePath}/files/1.html"
]

# create the required directories and files in the local storage
self.generateFolderStructure(folderStructure)
self.generateFileStructure(fileStructure)

# create the JSON file for the site, which contains code locations
with open(f"{sitePath}/siteDat.json", "w") as f:
    f.write("{\"pages\":{\"Home\":\"1.html\"},\"css\":{},\"js\":{}}")

# create the default webpage for the site
with open(f"{sitePath}/files/1.html", "w") as f:
    f.write(self.defaultHtmlPage(siteSettings["name"],
                                siteSettings["desc"],
                                siteSettings["user"]))

# Create the config parser for the site.ini file
with ConfigParser() as cfgContent:

    cfgContent.read(siteConfigFile)

    # create the required sections
    for section in ["settings", "color", "font"]:
        try: cfgContent.add_section(section)
        except: pass

    # Add basic information to the settings section
    cfgContent.set("settings", "name", siteSettings["name"])
    cfgContent.set("settings", "user", siteSettings["user"])
    cfgContent.set("settings", "desc", siteSettings["desc"])

    # Add the color palette
    for key in siteSettings["colorOptions"]:
        cfgContent.set("color", key, siteSettings["colorOptions"][key])

    # Add the typeface
    cfgContent.set("font", "header", siteSettings["fontOptions"][0])
    cfgContent.set("font", "body", siteSettings["fontOptions"][1])

```

```

# write to the site.ini file
with open(siteConfigFile,"w") as f:
    cfgContent.write(f)
    f.close()

# Create a new Site object using the varaibles given
newSite=self.Site(
    name=siteSettings["name"],
    datecreated=siteSettings["created"],
    private=not siteSettings["isPublic"],
    deleted=False,
    user_id=siteSettings["user"],
    sitePath=sitePath,
)

# Add and commit the new site to the database
self.db.session.add(newSite)
self.db.session.commit()

```

```

def generateFileStructure(self,files:list) -> None:
    for file in files: # Iterate through given list of files

        # Ignore if file already exists
        if self.os.path.exists(file): continue

        # Create the empty file
        try: with open(file,"w") as f: f.close()
        except OSError as e: raise e

```

Stage 5 - Code Documentation

To ensure that the code is easy to interpret by any other developers, and also allow it to integrate better into IDEs, I went through each function in `__init__.py` to add more detailed comments, variable type declarations, and Sphinx docstrings to the start of each function.

Sphinx Docstring Format

The `Sphinx` docstring format is a standard used to automatically generate documentation for code. It describes the purpose of the function, the parameters and their types, and the return value. The following is taken from [Sphinx Docstring Documentation](#).

In general, a typical `Sphinx` docstring has the following format:

```
[Summary]

:param [ParamName]: [ParamDescription], defaults to [DefaultParamVal]
type [ParamName]: [ParamType](, optional)
...
:raises [ErrorType]: [ErrorDescription]
...
:return: [ReturnDescription]
:rtype: [Return Type]
```

A pair of `:param:` and `:type:` directive options must be used for each parameter we wish to document. The `:raises:` option is used to describe any errors that are raised by the code, while the `:return:` and `:rtype:` options are used to describe any values returned by our code.

Note that the `...` notation has been used above to indicate repetition and should not be used when generating actual docstrings, as can be seen by the example presented below.

Every function in `__init__.py` has been documented in this format. The full results of this can be seen in Appendix A. Here are some examples, with example screenshots of when the functions are being used.

```

class Kraken():
    """Initializes the Kraken application and runs it on the given host and port

    :param host: The host of the application, given in the format
        ``"<i>.<i>.<i>.<i>""
    :type host: str
    :param port: The port of the application
    :type port: int

    :returns: Void
    :rtype: None
    """

```

```

class Kraken(
    host: str,
    port: int
)

```

Initializes the Kraken application and runs it on the given host and port

```

:param host: The host of the application, given in the format "<i>.<i>.<i>.<i>"
:type host: str
:param port: The port of the application
:type port: int
:returns: Void

```

```

def initFlask(self) -> None:
    """Initializes the Flask application by setting up the ``Flask`` object,
    configuring certain attributes, and creating the website pages via the
    function ``self.initPages()``.

    :returns: Void
    :rtype: None
    """

```

```

(method) def initFlask(self: Self@Kraken) -> None

```

Initializes the Flask application by setting up the `Flask` object, configuring certain attributes, and creating the website pages via the function `self.initPages()` .

```

:returns: Void
:rtype: None

```



```
def main_home() -> str:
    """Page for ``/home/``.

    This page is the homepage, containing links to all of the current user's sites.

    Requires a user to be logged in.

    Flashes a list of the users sites, in the format
    ``[<userid>,<sitename>,<isprivate>]``, if the user has any sites.

    :returns: The HTML content of the page, generated from the Jinja template syntax
    :rtype: str
    """
```

```
(function) def main_home() -> str
```

Page for `/home/` .

This page is the homepage, containing links to all of the current user's sites.

Requires a user to be logged in.

Flashes a list of the users sites, in the format `[<userid>,<sitename>,<isprivate>]` , if the user has any sites.

:returns: The HTML content of the page, generated from the Jinja template syntax

:rtype: str

```
def auth_login() -> str | Response:
    """Page for ``/login/``.

    This page is the login page. It contains the login form.
    Will redirect to ``main_home`` if a user is logged in.

    Flashes an empty list of values (``[False,"","","",""]``) to stop errors when
    generating the Jinja template.

    :returns: The HTML content of the page, generated from the Jinja template
              syntax OR a redirect to the ``main_home`` page if a user is logged in
    :rtype: str | Response
    """
```

```
(function) def auth_login() -> (str | Response)
```

Page for `/login/` .

This page is the login page. It contains the login form.

Will redirect to `main_home` if a user is logged in.

Flashes an empty list of values (`[False,"","","",""]`) to stop errors when generating the Jinja template.

:returns: The HTML content of the page, generated from the Jinja template syntax OR a redirect to the `main_home` page if a user is logged in

:rtype: str | Response

```
def auth_login_post() -> Response:
    """POST method of ``/login/``.

    Fetches the given user information from the form via ``flask.request``, and
    validates it. The user is logged in if the data is valid,
    and shown an error message if it is not.

    Flashes an appropriate error message, if required.

    :returns: A redirect to the ``main_home`` or ``auth_login`` depending on the
        success of the login attempt
    :rtype: Response
    """
```

```
(function) def auth_login_post() -> Response
```

POST method of `/login/` .

Fetches the given user information from the form via `flask.request` , and
validates it. The user is logged in if the data is valid, and shown an error
message if it is not.

Flashes an appropriate error message, if required.

:returns: A redirect to the `main_home` or `auth_login` depending on the
success of the login attempt

:rtype: Response

```
def site_create_generate() -> Response:
    """Method for ``/home/new/generate``.

    This page is the second third creation page. It contains the button styling
    selection system.
    Requires a user to be logged in.
    Requires the referrer to be ``site_create_options_2``

    :returns: A redirect to the new site homepage (``site_edit_home``) if
        successful OR a redirect to ``site_create`` if
        ``site_create_options_2`` was not the referrer.
    :rtype: Response
    """
```

```
(function) def site_create_generate() -> Response
```

Method for `/home/new/generate` .

This page is the second third creation page. It contains the button styling
selection system.

Requires a user to be logged in. Requires the referrer to be
`site_create_options_3`

:returns: A redirect to the new site homepage (`site_edit_home`) if successful
OR a redirect to `site_create` if `site_create_options_3` was not the
referrer.

:rtype: Response

```
def verifyField(self, field:str, fieldName:str, mustHaveChar:bool=True, minLen:int=3,
    canHaveSpace:bool=False, canHaveSpecialChar:bool=True) -> str:
    """The validation system for user inputs implemented in the signup page.

    :param field: The content of the field that is being validated.
    :type field: str
    :param fieldName: The name of the field that is being validated.
    :type fieldName: str
    :param mustHaveChar: Boolean flag determining whether the field must not be
        empty, defaults to ``True``
    :type mustHaveChar: bool, optional
    :param minLen: The minimum length of the field content, defaults to ``3``
    :type minLen: int, optional
    :param canHaveSpace: Boolean flag determining whether the field can have spaces
        defaults to ``False``
    :type canHaveSpace: bool, optional
    :param canHaveSpecialChar: Boolean flag determining whether the field is
        allowed to contain any of ``%&{ } \ < > * ? / $ ! ' " : @ + | = ` ` ,
        defaults to ``True``
    :type canHaveSpecialChar: bool, optional

    :returns: An empty string if the field is valid OR an error message if the
        field is invalid
    :rtype: str
    """
```

```
(method) def verifyField(
    self: Self@Kraken,
    field: str,
    fieldName: str,
    mustHaveChar: bool = True,
    minLen: int = 3,
    canHaveSpace: bool = False,
    canHaveSpecialChar: bool = True
) -> str
```

The validation system for user inputs implemented in the signup page.

```
:param field: The content of the field that is being validated.
:type field: str
```

```
:param fieldName: The name of the field that is being validated.
:type fieldName: str
:param mustHaveChar: Boolean flag determining whether the field must not be
empty, defaults to True
:type mustHaveChar: bool, optional
:param minLen: The minimum length of the field content, defaults to 3
:type minLen: int, optional
:param canHaveSpace: Boolean flag determining whether the field can have
spaces, defaults to False
:type canHaveSpace: bool, optional
:param canHaveSpecialChar: Boolean flag determining whether the field is
allowed to contain any of %&{ } \ < > * ? / $ ! ' " : @ + | = , defaults to True
```

```
:type canHaveSpecialChar: bool, optional

:returns: An empty string if the field is valid OR an error message if the field is
invalid
:rtype: str
```

```
def createSiteStructure(self,siteSettings:dict[str,str | any]) -> None:
    """Creates a new site in the database and in local storage.

    :param siteSettings: a dictionary containing the information about the site,
        with the keys: ``name`` ``user`` ``desc`` ``created``
        ``isPublic`` ``colorOptions`` ``fontOptions``
        ``buttonOptions``
    :type siteSettings: dict

    :returns: Void
    :rtype: None
    """
```

```
(method) def createSiteStructure(
    self: Self@Kraken,
    siteSettings: dict[str, str | Any]
) -> None
```

Creates a new site in the database and in local storage.

```
:param siteSettings: a dictionary containing the information about the site, with
the keys: name user desc created isPublic colorOptions
fontOptions buttonOptions
:type siteSettings: dict

:returns: Void
:rtype: None
```

```
def convertByteSize(self,bytes:int) -> str:
    """Converts a given byte value into a human readable version

    :param bytes: The amount of bytes to convert
    :type bytes: int

    :returns: A human readable version of the value
    :rtype: str
    """
```

```
(method) def convertByteSize(
    self: Self@Kraken,
    bytes: int
) -> str
```

Converts a given byte value into a human readable version

```
:param bytes: The amount of bytes to convert
:type bytes: int

:returns: A human readable version of the value
:rtype: str
```

Variable Type Declarations

Despite their non-functionality in Python, I decided to include variable type declarations for most variables so that IDEs can identify what a variable is intended for, speeding up testing and development, especially that done by other people, and because it makes reading the code easier and more informative by making it clear which variables are which type. This also goes for declaring input-output types in function declarations, which I was already doing.

Variable type declaration in Python is done with the syntax:

```
<variablename>:[type] = <content>
```

This can also be done in functions:

```
def <functionname>(<variablename:[type]=[defaultvalue]>) -> [returntype]:  
    pass
```

For example, the function `getSiteCfg` uses this form of declaration, which is especially useful considering that you can't look at the content of the function to identify what data type is being returned. However, the function declaration ends in `-> list[str]`, showing that the function will return a list of strings. This makes development workflow that bit faster due to clearly displaying variable types.

```
def getSiteCfg(self,siteName:str) -> list[str]:  
  
    ...  
  
    # This doesn't tell you what data type it is  
    cfgContent=ConfigParser()  
    cfgContent.read(cfgPath)  
  
    # You cannot identify what data type cfgContent is, so the list[str] line  
    # in the function declaration is useful in that sense  
    return cfgContent
```

Another example of variable type declaration is when the post methods are retrieving data from HTML forms:

```
# The declaration :str or :bool shows what the form element will return  
username:str = request.form.get("username")  
password:str = request.form.get("password")  
remember:bool = request.form.get('remember')
```