Table of Contents

Analysis

Problem identification

With the internet constantly growing and more and more people relying on it, the demand for websites is continuously increasing. They can range in style from business portfolios and online stores to games. Regardless of who you are or what you do, a website is often expected of you, especially for businesses. However, the problem of creating a professional and functional website can be a daunting task for many individuals and small businesses, especially those who need more technical expertise or resources. Existing website builders may be challenging to navigate, require extensive coding knowledge, or lack the necessary design flexibility to create a unique and effective online presence. Furthermore, many website builders are geared toward specific industries or types of websites, making it difficult for users to find a platform that meets their specific needs. The task of manually programming it can seem very daunting.

The main aim of this project is to develop a website that allows clients to produce their website via a simple user interface, which alleviates the technical intricacies of HTML, CSS, and JavaScript and will enable them to focus on creating a website that reflects their brand and meets their business goals. Clients can select from various styles and themes (or create their own), upload media such as images and videos, and then interact with a drag-and-drop interface to organise a webpage. Each website they create would have a page dedicated to it, with options to customise the styles of the site, the ability to create, organise, and link together pages of the website, preview the website in a variety of display sizes, and add pre-made elements and edit the parameters of the elements in the site.

A drag-and-drop website builder that is easy to use is increasingly important in today's digital age, where having an online presence is crucial for businesses, organisations, and individuals. A user-friendly website builder that offers a wide range of customisable design options would greatly benefit users who may not have the technical know-how to create a website independently

Moreover, the rise of mobile devices and the increasing importance of responsive design have made it crucial for websites to be optimised for different screen sizes and devices. As such, this project would also be geared towards easily creating mobile-friendly websites to ensure optimal user experience.

The requirements for a client to be able to use it would also be low due to the entire application being contained within the website: this means that the client would only need a web browser and an internet connection. They do not need to install software onto their computer, nor do they need to worry about software updates.

In summary, the problem that this website builder aims to solve is to provide an easy-to-use, drag-and-drop website builder for individuals and small businesses, which can be easily customised to their specific needs and optimised for different devices without requiring technical expertise. This will greatly benefit users by allowing them to create a professional and functional website that meets their business goals and reflects their brand without needing a background in technology or coding.

Stakeholders

The clients for this solution could be a wide range of individuals or organisations looking to create a website without the need for technical competency or dedicating large amounts of time to it.

Some potential clients could include:

- Individuals: People who want to create a personal website, such as a blog, portfolio, or online resume.
- Small businesses: Owners or managers of small businesses who want to create a website for their business, such as an online store or service provider website.
- Entrepreneurs: Entrepreneurs who want to create a website for their startup or new business venture.
- Non-profit organisations: Non-profit organisations that want to create a website to promote their mission and raise awareness.
- Freelancers: Freelancers and independent contractors who want to create a website to showcase their work and promote their services.

The stakeholders are divided in two main groups: those who have some technical knowledge and can help with alpha-testing and feedback, and those who are not very technically literate, as that is the target audience of this solution. Having two groups will mean I get a variety of useful feedback when developing and beta-testing.

Stakeholder	Role	Technical Knowledge	Notes
Archie	Student	Medium	Will be able to help me with testing. Has some experience using website builders
Luke	Student	Medium	Will be able to help me with testing. Enjoys creating websites in his free time, well versed in usability and accessibility

Stakeholder	Role	Technical Knowledge	Notes
Sarah	Member of local Scout group	Little	Looking for a way to create a website without hiring a professional developer
Jake	Small business owner	Little	Interested in integration with social media
Kevin	Professional web hosting provider	High	Can provide insights into platform's compatibility with different hosting environments. Has insight into Site Engine Optimisation (referred to as SEO throughout)
David	Barber shop owner and tech enthusiast	High	Excited to test features and provide feedback on capabilities and limitations

Why it is suited to a computational approach

This problem is suitable for a computational approach because it automates many technical and design tasks typically associated with creating a website. This is achieved by breaking down the website-building process into smaller, more manageable tasks that a computer program can easily handle.

For example, it can implement a drag-and-drop interface to allow users to easily add and arrange elements on a webpage, such as text, images, and videos, without coding. The website builder can also include a visual editor that allows users to easily customise the design and layout of their website, such as selecting from a variety of pre-designed templates or themes. The website builder can also apply pre-defined styles and formatting to the website and organise and link the pages together.

In addition, the solution can also use computational algorithms to optimise the website for different devices and screen sizes, ensuring that the website is responsive and easy to navigate on any device. This can be done by using CSS media queries, which can change the website's layout based on the screen's width, and JavaScript libraries that can detect the device and size of the screen and adjust the layout accordingly.

Using a computational approach can also ensure that the website is secure, fast, and reliable. It can use techniques like minifying, compression, and caching to make the website load faster and use authentication and authorisation mechanisms to ensure the website is secure.

The solution will be accessible through a server that hosts a website, which requires a computer to use. No alternative would not require a computer to be able to create a website, as at some point, the user will need to write and host the code that they have produced. This solution would act as a bridge between the client and the code, making it easy for users to create a website without needing to understand the technical details of coding.

Computational methods that the solution lends itself to:

Problem recognition

This solution is suitable for problem recognition because it addresses a specific problem that many individuals and small businesses face: the difficulty of creating a professional and functional website without technical expertise or resources. By providing a user-friendly, drag-and-drop interface, a website builder allows users to quickly create and edit their website without the need for coding or technical expertise. This addresses the problem of users not having the technical skills or resources to create a website themselves.

Furthermore, a website builder can also use computational algorithms to optimise the website for different devices and screen sizes, ensuring that the website is responsive and easy to navigate on any device. This addresses the problem of the rise of mobile devices and the increasing importance of responsive design in today's digital age.

Problem decomposition

The problem can be broken down into smaller tasks that must be programmed for the program to operate effectively.

- Creating an account system and a database to store the user's data.

> The account system would allow many users to access the server and edit their sites. The data would be stored in a server-side SQL database. The sites would be stored separately on the server.

- Creating a menu system for users to navigate to access different sites.
- Storing a list of template elements the user can preview and use in their sites.

> I need to decide how to store the template elements, display them to the user, and then implement them into a user's site while still allowing them to edit them.

- Creating a simple drag-and-drop interface that is easy to use and understand.

> This will probably be based on the grid-based positioning system that many existing website builders use or the constraint system that applications such as android studio use; however, the constraint system may not work with how HTML is created.

- Creating an effective way of storing the users' sites on the server.

> They would either be stored in HTM, CSS, and JS files, which would remove the need to convert them, or they could be stored all in XML files, which would make accessing the files easier: the program could convert the XML elements into HTML, CSS, and JS to allow it to be shown. This would allow for the storage of more information about the site and elements and would mean it could all be in one file, including all the separate pages.

- Converting the user's site into runnable HTML, CSS, and JavaScript so they can download and use it.

> There would need to be a way for JavaScript to do it so that the user can edit the site in the editor and a way for the server (written in Python) to do it as well

Divide and conquer

These smaller steps are all doable on their own, and combining them would make a divide-and-conquer approach. The advantage of it being coded in a modular way is that each part can be tested and built on its own without relying on other parts of the project.

Abstraction

The program uses abstraction as it removes the complex process of programming code by transferring it into a more straightforward, graphical interface - it provides a high-level interface for users to create a website while hiding the underlying implementation details. This removes the need for the client to have knowledge and experience in programming, opening the market to a much larger audience.

For example, the proposed drag-and-drop interface abstracts the implementation details of adding and arranging elements on a webpage using HTML and CSS code.

Initial talks with stakeholders

After meeting with each of the stakeholders and proposing the solution to them, I asked them each for feedback on what they would want the project to include, how they would want it to function, and, if they had technical knowledge, how I could implement the functionality. The questions I asked each person were:

1. "How much technical experience do you have?"
2. If they have technical experience: "Do you have any experience creating websites yourself, and if so how much?"
3. "Do you have any experience using existing website builders, and if so, what are your thoughts on them?"
4. "What are the ideal features that you would look for in a website builder?"
5. If they have a lot of technical knowledge: "What methods would you suggest when creating this project?"
6. "Do you have any other comments, ideas or requests about my proposed project?"

Archie, Student

1. Technical experience

> I study [A level] Computer Science and have used Python for a long time. The course teaches many various aspects of building applications and programming paradigms, which will be useful in the creation of a website builder.

2. Creating your own websites

> I haven't really used HTML, CSS, and JavaScript outside of small things here and there

3. Existing website builders

> A lot of website builders that I have seen before are very feature heavy and convoluted, which can often obscure the creation of a website. The paywall against many of their features often annoys me

4. Ideal features

> If I were looking for a website builder, I would want it to be easy to pick up how to use, and have it so that I could edit an existing site quickly - very user friendly. The ability to invite other people to help with the creation would be good, and the ability to import and export sites. I'd also want it to have a lot of customisability so that I can make the website look how I want.

6. Other comments

> Make sure to use VS code as its 100% better than everything else.

Luke, Student

1. Technical experience

> I mainly program in C (and sometimes Java), having learnt them while doing computer science. I've used web languages quite a lot.

2. Creating your own websites

> I find it fun to create websites with crazy premises in my free time. It helps me expand my knowledge and skills. Each one is better than the last. Recently I've been trying to implement more usability and accessibility requirements into the websites that I create, such as contrast ratios and ARIA tags.

3. Existing website builders

> I haven't really looked into website builders that much as I get more enjoyment from building them myself - its more of a hobby than a serious thing, and you have to pay for most builders. I have taken a look at bootstrap before, which is quite powerful.

4. Ideal features

> Well, you start with the drag-and-drop, and build from there. You want an easy way to position and reposition elements throughout the site - the ability to reorganise everything is really important. Styling customisability, so global styles and then the ability to change individual elements as well. Custom CSS and JS could also be a cool idea, but you'd need to look out for people putting in malicious code. Various templates that people can choose from is always a good idea, along with tags and filters for them. And accessibility: automatically assigning ARIA tags so that the user doesn't have to do it, checks to make sure the colours are high contrast enough, screen reader support. I'd be happy to help you with getting that stuff correct.

5. Programming methods

6. Other comments

> When I took a look at android studio, the way they do their positioning system is with constraints and XML formatted elements. That could be a cool way to do the positioning system, but it might also be a lot harder to program.

Sarah, Scout Leader

1. Technical experience

> No

3. Existing website builders

> The scout group is beginning to look into creating a website, so you've come along at the perfect time. In terms of what we've seen already, a couple of ideas have been put backwards and forward but we haven't come to a decision yet.

4. Ideal features

> I mean, as a scout group, we would want to be able to put up photos and videos of what we've been doing so that people are encouraged to join - we need to be able to change it quickly, and have multiple people be able to change it. We'd need to be able to change the colour scheme so that it matches with the official scout branding. Sign up forms would be a feature we'd need so that people can sign up straight from the website. And the ability to add a privacy policy, so that it's compliant with our GDPR regulations. If I'm honest, a lot of us aren't very technically savvy, so it would need to be pretty basic.

Jake, Business Owner

1. Technical experience

> I did do some stuff at school, but it was only a little and quite a while ago

3. Existing website builders

> I've not really considered having a website for [the business], as it's quite small, and I've found that social media is good enough at the moment

4. Ideal features

> Well, the biggest thing that comes to mind is having my social media feeds on the front page so that people can easily get up-to-date information and see the latest posts quickly. I'd want a lot of styles to choose from so that I can make it look how I want it too - but not so many that I can't choose.

Kevin, Web Hosting Provider

1. Technical experience

> I work as a professional web hosting provider, making sure that clients have the best hosting plan based on requirements such as ease of use, security requirements, or expected lifetime

2. Creating your own websites

> My job doesn't really require me to, but I do often have to look through clients' code to ensure certain requirements are met, such as accessibility guidelines or creating suitable SEO information for the site. I do have knowledge of how the web languages work.

3. Existing website builders

> The company that I work for provide a website builder as part of one of the pricing plans, but I'm not involved in its programming. It functions fine, but there are a lot of things that I would change about it.

4. Ideal features

> The visual aspect of any program is key - if it doesn't look good, is too cluttered, or doesn't look easy to understand, it will put people off using it. Nailing the design is an important aspect of any project, and you should make sure to prioritise it. Another thing to consider is site speed - if it takes forever to update the website, or it lags when you move things around, it will make people not want to use it. Other than that, you've got the other customisation features, organising objects, styles, content management, SEO customisability, and so on.

5. Programming methods

> I'd suggest using Flask for this project, as you are already quite used to python, and it is easy to set up. The go to frontend languages would be HTML, CSS and JS. Testing isolated features throughout will make the development faster, and make it easier to link components together.

6. Other comments

> Make sure you talk to your other stakeholders throughout to ensure that the final project will be something they'll want to use - their input can be vital.

David, Barber Shop Owner, Tech Enthusiast

1. Technical experience

> I've done a lot in terms of technical stuff - I program a lot in my spare time to help with tasks, I have a pretty nice computer setup, a couple of raspberry pies setup around my house to run automated processes, and I'm big into collecting older tech.

2. Creating your own websites

> I've used HTML before. I wouldn't say I'm the best at it, but I know how it works. As for creating websites, I've not gone that far.

3. Existing website builders

> I tend to stay away from automated solutions to technical programming, especially online ones, where subscriptions and paywalls are a massive component of the industry. I'm beginning to look into creating a website for my shop, but I'm cautious about using website builders and may go down other avenues.

4. Ideal features

> The design is the most important bit for me - I should be able to create the entire site to look exactly how I want it, and then be able to release it to the public. Making sure that everything looks how it should, that precision is a big importance to me.

5. Programming methods

> You should definitely look at how other services have created their versions before, as it can give you inspiration and help you work out how you want to approach the construction of it. Make sure to be constantly testing and looking for feedback - I'd be happy to help with that.

Analysis from stakeholders

After gathering feedback from the stakeholders, it became clear that there are several key features that they wanted from a website builder:

- Easy to use and user-friendly interface
- Ability to edit an existing site quickly
- Ability to invite others to collaborate on the site
- Import and export sites
- Enough customizability to make the website look as desired
- Styling customizability with global and individual element options
- Custom CSS and JS options

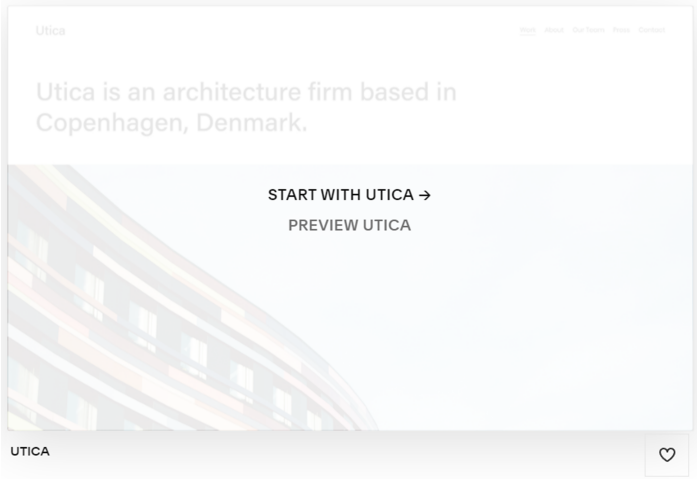
- Various templates with tags and filters
- Accessibility features, including automatic ARIA tags and contrast ratio checks for colour
- Support for screen readers
- Ability to change colour schemes to match brand identity
- Sign-up forms and privacy policy options
- Integration with social media feeds

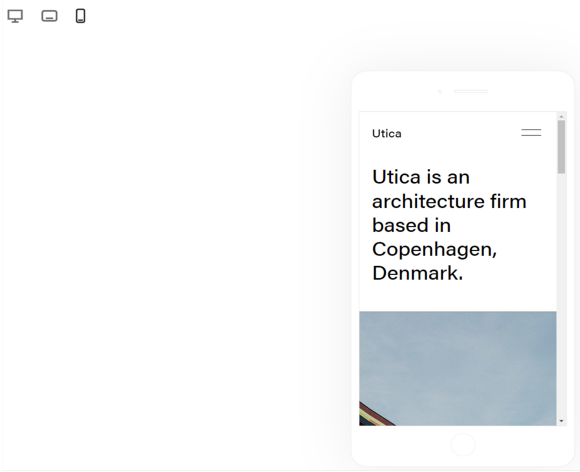
It's also worth noting that some stakeholders expressed a willingness to help with the implementation of certain features, so it may be beneficial to involve them in the development process. Additionally, the website builder should aim to be basic enough for those with minimal technical experience to use, while also providing enough features and customizability for those with more technical experience to make use of.

Research

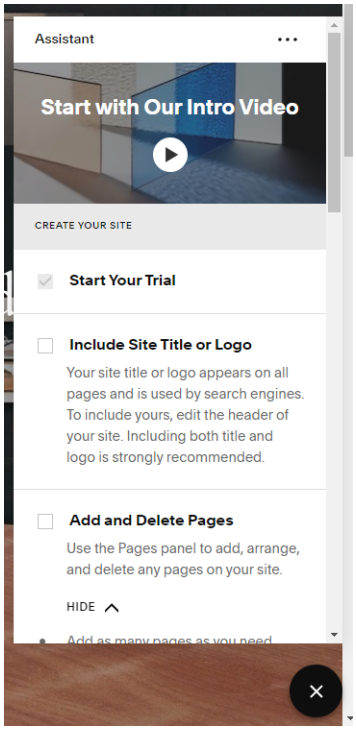
Existing solution - Squarespace

Their template list gives the option to preview the website, with all its functionality on a separate page. They allow their users to view it in different sizes as well.

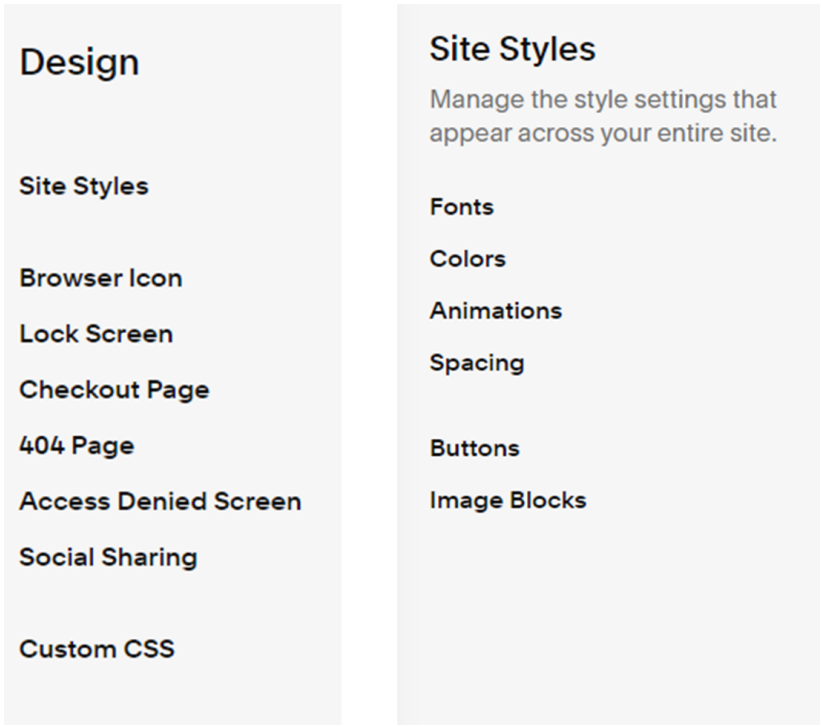




When first editing the site, Squarespace offers an assistant with some basic first steps to creating the website, making it easier for the client to understand how the editor works and how to use it effectively.

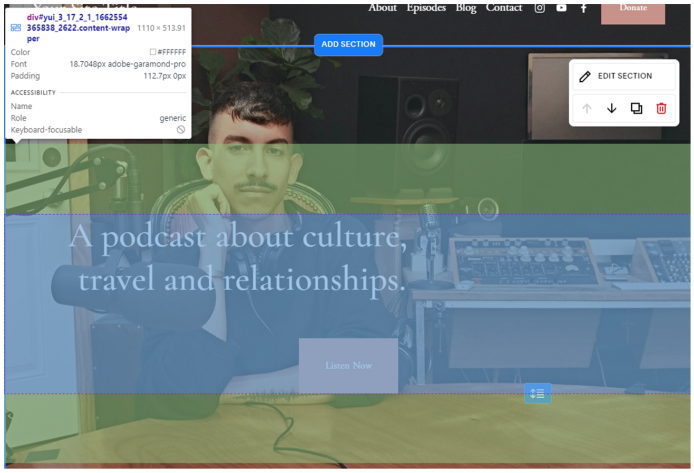


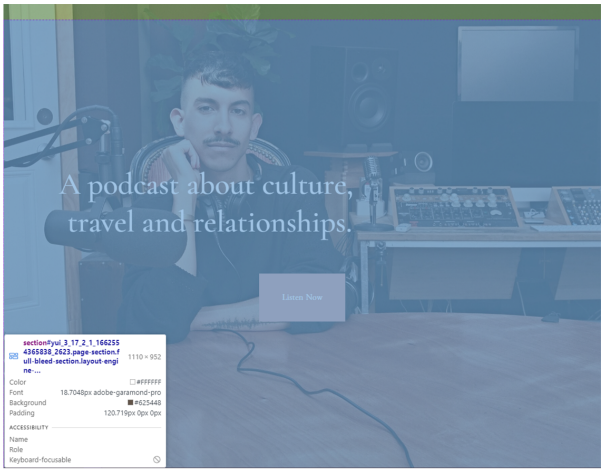
Their design options include styles, browser icons, 404 pages, and custom CSS. They can change the fonts, colour scheme, global animations, spacing, and default styles for certain widgets.



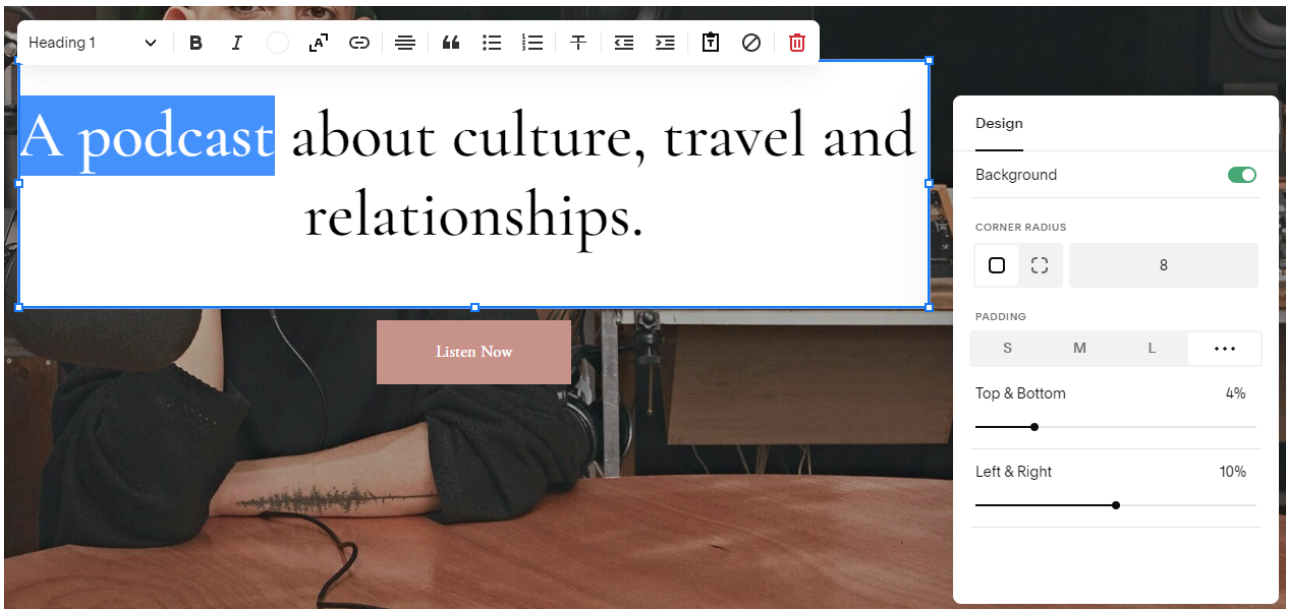
Their editor works in the conventional way of a grid-based system, where the user can place elements anywhere on the grid. It will then assign the item the style property `grid-area:row-start/col-start/row-end/col-end` or `grid-area:y/x/height/width` to define the position of the element. They have different attributes for different screen sizes, and the user can edit both styles separately by switching between laptop and phone modes.

The website is split into sections, where each section contains a content wrapper with the grid positioning system inside.





Selecting text gives the user a popup that displays the text formatting options. Whenever the user clicks on an element, they get a different popup that displays its design options.



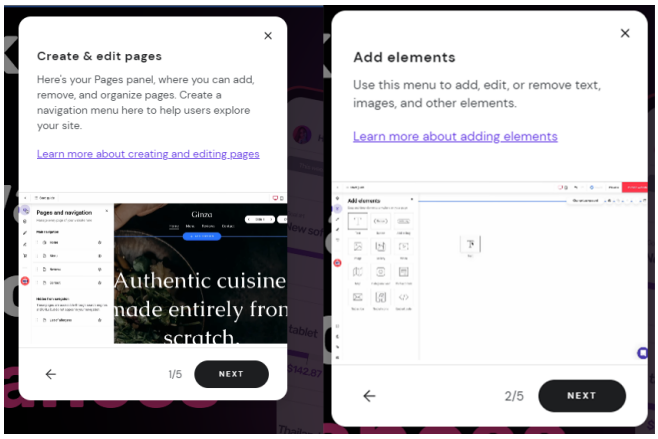
Parts I can apply to my project

This solution will have the option to preview and use templates similarly to how Squarespace does it, along with their grid positioning system, which is, for lack of a better phrase, an "industry standard." There will also be a similar formatting option setup, but it will be docked on the right-hand side with all the formatting settings in the same place.

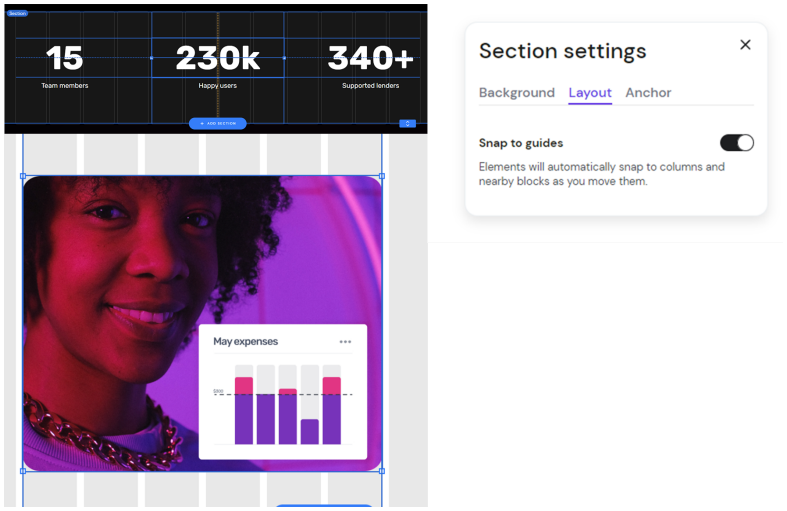
Squarespace also offers a variety of features such as e-commerce integration, SEO tools, analytics, 24/7 customer support, and a Content Management System (CMS) that allows users to update and manage their website's content easily. Although these features are helpful in a website builder, I intend leave them out of the solution's first release due to the timeframe for it and how long these features would take to make. This solution would have a CMS, but not at the scale or capability of Squarespace.

Existing solution - Zyro

Zyro also uses an assistant to help users understand how to use their editor.



Zyro has two ways of positioning objects; one is very similar to the way Squarespace does, with a grid positioning system, and the second is a smart layout. It instead uses only columns to position, and the elements can be moved up and down said columns freely and snap to other elements, like how many editors like photoshop might do. The user can toggle the snapping to other elements in section settings.



Something else Zyro does is have all of their style attributes defined in one class, which relies on variables such as `--grid-row`, `--m-grid-column`, and `--element-width` that are defined in `element.style` (the style attribute of the HTML object), which have been inserted with JavaScript.

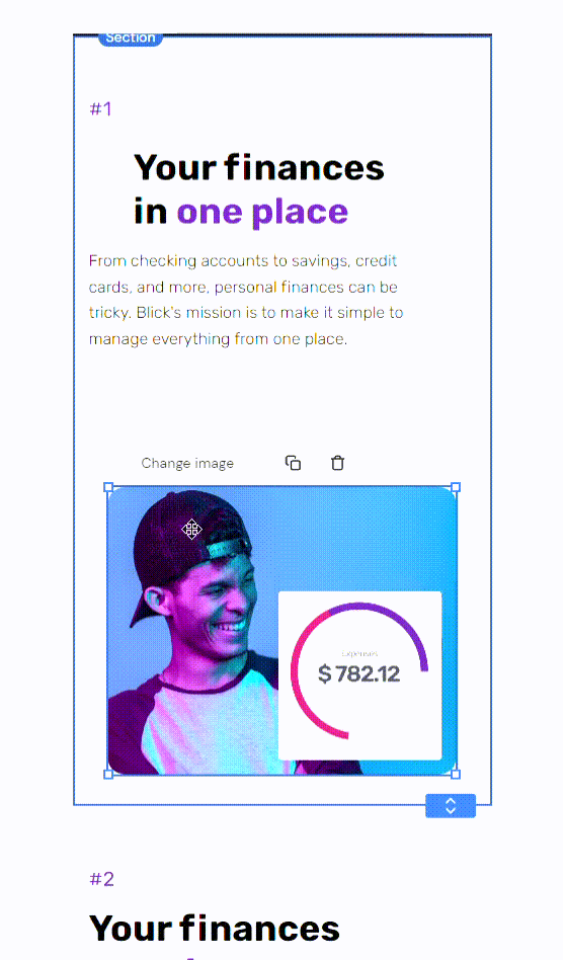
The CSS class with all of the variable references:

```
.layout-element {
  position: relative;
  left: var(--left);
  z-index: var(--z-index);
  display: grid;
  grid-row: var(--grid-row);
  grid-column: var(--grid-column);
  width: var(--element-width, 100%);
  height: var(--element-height, 100%);
  text-align: var(--text);
}
```

The HTML style attribute with all of the variable declarations in it:

```
element.style {
  --text: left;
  --align: flex-start;
  --justify: flex-start;
  --m-element-margin: 0;
  --z-index: 4;
  --grid-row: 3/4;
  --grid-column: 3/6;
  --m-grid-row: 2/3;
  --m-grid-column: 1/4;
}
```

Their image resizing system is good. It uses the `object-fit: cover` property in the style of the image and changes the width and height attributes when being dragged, as explained later.



When moving elements around the layout, Zyro adds four variables to the element, `top`, `left`, `width`, and `height`, which they use to render the element positioning while you are moving it. When the user releases the element, these values are removed. This positioning will be done in JavaScript by taking the position of the cursor when the user clicks on the element, getting the element's position when they click on it, and then offsetting the element's position by the amount they move the cursor. Then, when the user releases the cursor, it runs the code to calculate the new grid positioning of the element. They also have a max width for desktop mode, where the element cannot be moved further.

Parts I can apply to my project

Like Squarespace, Zyro uses a grid positioning system that will be implemented in this project. Their system for moving elements around with temporary `element.style` declarations is also a good way of implementing it that will be used as well.

Zyro also includes a built-in AI-based website creation tool called Zyro AI Writer, which can help users easily create website content. Similar to the Squarespace e-commerce system, this will not fit into the timeframe of the initial release of this solution, so it will not be implemented.

Key features of the solution

The proposed solution is a multi-user, web-based program that is designed to make website creation easier for a diverse range of users, including individuals, small businesses, freelancers, and non-profit organizations. The key features of the solution are outlined below:

- Grid-based, drag-and-drop system: The system is based on a grid layout that makes it easy for users to place and arrange various elements on their pages. The drag-and-drop feature allows users to add elements to their pages and reposition them as desired.
- Pre-defined template elements: The solution comes with a wide range of pre-defined template elements that users can use to create their websites quickly. These elements can be customised to match the user's needs.
- Customisable styles: Users can customise the style of their website by changing the font, colours, background, and other design elements. This feature allows users to create unique designs that match their brand or personal preferences.
- Organised pages: The solution provides users with tools to organise their pages in a way that makes sense to them. This feature enables users to create a clear, logical structure for their websites, which makes it easy for visitors to navigate.
- Library of pre-defined templates: The solution comes with a library of pre-defined templates for widgets such as text, buttons, or links. These templates are designed to save users time and effort by providing them with ready-made elements that they can use on their pages.
- Control of styling for each element: Users have complete control over the styling of each element on their pages. This feature allows users to create visually appealing designs that match their brand or personal preferences.

The overall aim of the solution is to provide an easy-to-use, intuitive system that has a low learning curve, making it accessible to anyone who wants to create a website. The user interface is designed to be visually appealing and straightforward, with a minimal cognitive load, enabling users to focus on designing their websites rather than struggling with technical details. The solution is designed to be flexible and adaptable to suit a wide range of needs, from simple personal sites to more complex business sites.

Limitations of the solution

The main limitation is that, as a server-side application, the user will always need an active internet connection to access it, and if the server goes down, there will be no way of using the program.

Hardware and Software Requirements

Hardware Requirements

The client will need a computer capable of accessing the internet.

In terms of the server side, hardware requirements may include the following:

- A server to host the website builder application and handle user requests. This server could be physical or virtual and run on any operating system that can also run Python (as it is the language I will be using to program the backend).
- The server will require a CPU with enough processing power: it may need a high number of cores and a fast clock speed to handle high traffic and many concurrent users.
- The server will need enough memory (RAM) to handle the requests and processes of the application.
- The server will require sufficient storage to store the website builder application and the images and files uploaded by the users to the CMS.
- The server will need a high-speed network connection to handle incoming and outgoing traffic between itself and the users. The faster the connection, the better the user's experience will be.

In addition to the above hardware requirements, it should be designed with scalability in mind so it can be adjusted if the application size or the number of users increases.

Software Requirements

The client will need a JavaScript-compatible web browser and an active internet connection.

For the server, software requirements may include the following:

- The operating system used on the server should be able to run Python alongside other software listed below.
- Python - it will need to be able to run Python in order to host the website builder.
- A file structure system to store the code and assets for the website builder.
- A database or file structure system to manage and store the data and media uploaded by the users to the CMS.
- Firewall software to monitor incoming and outgoing traffic and stop potential threats from corrupting the system.
- Load balancer software may be required when the user base increases to manage server resources and route traffic to available servers.

Success Criteria

Essential Features

- Login system
- the ability to view the password with the all-seeing eye
- Signup fields to be name, email, username, and two passwords to make sure they get it correct
- SQL database that stores user and site information
- Fully functional error checking on all fields as follows

> - All fields must not be empty
> - Name can have spaces and non-alphanumeric characters and must be longer than 2.
> - Email must be in an email format.
> - Username cannot have non-alphanumeric characters and must be longer than 2.
> - Password must be longer than 8.
> - The repeated password must be identical to password.
> - Email cannot already be in the database.
> - Username cannot already be in the database.

- The homepage, when there are no sites, displays a prompt to create a new site
- The homepage, when the user has created sites, lists all of them along with a "create new site" button
- When creating a site, the user will get the following options

> - Website Name: at least four characters, and any illegal characters are converted into dashes. The user is given a preview of what their site name will look like when it does not match the criteria.
> - Description: optional
> - Whether the site is public or private: determines who has access to the site URLs

- Sites can be accessed with the URL: `/<username>/<sitename>`, and, if public, can be viewed (but not edited) by anyone from this URL. If private, other users will be told this and redirected home.
- The site will have a config file, where it stores all of its global variables - mostly style choices - which have been selected when creating the site. These can also be edited at any time on the site's homepage.

> These variables include primary, secondary, accent, and grey colours, primary and secondary fonts, and animation types.

- File storage system to store user site files

- CMS system for users to be able to upload custom content
- The site page (`/<username>/<sitename>`) can be programmatically assigned due to the Python backend: it can take both parameters, search for them in the database, make sure that the current user has permissions, and display the appropriate site.
- On the site page, the user will get a preview of the website, along with customisability options for the website: the ability to edit the site, reorganise the site structure (which pages go where), edit site settings (such as default colours), and export the site.
- When editing the site, the organisation will look like this

  > - A navigation bar on the left that contains the options: "website pages", where you can navigate to a different page, "add section", where you can add another template section to the current page, "website styles", where you can change global settings such as fonts and colours, and "add element", where you can drag and drop individual elements into the canvas to edit.
  > - A central canvas where the actual web page can be previewed
  > - A popup modal that appears when the user needs to add an element or section
  > - A styling section on the right-hand side where the user can edit all of the styling properties for a selected element

- The central canvas will import the raw HTML and CSS files from the server and rely on data tags in the HTML element to understand what does what and how to edit it.
- Whenever a widget is selected, a box will be drawn around it, with the ability to resize it. The style menu on the right will also populate with style options for the selected element that can be changed in real-time and previewed when hovered over so that the user can easily understand what certain buttons will do.
- Whenever a widget is selected and held, an outline of the parent section's grid system is previewed, and the element can be moved around. It does this by tracking the cursor's position and relating that to the start position of the cursor on the widget (the anchor point) to render it in the correct place using left, right, top, and bottom CSS tags. When released, the widget will snap into the nearest grid space to where it was released. A similar thing happens when the user selects and holds one of the resize elements on the outline, where it tracks the cursor and then snaps into the closest grid space to resize it.
- The position parameters that are changed, as described above, are separate for the desktop and mobile views of the web page. Changing the position when the page is in desktop mode will not affect the position in mobile mode and vice versa.
- When a widget is right-clicked, it will show useful commands such as copy, paste, delete and duplicate.

Desirable Features

- Ability to (export and) import sites in a zip file so you can transfer them between sites, which is different to downloading a usable copy of the website. An export function may not be necessary as it is given in the site settings.
- An easy-to-use settings interface where the user can customise things such as public profile appearance and account settings.
- A list of predefined templates for sites when creating a site, that can be organised and filtered via relevant tags.
- When creating the site, the user can select options that allow them to change the default styling properties of the site.

> These will be the options for colour palettes and font families.

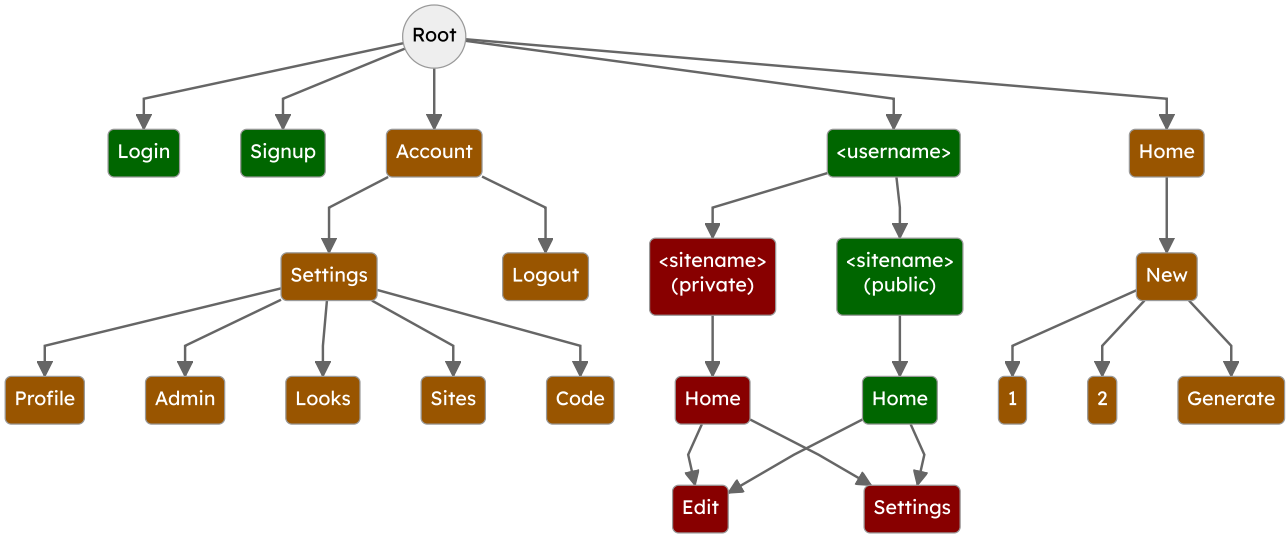
- The site owner can assign other users the ability to edit public or private sites, but there cannot be two people editing simultaneously.

> This would be achieved by modifying the database structure to have a linking table between the User and Site tables. This is explained later in the design section.

- Accessibility and support: Automatic ARIA tag assignment inside website builder, accessibility analysis tools to ensure that the current colour palette is acceptable, and adequate support for screen readers in created websites.
- Audience interaction features such as forms, social media feeds, and article posts.
- To export the site, the user will have two options that will be clearly defined in the UI
- They can download the site, which will download a zip file containing all the required HTML, CSS, and JavaScript code, so that they can unpack the archive and run the webpage by simply opening the HTML file.
- They can export the site, which will download a different zip file that contains all of the internal files that Kraken uses to run the editor for the page. This means the user can download backups and send their websites to others.

Design

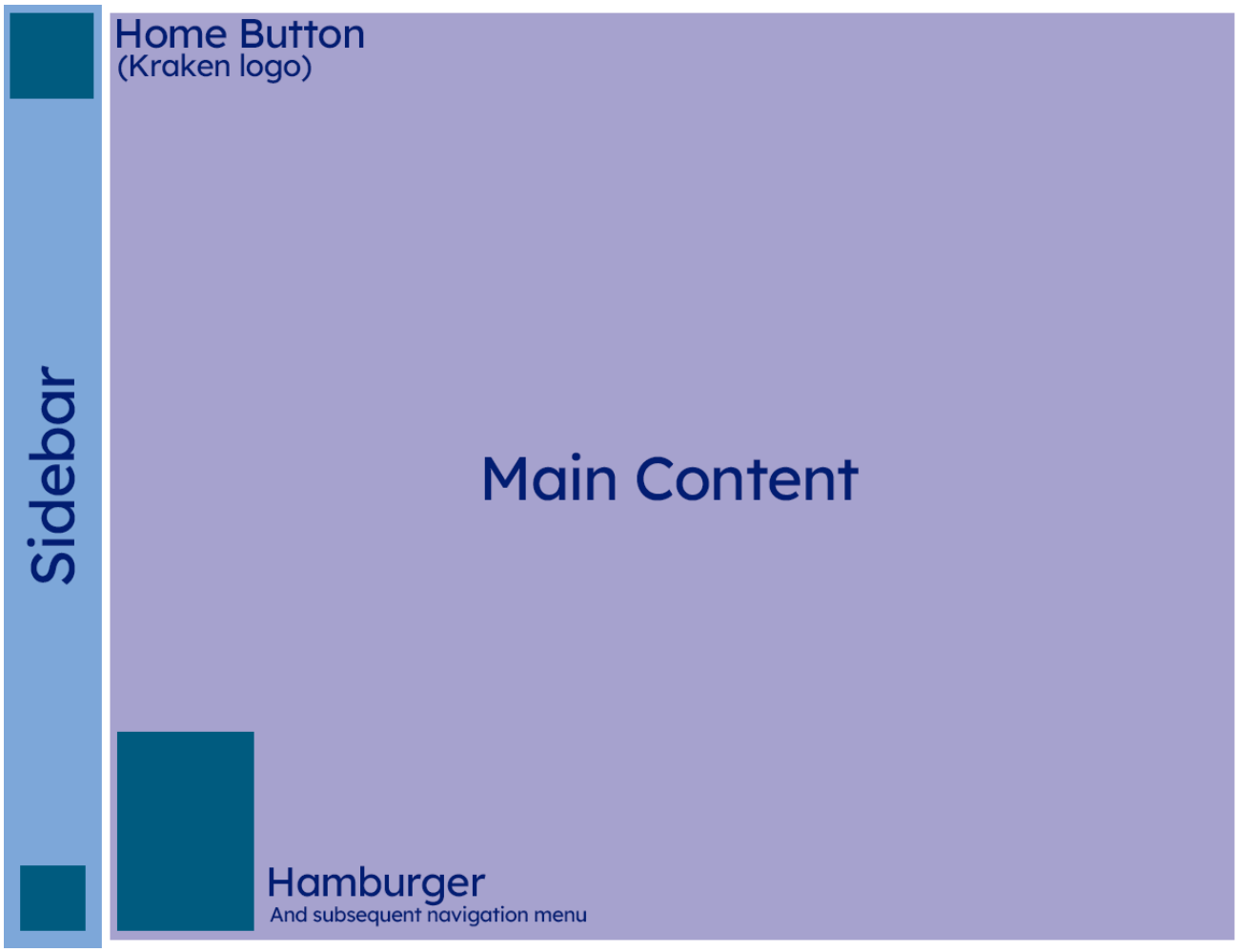
URL Navigation



The different nodes are colour coded based on the permission required to access those pages. If a user does not have the required permission, the user will be redirected to the nearest parent node that the user has access to. Most pages will redirect the user to the Login page if they are signed out. The colour coding is as such:

- Green: Any user can access this page and does not need to be signed in; it is public.
- Orange: The user needs to be logged in to access this page. This mainly relates to account-based pages such as the settings menu or creating a new site.
- Red: You need to be the owner of this website or have sufficient permissions granted by the owner. This only applies to the user websites that are set to private.

User Interface Design



This is the main "template" on which all pages are built, where the main content will be displayed inside. Defining this in a separate file beforehand (`/templates/base.html`) means that the website has a more unified feel and allows the user to go home and access the navigation menu. Defining it in a separate file also removes redundancy, as the code for it only appears once.

Home Button

> The Home button is placed in the top left-hand corner so that it is easy to find and matches how other websites do it; it conforms to the established design patterns and standardised expectations that the user will be familiar with.

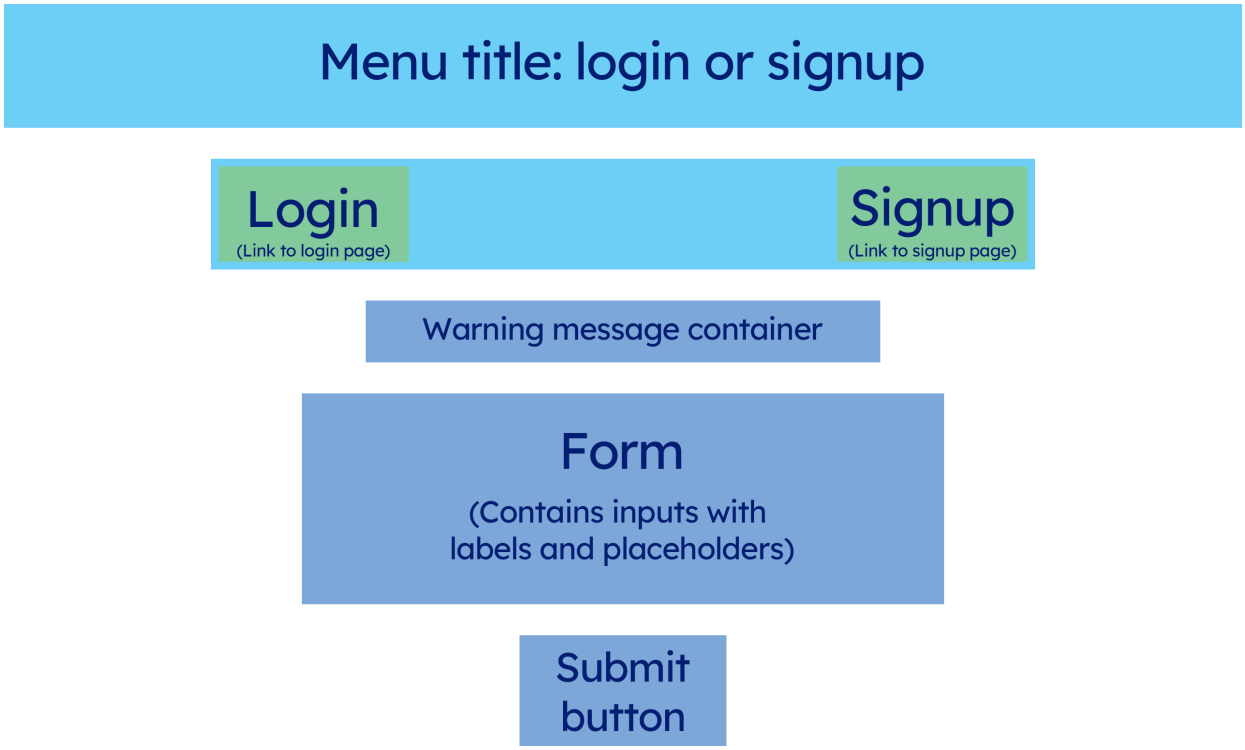
Hamburger

> The navigation menu has been hidden behind a hamburger at the bottom of the sidebar. This is because, due to the navbar being docked on the side, having lots of links on it would look messy and be hard to read. Therefore, the user can click the hamburger, and a modal will appear with all those links. Clicking the hamburger again, or anywhere else on the screen, will close the modal. The reason the navbar is on the side of the screen is not only a design choice, but it means that the website builder has more space vertically.

Main Content

> This area is where most of the interactive elements will be. These can be seen in the following diagrams.

Login and Signup Pages



This shows the layout of the login and signup pages. Built inside the main template, it contains the following:

- The header to tell the user what they are doing
- The buttons to toggle between login and sign up
- A warning message area for incorrect credentials or invalid input
- The form area, which is populated by inputs with labels next to them
- The submit button at the bottom

Homepage

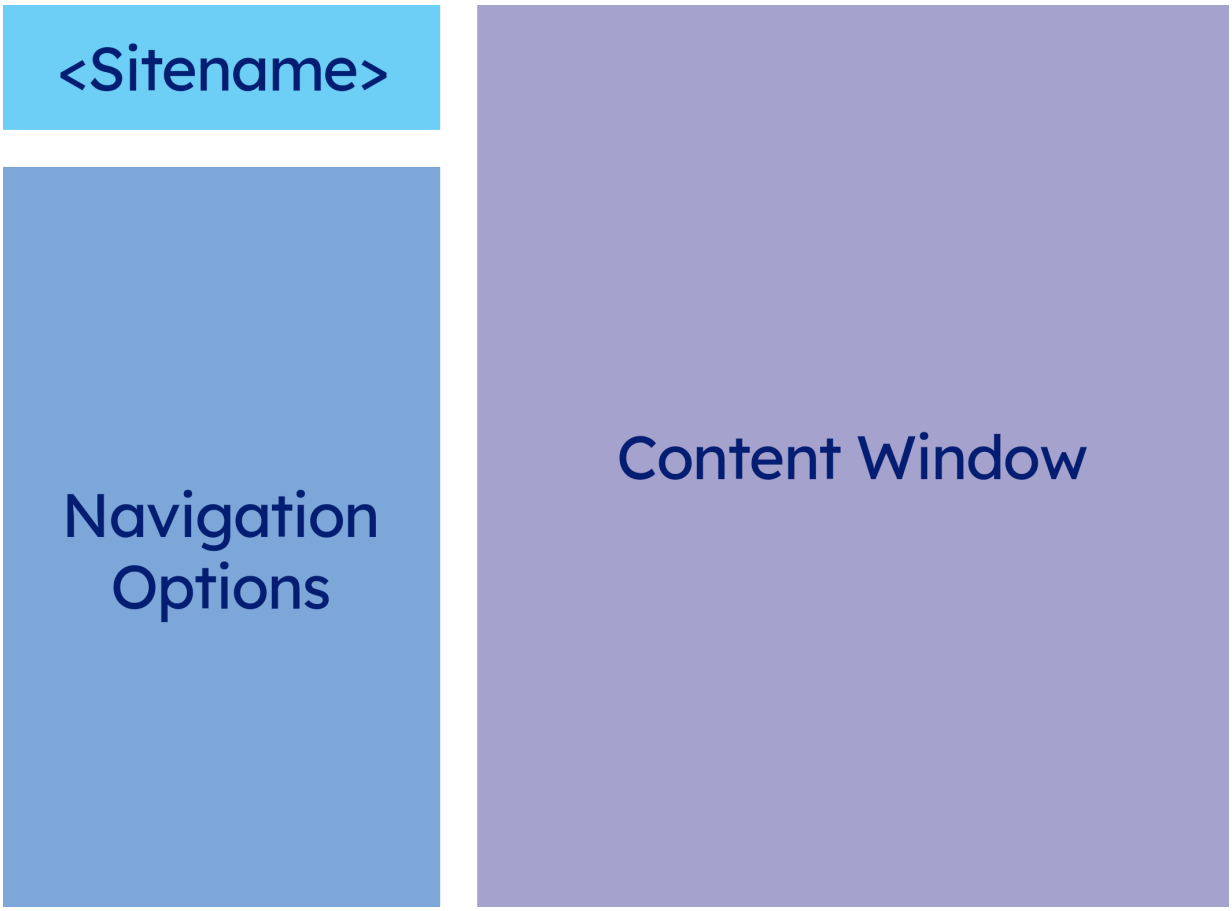
This shows the layout of the homepage once the user has logged in, if the user already has created a website. If the user has not made any websites yet, it will look similar to this but without the grid of their websites. Built inside the main template, it contains the following:

- The header, saying "Welcome, `<username>`" so that they know that it is the homepage
- A grid of all of their current sites.

> The grid will programmatically change the number of columns based on the display size. It contains square `div`s, each showing the title of the website, an icon informing the user as to whether it is public or private, and is coloured based on the primary colour of said website. The text colour redefines itself based on what the background colour of the `div` is, to make sure it is easy to read

- A create new site button with the same dimensions as the site `div`s, at the end of the grid layout.

Site Home



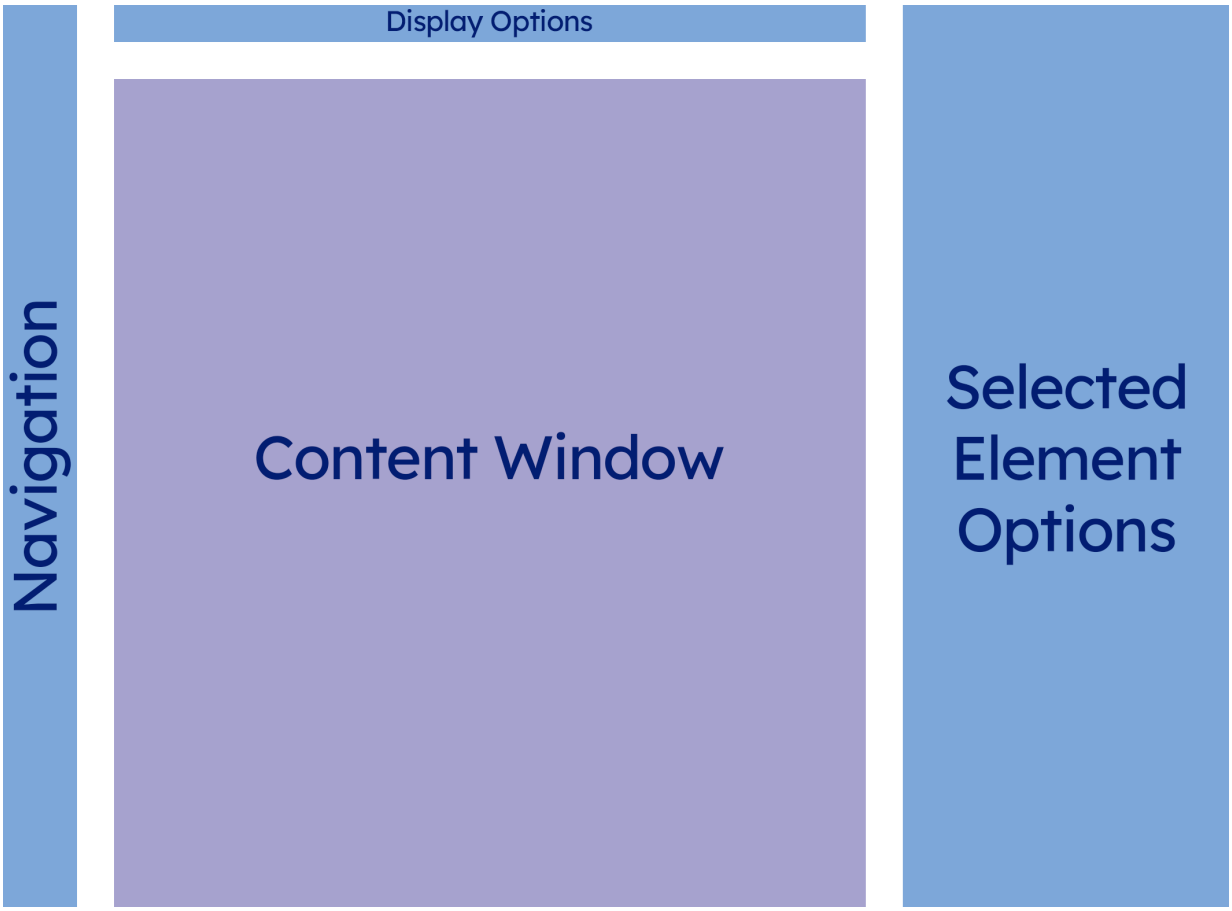
This shows the layout of the site page for the owner (when they visit `/<username>/<sitename>`). Built inside the main template, it contains the following:

- The header displays the site's name so that the user knows which site they are editing.
- Navigation options, a list of links that allow the user to navigate the menu system

> The links include:
>
> - Home, which will display a preview of the website in the content window
> - Edit site, which links to the editor
> - Site preferences, site styles, and site settings all open setting menus in the content window.

- The main content window will display content based on what is selected in the navigation options. By default, it will display a preview of the website but can also display setting menus as well.

Site Edit



This shows the layout of the site editor. Built inside the main template, it contains the following:

- The navigation bar, docked to the left, contains icons for different links. When hovered, these icons will display a label for what they will open.

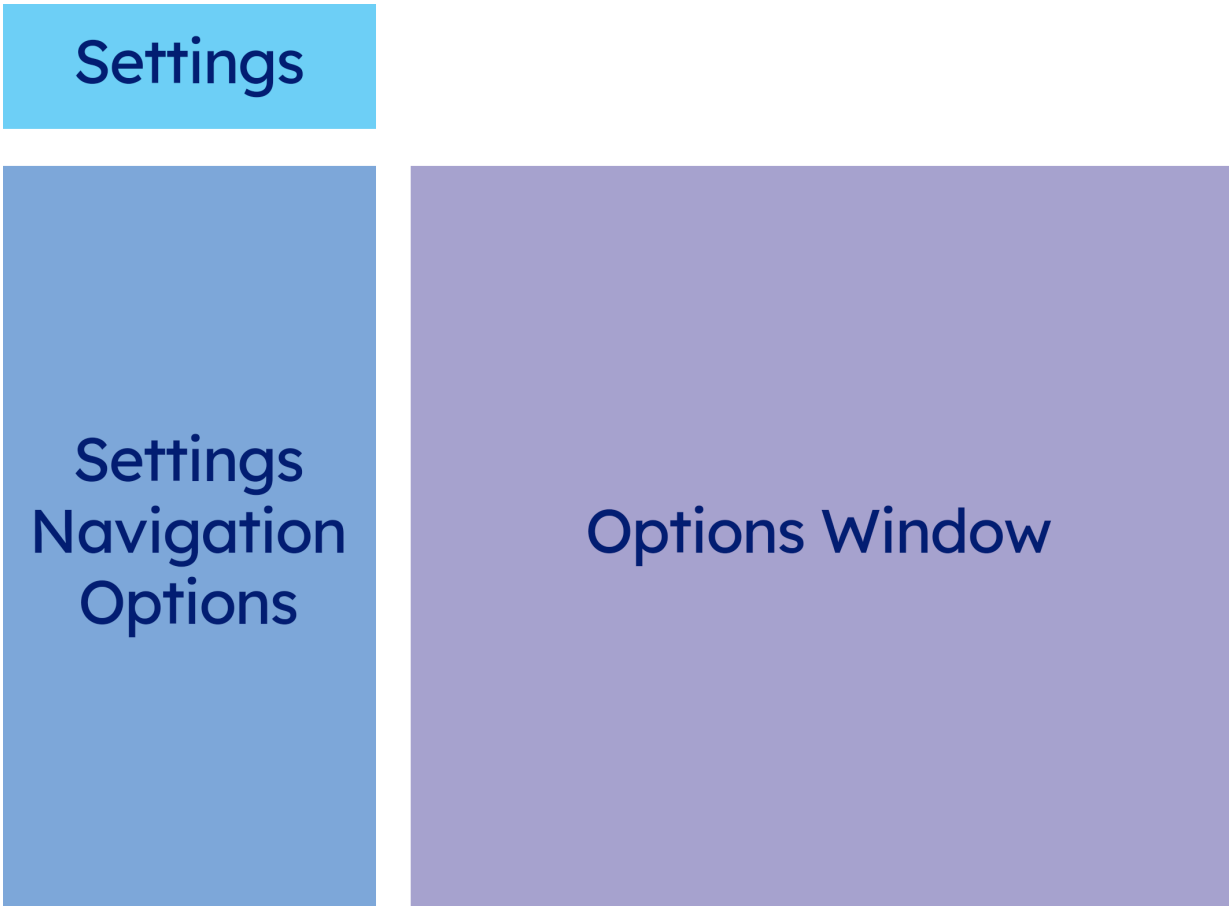
> Some of these options will be: Add section, add element, website pages, website styles, and website settings

- The display options bar, docked to the top, will contain some settings for how the editor is shown and are put here so the user can easily access them.

> These will include display size, switching between desktop, tablet, and mobile aspect ratios, previewing the website, and other settings.

- The selected element options, docked to the top, will contain quick-access site settings until an element is selected in the content window. When an element is selected, it will display style settings.
- The content window will display the site so the user can edit it. There is more information on the mechanics of this section in other parts of the report.

User Settings Pages



This shows the layout of the settings for the logged-in user. Built inside the main template, it contains the following:

- The header displays "Settings" to inform the user of what page they are on.
- Settings navigation options, a list of links that separate different categories of settings

> The links include:
>
> - Public Profile - settings that dictate how your profile will appear to other people, such as name, profile picture, and bio.
> - Account - account based settings, most of which have warnings next to them, such as change username, export account data, archive account and delete account.
> - Appearance & Accessibility - Accessibility settings such as high contrast mode, and utility settings such as tab preference for writing code.
> - My Websites - A table of the users websites. Contains a link to the website, how large the size of the website is, the name of the site, and its privacy status.
> - Custom Code & Elements - A beta option for storing custom code and elements, in a similar fashion to the above website menu.
> - Help and Documentation - Documentation and a help guide for the application.

- The options content window will display content based on what is selected in the settings navigation options.

Usability

The usability features I have considered ensure that the program is easy to use for as many users as possible, including those with accessibility issues. All of the buttons in the designs are large and easy to notice. When font selection or styling is used, previews for what the font looks like are shown so the user can clearly see what it will look like. This functionality is borrowed by other styling and positioning functions in the editor.

To make it easier for users to navigate and reduce cognitive load on users, the site will have a deliberately simple structure, with many features hidden behind modals or popup boxes. Elements such as the home button will be placed in conventional positions to make it easier for the user to find.

Accessibility

All colours will be checked to ensure a large enough contrast ratio so that people with colour deficiencies will see an adequate contrast between the text and the background. This includes elements such as colour pickers, where the label text that shows the hex code will change colour depending on the background to ensure that it is still readable. WebAIM, a website used to improve accessibility on the internet, will be used to ensure that there is enough contrast in the text. Furthermore, links in text blocks will also be checked to ensure that they have at least a 3:1 contrast ratio with the surrounding text and are visible enough. Quoted from WebAIM,

> "Often, these [accessibility features] promote overall usability, beyond people with disabilities. Everyone benefits from helpful illustrations, logically-organised content and intuitive navigation. Similarly, while users with disabilities need captions and transcripts, they can be helpful to anyone who uses multimedia in silent or noisy environments."

The basic accessibility requirements that are suggested, and that could apply to this project, include the following:

Requirement	Explanation
Provide equivalent alternative text	Provides text for non-text elements. It is especially helpful for people who are blind and rely on a screen reader to have the content of the website read to them.
Create logical document structure	Headings, lists, and other structural elements provide meaning and structure to web pages. They can also facilitate keyboard navigation within the page.

Requirement	Explanation
Ensure users can complete and submit all forms	Every form element (such as text fields, checkboxes, and dropdown lists) needs a programmatically-associated label. Some text may not be focused by tabbing through the form. Users must be able to submit the form.
Write links that make sense out of context	Every link should make sense when read out of context, as screen reader users may choose to read only the links on a web page.
Do not rely on colour alone to convey meaning	Colour can enhance comprehension but cannot alone convey meaning. That information may not be available to a person who is colour-blind and will be unavailable to screen reader users.
Make sure content is clearly written and easy to read	Write clearly, use clear fonts, and use headings and lists logically.
Design to standards	Valid and conventional HTML & CSS promote accessibility by making code more flexible and robust. It also means that screen readers can correctly interpret some website elements.

ARIA

ARIA (Accessible Rich Internet Applications) attributes will be used throughout the website to allow screen readers to navigate the website. These attributes can be used by assistive technologies, such as screen readers, to provide a more detailed and customised user experience. It is particularly useful for improving the accessibility of dynamic content and advanced user interface controls, such as those used in rich internet applications.

Stakeholder input

Website structure and backend

Flask

I have decided to use the Flask Python library as the backend for this website, as I have prior experience in using it, and it suits this project. It is well-documented online, relatively lightweight, and easy to use. Although it does not include as many built-in features as other libraries (such as Django), there are plenty of other Python libraries, such as `flask-login` and `flask-sqlalchemy`, that can add in all of the functionality that is missing from the framework.

Jinja

I have decided to use the Jinja2 template syntax for storing the HTML files, as it has in-built functionality with Flask via the `flask.render_template()` function. All of the HTML files used for the website will be stored in the `templates/` folder in the server directory.

Jinja is a template system built for Python and Flask (other frameworks have different template systems). It uses templates to reduce duplicated code and make it easier to develop. It enables logic operations in the template file, with functionality for `if`, `while`, `for`, and variable declaration and usage.

An example system of a Jinja file structure might look like this:

base.html

```
<html>
  <head>
    <title>Jinja Example</title>
  </head>
  <body>

    {% block content %}
    {% endblock %}

  </body>
</html>
```

itemlist.html

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

{% block content%}

<h1>A list of items</h1>
<ul>
{% for item in items %}
  <li>{{ item }}</li>
{% endfor %}

{% endblock %}
```

main.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    items = ["apple", "banana", "cherry"]
    return render_template("itemlist.html", items=items)
```

In `base.html`, you can see the `{% ... %}`, which states that this is a control statement. In this case, it defines that the `block` referenced as `content` is to be inserted here. Jinja can have multiple blocks with different names so that a child file can add multiple blocks of content in different places throughout the template.

In `itemlist.html`, you can see that it `extends base.html`, meaning that it is using `base.html` as a template and looking at that file to find where to insert the code inside `block content`. It also uses a `for` loop to programmatically add list items to the website, based on the list `items` imported in `main.py`, in the `render_template()` command. The `{{ ... }}` indicates that the contents are a Jinja expression.

HTML, CSS and JavaScript

As with many websites, this project will be built using HTML (via Jinja's template generation), CSS, and JavaScript. This is because they are all web standards and are therefore supported by all modern web browsers and devices (although devices are not much of an issue as it is designed to be run on a high-resolution landscape display). It will allow for SEO compatibility, is open source (and therefore free), is highly versatile, and can accomplish a lot. I am also very competent with HTML, CSS & JavaScript and have lots of experience programming with them.

Data storage

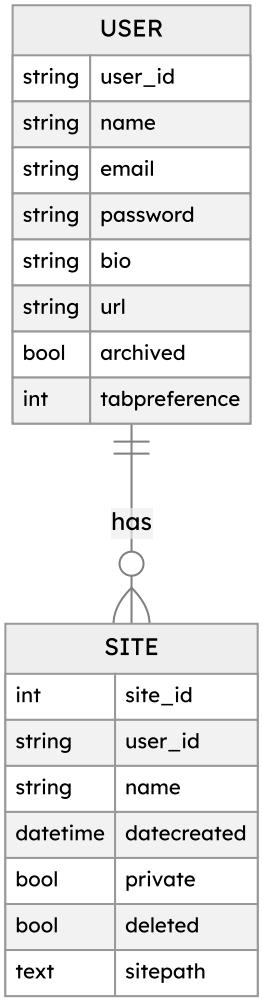
There will be two different methods of storing information for the website:

- The multi-user system, including the information about the users' sites, will be stored in an SQL database using the `flask_sqlalchemy` Python library so that it can easily be integrated into the Flask backend.
- The server will store the users' sites, including the HTML, CSS, and JavaScript code.

SQL database storage

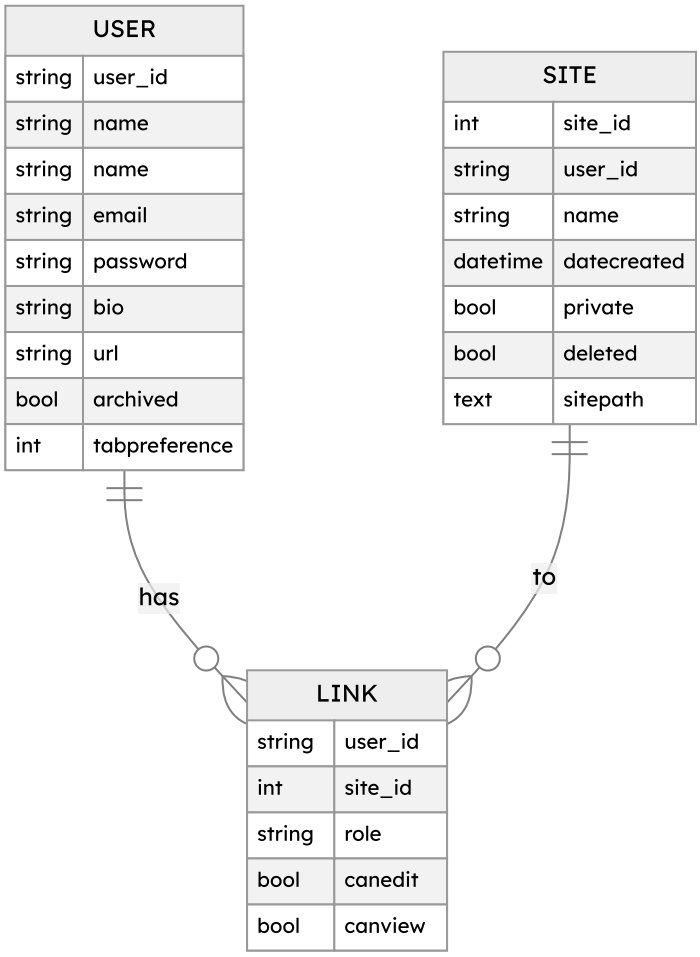
This project will use SQL to store the multi-user information as I have previous experience using SQL, so it will be easy to set up and use. It is also useful due to its entity-relationship capability, meaning it will be well-suited for storing information about users' sites. It also has a Python library that integrates into the current backend library that is being used, Flask. This means there will be less work, as most of the functionality needed is already built-in and tested.

This is the planned entity relationship diagram for the SQL database. It contains two entities, `USER` and `SITE`, connected with a one-to-many relationship with `user_id` being the foreign key in `SITE`.



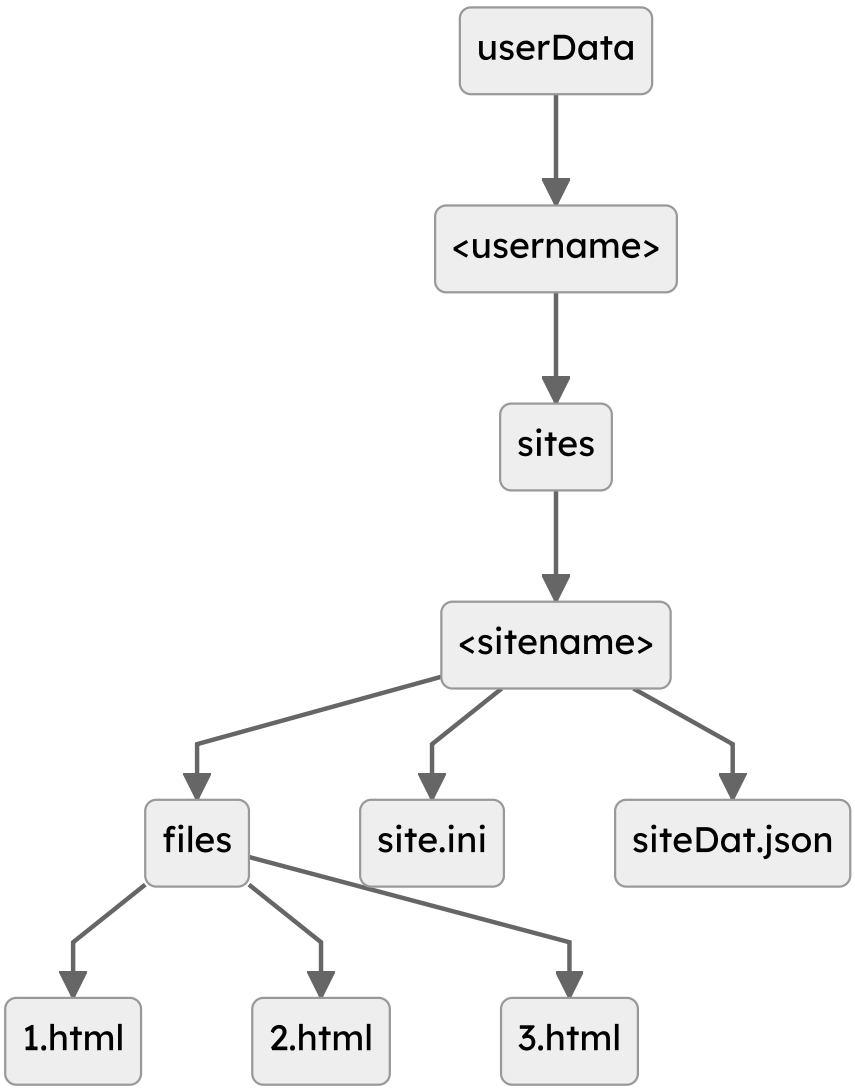
To incorporate a multi-user editing system for certain sites, the entity relationship diagram for the database will look like this. However, this may not be implemented due to time constraints.

It contains three entities, `USER`, `SITE`, and `LINK`. It is similar to the previous one, with a linking table added between the two original entities, allowing multiple users to edit multiple sites. Each `LINK` also contains information about the `USER`'s permissions for the `SITE`.



Server-side file storage

For the actual user website files, server-side storage will be used as it cannot be easily stored in SQL. It is all stored server-side so the user can access their files from any computer with an internet connection. The way I intend to store the site information is shown below:



The `username` and `sitename` folders will be named by the primary keys of the data in the SQL database to avoid duplicate folder names.

The `files` folder will contain all the HTML files, named sequentially, and any custom CSS and JavaScript files the user has added.

The `siteDat.json` file will contain all the information about the site file structure, referencing which page requires which HTML file in the `files` folder and which CSS and JavaScript code blocks need to be imported.

> The server will have a store of CSS and JavaScript that will format every template element and section.

The `site.ini` config file will contain all of the information about the site settings, theming, and preferences.

Algorithms

The main parts of this solution are using a SQL database to store information about the multi-user system, the UI design and interactivity, and the actual drag-and-drop editor, with the drag-and-drop editor being the most complex.

Drag-and-drop editor algorithms

The main things that the drag-and-drop editor should be able to do are:

- Display a resize box around a clicked element.
- Resize an element when its resize box is dragged.
- Preview a live display of a dragged element (moving with the cursor), along with the resize box rendering where the element will land when dropped (snapping to the grid).
- Display a set of styling features in the right-hand menu for a selected element or section.
- Display a set of options next to a selected element.
- Display a text editor for a selected element with editable text.
- Display options in the bottom corner of a selected section.
- Preview a live display of a dragged section, and renumber the sections into their new positions when dropped.

In the JavaScript code, each element will have a set of event listeners on them, defined by data tags in the HTML elements:

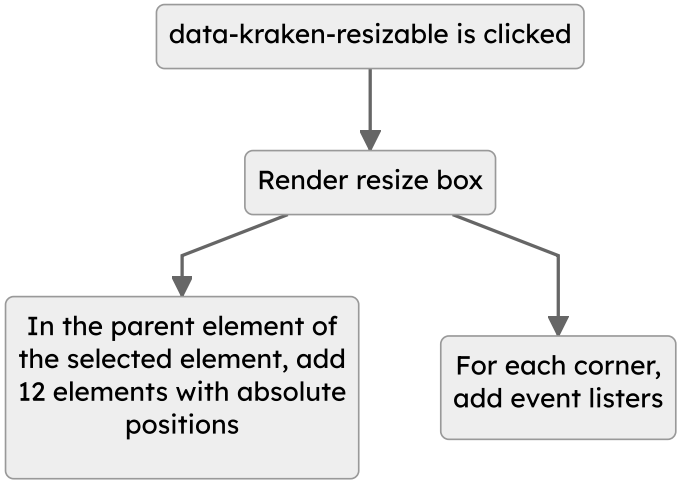
- `data-kraken-resizable`
- `data-kraken-draggable`
- `data-kraken-editable-text`
- `data-kraken-editable-style`
- `data-kraken-locked`

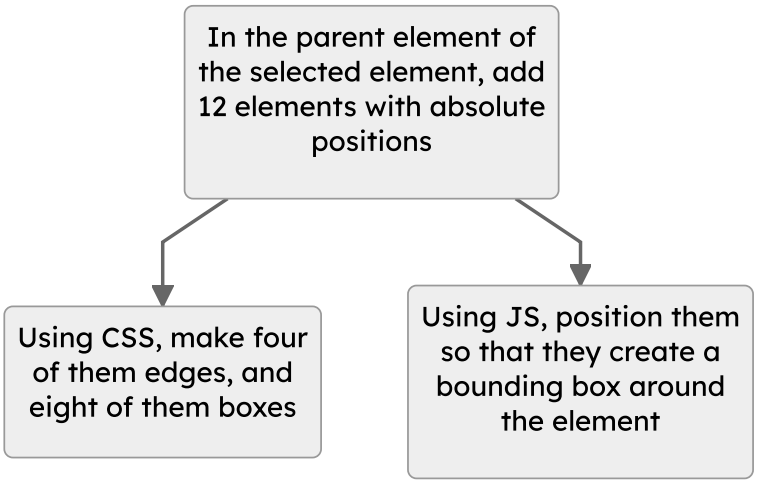
These will define which functionalities can be used for each element. For all of the below diagrams, if the element has the tag `data-kraken-locked`, it will only show a button next to the element/section to unlock it.

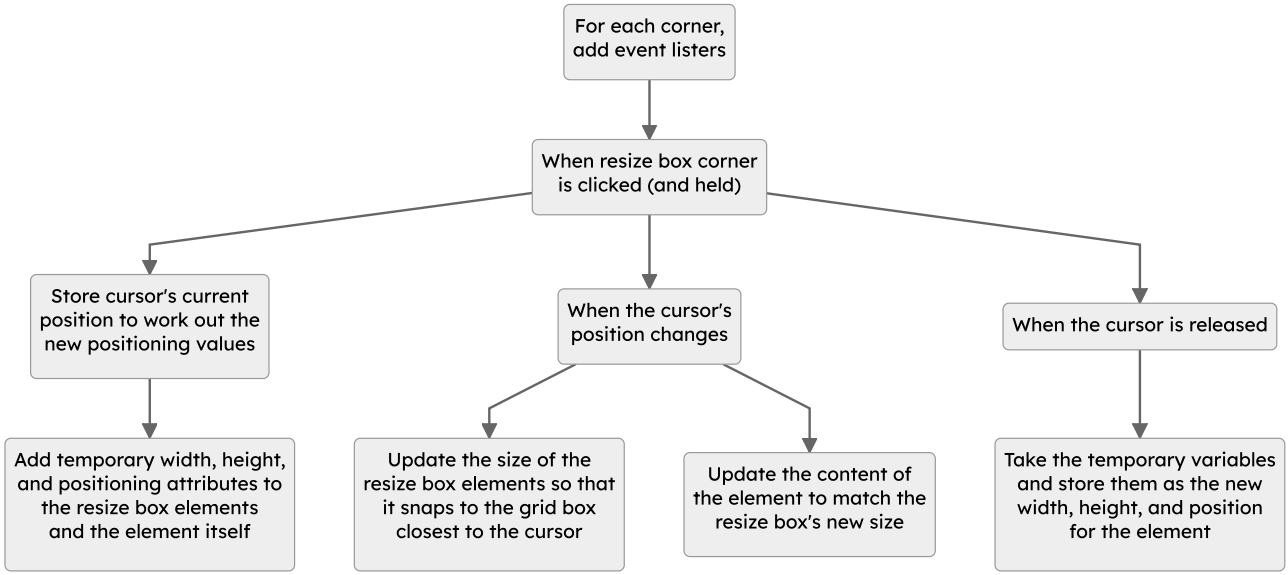
When an element is selected, depending on its function, it will be tagged with one of these attributes so that the JavaScript can easily edit it:

- `data-kraken-selected-text`
- `data-kraken-selected-style`
- `data-kraken-selected-resize`
- `data-kraken-selected-drag`

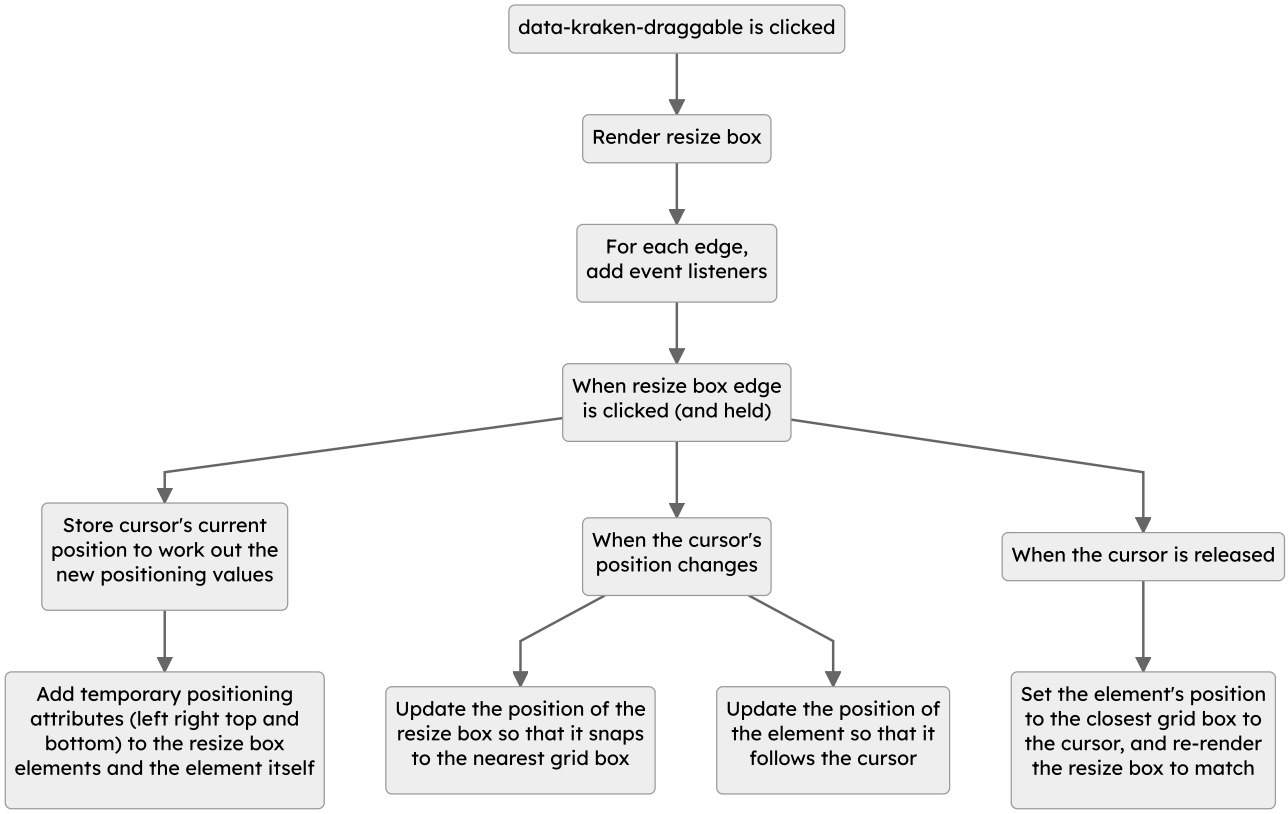
Resize box


data-kraken-resizable is clicked
Render resize box
In the parent element of the selected element, add 12 elements with absolute positions
For each corner, add event listers

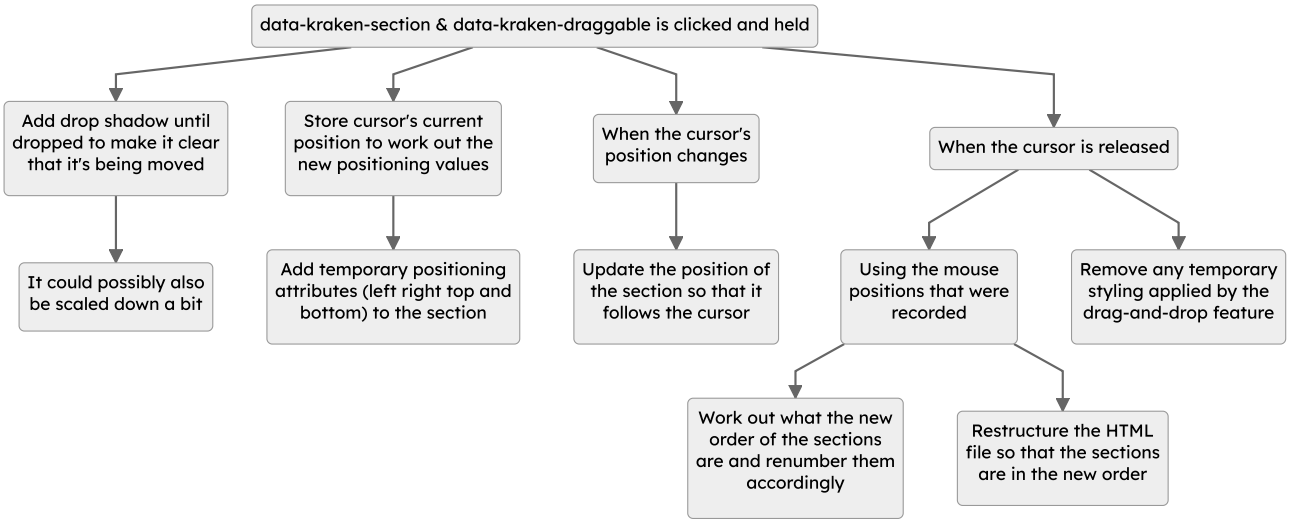

In the parent element of the selected element, add 12 elements with absolute positions
Using CSS, make four of them edges, and eight of them boxes
Using JS, position them so that they create a bounding box around the element


For each corner, add event listers
When resize box corner is clicked (and held)
Store cursor's current position to work out the new positioning values
When the cursor's position changes
When the cursor is released
Add temporary width, height, and positioning attributes to the resize box elements and the element itself
Update the size of the resize box elements so that it snaps to the grid box closest to the cursor
Update the content of the element to match the resize box's new size
Take the temporary variables and store them as the new width, height, and position for the element

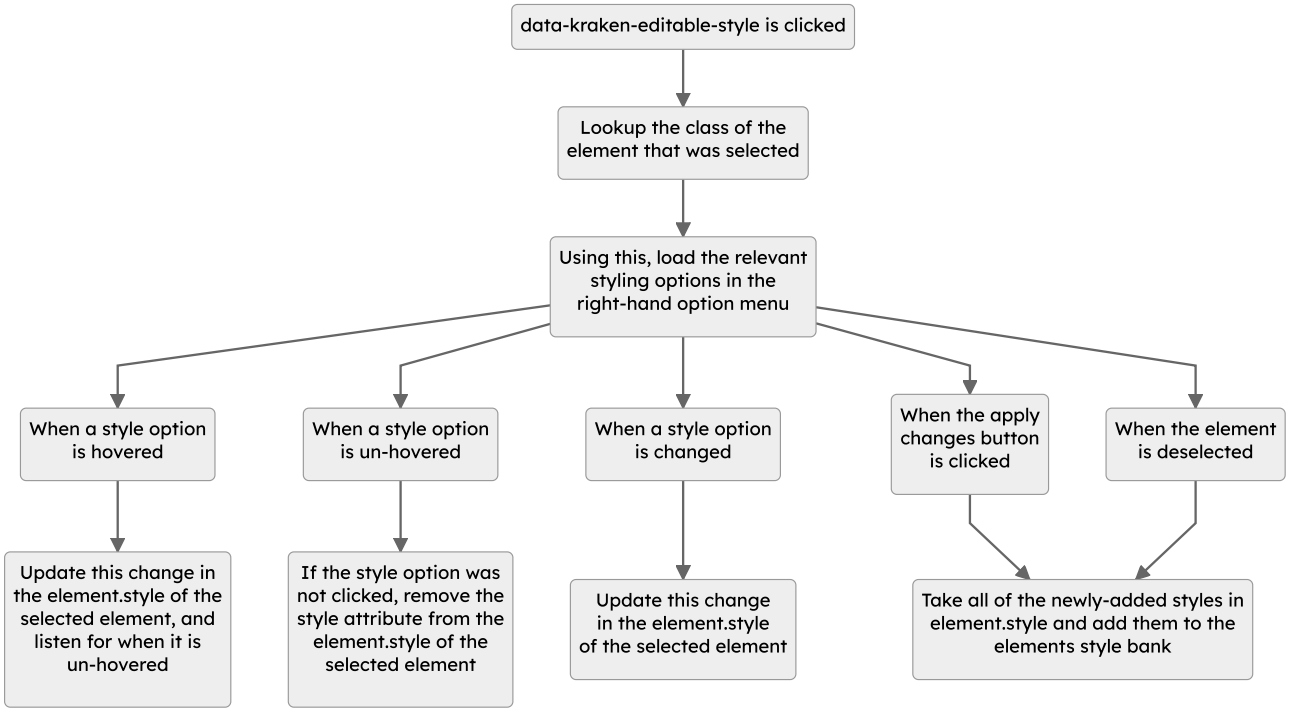
Dragging and dropping elements



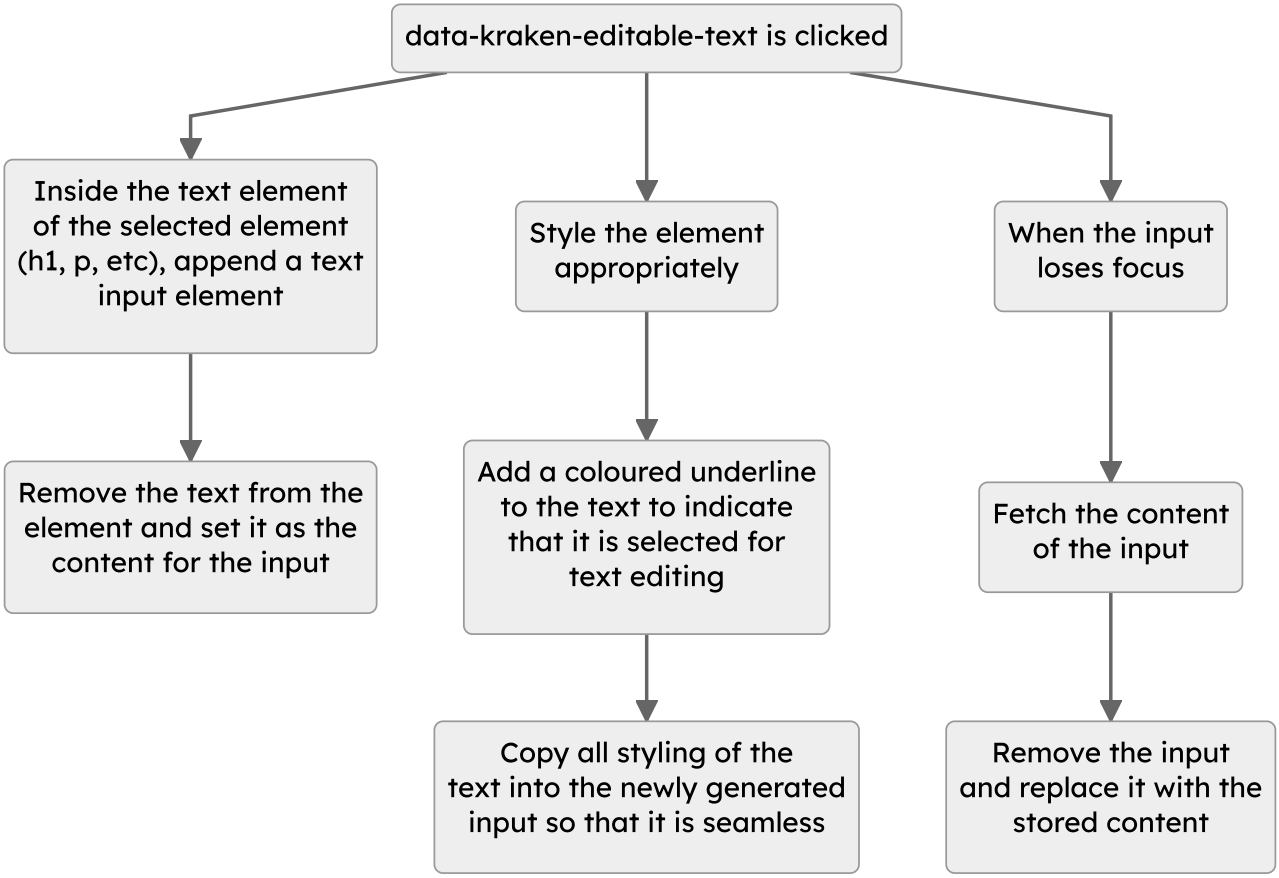
Dragging and dropping sections



Displaying element and section options



Displaying text editors



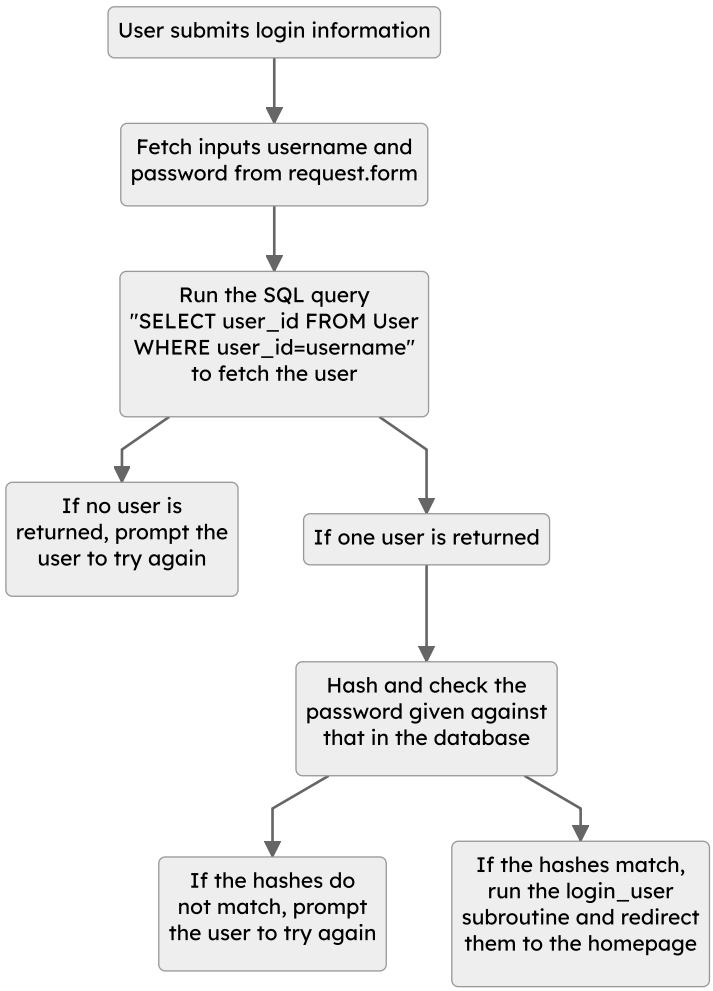
Multi-user system algorithms

A lot of the SQL and multi-user algorithms are handled by libraries that are being used, which means that function calls can be used, such as `login_user(user)` from the `flask_login` library or `user = self.User.query.filter_by(user_id=username).first()` that uses an inherited class in the `models.py` file to perform an SQL query. As such, it will reduce a lot of the programming work required, as I can call a function from a different module to do it for me.

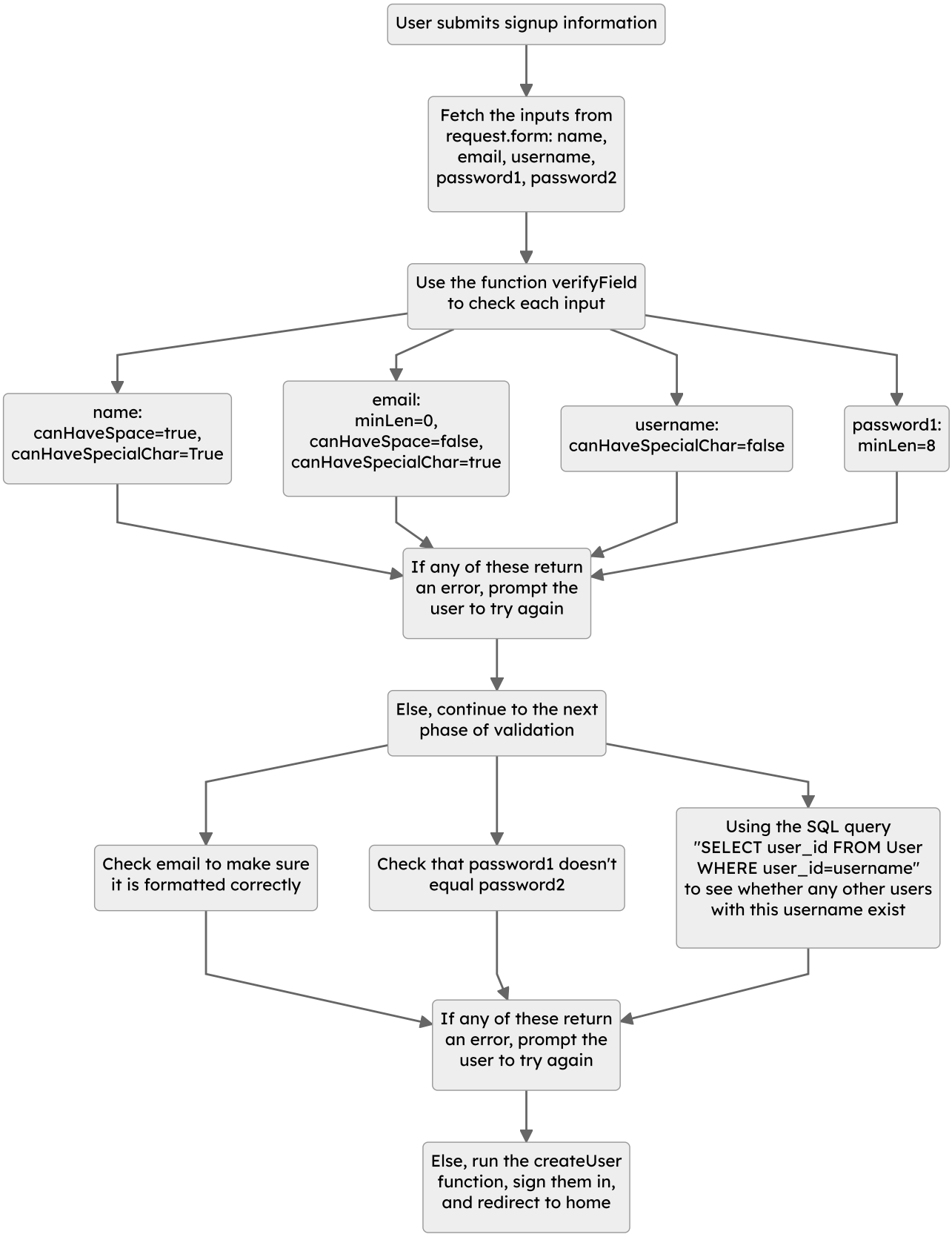
The main things that the multi-user system should be able to do are:

- Read and write to an SQL database that contains information on the users and their sites
- Use the database to allow the user to log in
- Use the database to allow a new user to sign up
- Verify the user's inputs to make sure they are valid
- Use the database to organise sites and permissions
- Use the database to store and create sites for the users
- Generate folders and files on the server to store user and site information
- Import and export sites that are stored in the database and on the server

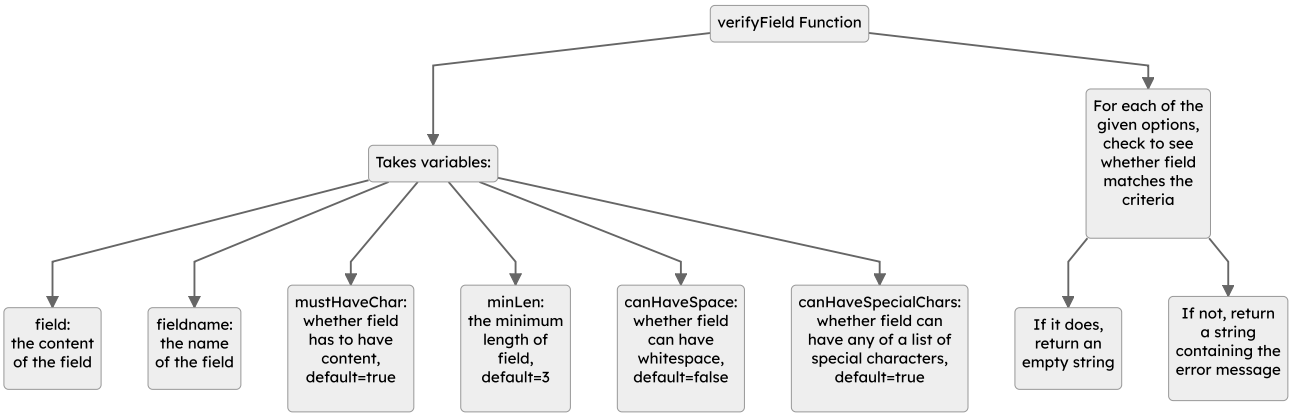
Login page



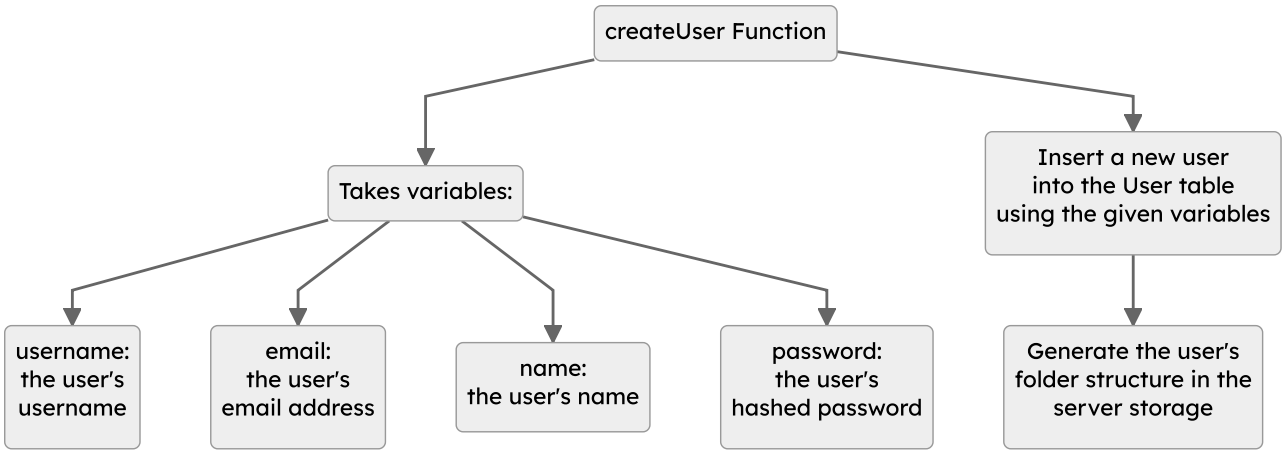
Signup page


User submits signup information
Fetch the inputs from request.form: name, email, username, password1, password2
Use the function verifyField to check each input
name: canHaveSpace=true, canHaveSpecialChar=True
email: minLen=0, canHaveSpace=false, canHaveSpecialChar=true
username: canHaveSpecialChar=false
password1: minLen=8
If any of these return an error, prompt the user to try again
Else, continue to the next phase of validation
Check email to make sure it is formatted correctly
Check that password1 doesn't equal password2
Using the SQL query "SELECT user_id FROM User WHERE user_id=username" to see whether any other users with this username exist
If any of these return an error, prompt the user to try again
Else, run the createUser function, sign them in, and redirect to home

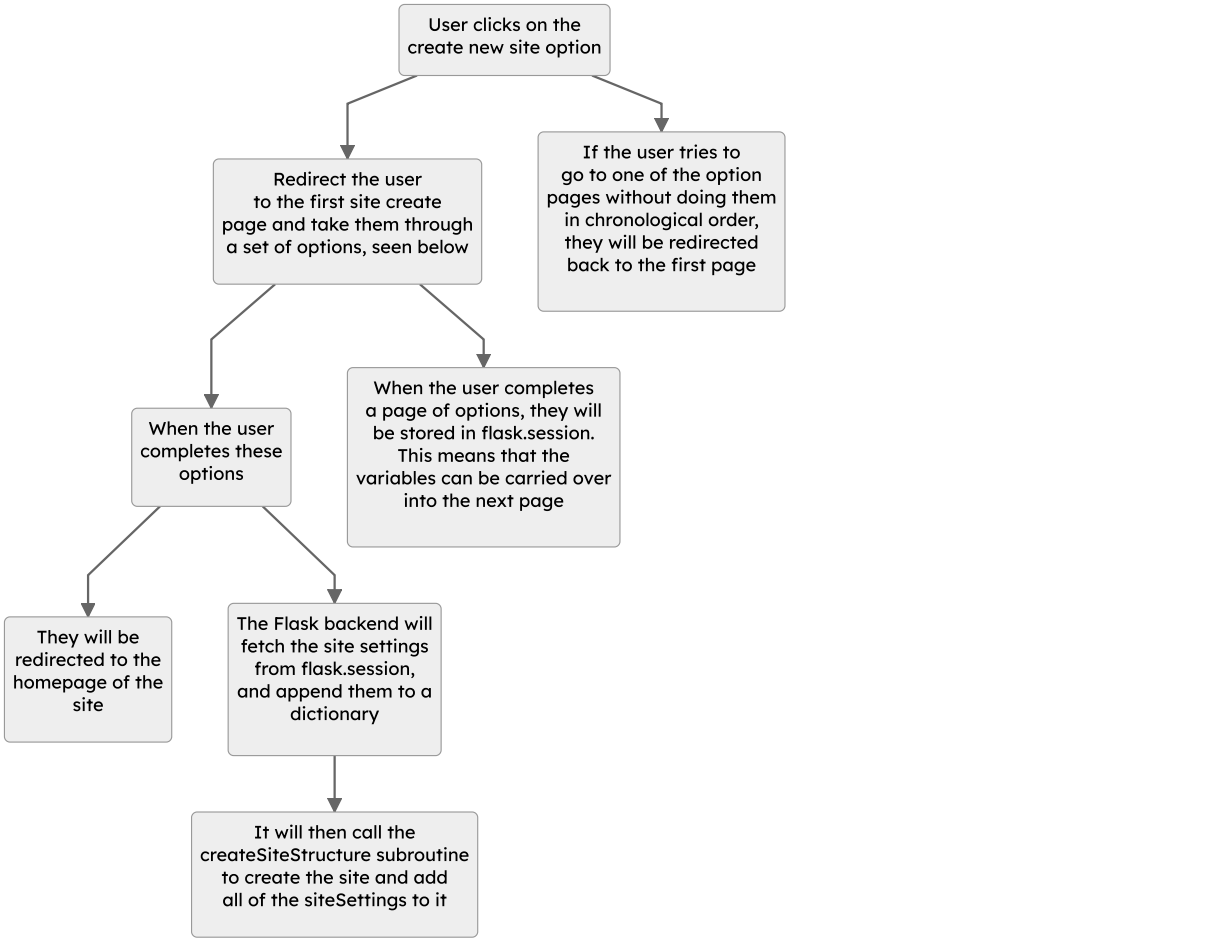
Field verification

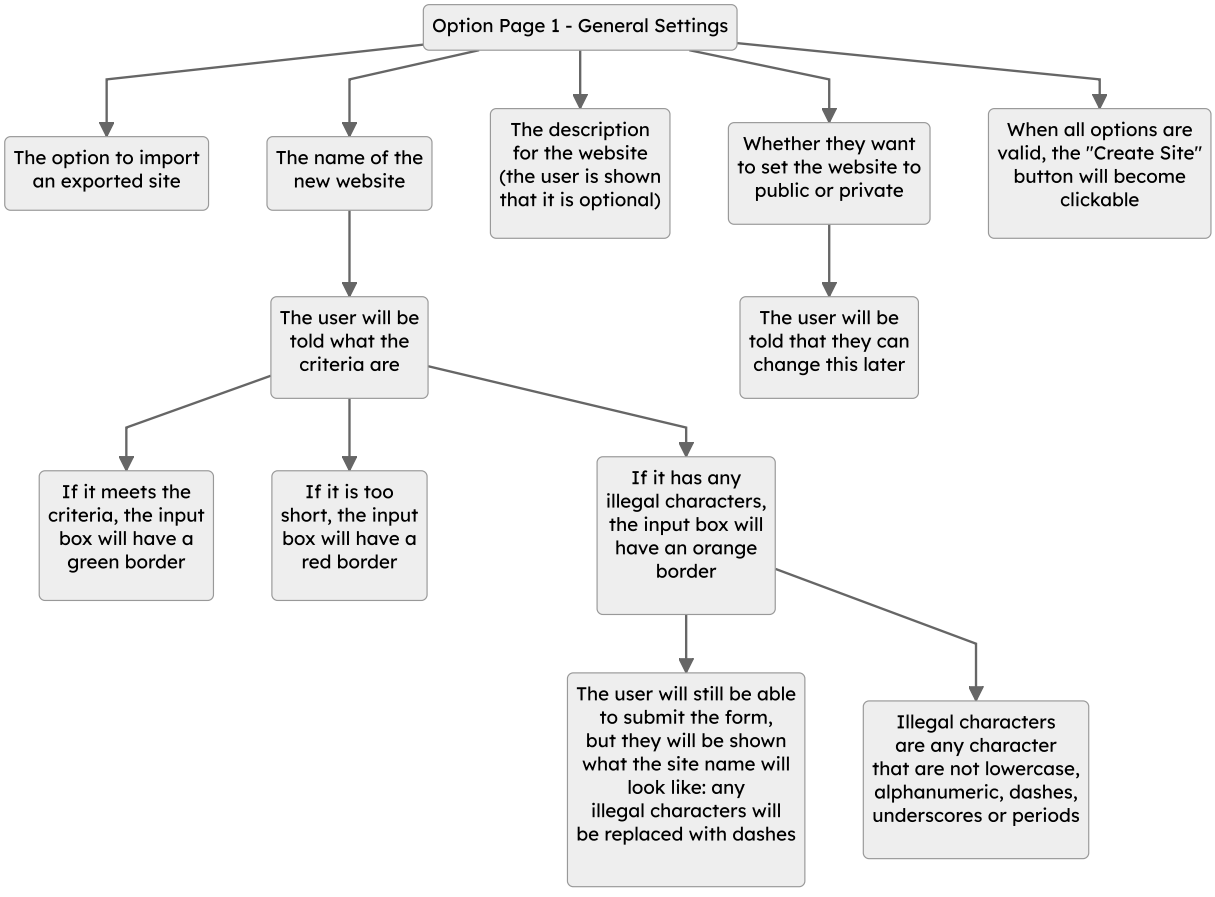


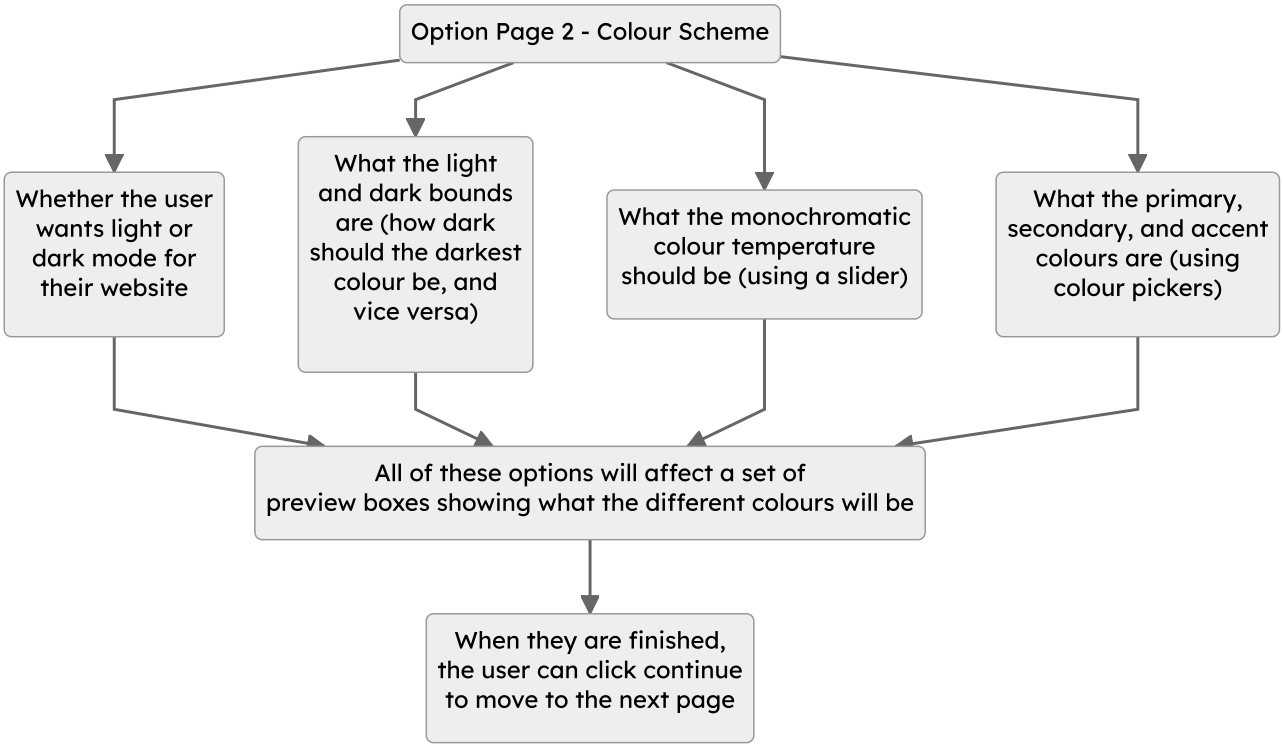
Creating a new user



Creating a new site






Option Page 2 - Colour Scheme
Whether the user wants light or dark mode for their website
What the light and dark bounds are (how dark should the darkest colour be, and vice versa)
What the monochromatic colour temperature should be (using a slider)
What the primary, secondary, and accent colours are (using colour pickers)
All of these options will affect a set of preview boxes showing what the different colours will be
When they are finished, the user can click continue to move to the next page

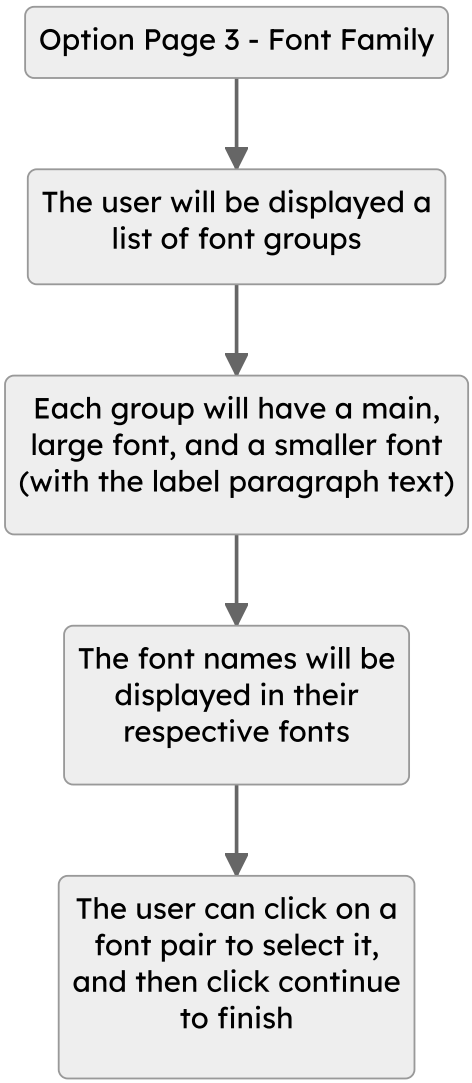

Option Page 3 - Font Family
The user will be displayed a list of font groups
Each group will have a main, large font, and a smaller font (with the label paragraph text)
The font names will be displayed in their respective fonts
The user can click on a font pair to select it, and then click continue to finish

Diagram showing how the subroutines link

Subroutines

Now that I have a rough idea of what the subroutines will do and how they will fit together, I can start planning them in pseudocode. The multi-user subroutines will be written in Python, as it is used for the backend, whereas the subroutines for the website builder will be written in JavaScript and imported into the HTML.

Multi-user system - login system

auth_login_post

The Flask backend will call this subroutine when the user submits the login form. The subroutine can fetch input data from the form using the `flask` `request` import. The password it receives will already have been hashed on the client's side so that it is being sent over the internet encrypted.

```
function auth_login_post() { // run when the user submits the login form
  flask.route("login",method=post)

  // fetch inputs from the form using the ids of the inputs
  username,password,remember = flask.request.form.get()

  // fetch a list of users that match the username from the database
  user = db.query(f"SELECT * from USER where user_id={username}").fetchall()

  // if the list of users is empty or the password doesn't match
  if user.length = 0 || user[0].password == password {
    flash("Please check your login details and try again")
    return flask.redirect(flask.url_for("auth_login"))
  }

  flask.login_user(user,remember=remember) # login the user
  return flask.redirect(flask.url_for("main_home"))
}
```

auth_signup_post

The Flask backend will call this subroutine when the user submits the signup form. It uses similar functionality to the `auth_login_post` function, including the passwords being hashed client-side. It uses the `verifyField` and `isEmailFormat` subroutines to check that fields are valid and the `createUser` subroutine to insert a new user into the database and add them to the server storage. Both subroutines are shown later.

```
function auth_signup_post() { // run when the user submits the signup form
  flask.route("signup",method=post)

  // fetch inputs from the form using the ids of the inputs
  name,email,username,password1,password2 = flask.request.form.get()

  // Use the verifyField function to check the inputs are valid.
  out = verifyField(name,"Name",canHaveSpace=True,canHaveSpecialChar=True)

  if (out) return flask.redirect(flask.url_for("auth_signup"))

  out = verifyField(email,"Email",minLen=0,canHaveSpace=False,
  canHaveSpecialChar=True)

  if (out) return flask.redirect(flask.url_for("auth_signup"))

  // Check to see whether the email is in a valid format
  if (not isEmailFormat(email))
    return flask.redirect(flask.url_for("auth_signup"))

  // Run an SQL query to check whether this email already has an account
  user = db.query(f"SELECT * from USER where email={username}").fetchall()

  if user return flask.redirect(flask.url_for("auth_signup"))

  out = verifyField(username,"Username",canHaveSpecialChar=False)

  if (out) return flask.redirect(flask.url_for("auth_signup"))

  out = verifyField(password1,"Password",minLen=8)

  if (out) return flask.redirect(flask.url_for("auth_signup"))

  // Make sure the passwords match
  if (password1 != password2) return flask.redirect(flask.url_for("auth_signup"))

  // run the createUser function to insert a user into the database
  createUser(username,email,name,password)

  // redirect to the home page
  return flask.redirect(flask.url_for("main_home"))
}
```

verifyField

This subroutine will be called from `auth_signup_post` to ensure that all of the fields the user inputted are valid. It takes four variables, the requirements that the field has to meet, along with the field's content and name for any error messages. It will return an empty string if the field meets all the requirements and an error message if it does not.

```
function verifyField(field,fieldName,mustHaveChar=True,minLen=3,
  canHaveSpace=False,canHaveSpecialChar=True) {
  // field, required, string, the content of the field
  // fieldName, required, string, the name of the field inputted
  // mustHaveChar, optional (default=true), boolean, whether or not field must
  // contain characters
  // minLen, optional (default=3), integer, the minimum length of field
  // canHaveSpace, optional (default=false), boolean, whether or not field can
  // contain whitespace
  // canHaveSpecialChar, optional (default=true), boolean, whether or not field
  // can contain any of a list of special characters

  // the list of special characters that canHaveSpecialChar refers to
  specialChar = "%&{}\\<>*?/$!'\":@+`|="

  // Make sure that field is the correct datatype
  if field.type is not str {
    raise Exception(
    f"Invalid data type for field. Expected string, received {field.type}")
  }

  // If field is empty and mustHaveChar is true
  if (field.length == 0 and mustHaveChar) return f"{fieldName} is not filled out."

  // If field is shorter than minLen
  if (field.length < minLen)
    return f"{fieldName} must be greater than {minLen-1} characters."

  // If field contains spaces and canHaveSpace is false
  if (not canHaveSpace and " " in field)
    return f"{fieldName} cannot contain spaces."

  // If the field contains any of the specialChars and canHaveSpecialChar is false
  if (not canHaveSpecialChar) {
    for (char of specialChar) {
      if (char in field) return f"{fieldName} cannot contain '{char}'"
    }
  }

  return ""
}
```

createUser

This subroutine will be called from `auth_signup_post` when it wants to add a new user to the system. Using the arguments given, it will insert a new user into the database and generate the required folder structure for the user using the subroutine `generateFolderStructure`. It is a procedure, so it will not return anything.

```
function createUser(username,email,name,password) {
  // generate the model for a new user
  newUser = db.User(
    user_id=username,
    name=name,
    email=email,
    password=password,
    archived=False,
  )

  // the base path for where the folders should be created
  prefix="static/data/userData/"

  // using the os.path module, get the absolute paths of the required folders
  folderStructure=[os.path.abspath(f"{prefix}{u}"),
                   os.path.abspath(f"{prefix}{u}/sites/")]

  // create all the required folders
  generateFolderStructure(folderStructure)

  db.session.add(newUser)
}
```

Multi-user system - creating a new site

First page submit button subroutine

On the first page for creating a new site, the `checkFormSubmitButton` subroutine is called to see whether all inputs have been filled out and are valid. If everything is acceptable, it will remove the `disabled` tag from the button element, defined as `formSubmit` in the JavaScript.

```
function checkFormSubmitButton() {
  // if the website name is either marked as success or warning
  // AND one of the form privacy options is checked, remove the attribute
  // else set the button to disabled

  if  (formName.getAttribute("data-form-input-display") == "success" ||
       formName.getAttribute("data-form-input-display") == "warning")  &&
      (formPrivacy1.checked || formPrivacy2.checked) {
        formSubmit.removeAttribute("disabled","")
  } else formSubmit.setAttribute("disabled","")
}
```

Website name formatting subroutines

There are certain requirements for a site name that need to be fulfilled, including limiting it to only certain characters and no spaces (the specific requirements are listed elsewhere in the document). To ensure that the user knows what their final site name looks like, the JavaScript converts the input into a valid name, that will be used when it is created server-side. As such, some of these functions will exist both client-side and server-side.

Whenever there is a change in input to the website name input, defined by `forminput_websiteName`, it will call the `verifyNameField` function to validate the input. The function will return a class name based on the result of the validation process, that will change the styling of the input to match.

`verifyNameField` uses functions such as `replaceRepeatedDashes`, which uses recursion to replace any substrings of dashes in a given string, in this case the input given by the user.

Variables such as `messageSpan` and `messageContainer` are defined earlier using `document.querySelector` calls to fetch HTML elements from the DOM (document object manager).

```
forminput_websiteName = document.getElementById("input_websitename");

forminput_websiteName.addEventListener("keyup",(event) => {
  // set the color of the input
  forminput_websiteName.classList.append(verifyNameField())
  checkFormSubmitButton();
})
```

```
function hideFormMessage() {
  // clears the current contents of the warning dialog
  // used in the verifyNameField function.
  messageContainer.classList.add("visibly-hidden")
  messageSpan.innerHTML=""
}
```

```
function hasRepeatedDashes(val) {
  // checks whether a string has repeated dashes in it
  for (i in range(val.length))
    if (val[i] == "-" && val[i+1] == "-")
      return true
  return false
}
```

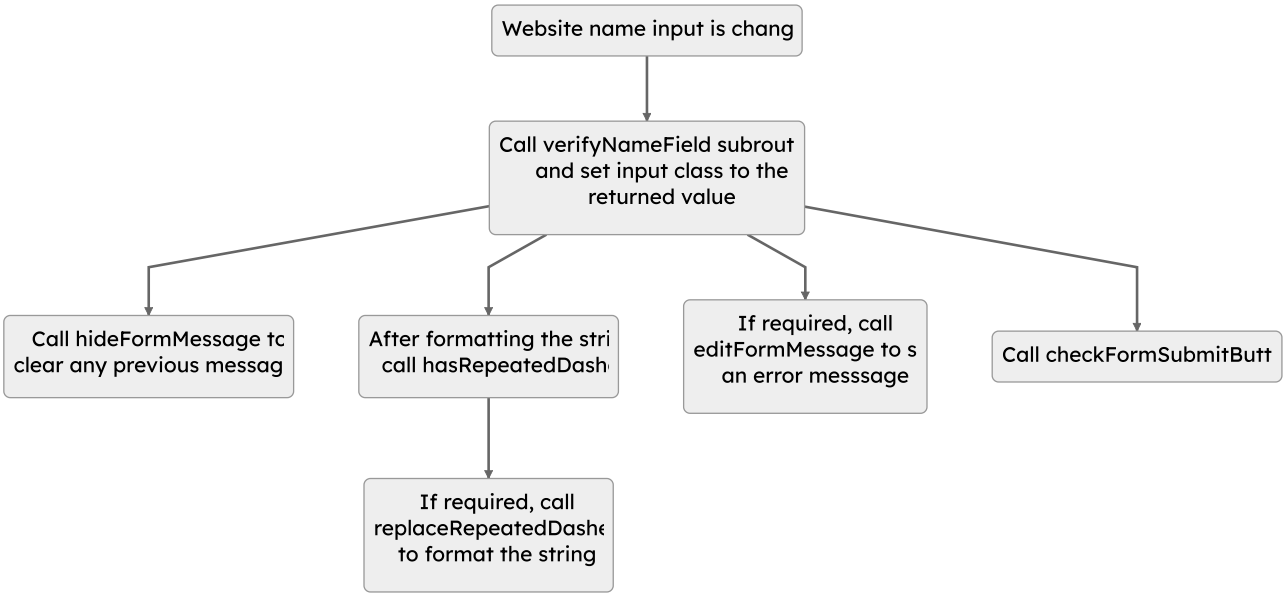
```
function replaceRepeatedDashes(val) {
  // replaces all sets of repeated dashes with a single dash
  // by implementing recursion
  for (i in range(val.length)) {
    if (val[i] == "-" && val[i+1] == "-") {
      val.remove(i+1)
      val=replaceRepeatedDashes(val)
    }
  }
  return val
}
```

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

  for (i in range(val.length)) {
    letter=val[i];
    if !(allowedChars.includes(letter)) val=val.replaceAt(i,"-")
  }

  if (hasRepeatedDashes(val)) { val=replaceRepeatedDashes(val) }

  messageSpan.innerHTML=val
}
```

```
function verifyNameField() {
  // Adds content to the warning dialog (if required), and returns a class name
  // that will color the name input box

  val = forminput_websiteName.value;

  hideFormMessage()

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

  // must include at least one of the given characters

  check=true
  for (i in range(val.length)) {
      letter = val[i]
      if ("qwertyuiopasdfghjklzxcvbnm1234567890".includes(letter)) { check=false }
  }

  if check return "danger"

  sitenames = request.db.query(
    f"SELECT sitename from SITE where userid={session.user.id}")
```

```
  if (val in sitenames) {
    messageSpan.innerHTML= "A site with this name already exists!"
    return "danger"
  }

  for (letter of val) {
      if (letter not in "qwertyuiopasdfghjklzxcvbnm-._1234567890") {
        editFormMessage(val)
        return "warning"
      }
  }

  if hasRepeatedDashes(val) {
    editFormMessage(replaceRepeatedDashes(val))
    return "warning"
  }

  hideFormMessage()
  return "success"
}
```

Diagram showing how these subroutines link



Website colour palette selection subroutines

There will be data structures (such as `userSelectedColors` and `defaultColors`) storing the different colours that the user selects, and their corresponding generated colours.

The `updateStored` subroutine will take the content of the `userSelectedColors` data and append it to a HTML element (`colorOutputSpan` in the JS code) inside a `<form>`, so that it can be sent to the server. This is the only way I've been able to send JavaScript-generated information to the server, after doing some testing. The `updateStored` function is called every time the user changes an input, or when a new set of colours is generated.

```
function updateStored() {
  keys=userSelectedColors.keys()
  for (key in keys) out+=key+":"+colors[key]+","
  colorOutputSpan.innerText = out
}
```

The `updateColorVariables` subroutine is called each time any of the colour pickers are changed, to generate lighter and darker versions of the given colour.

```
function updateColorVariables() {
  for (color in colorPickers) {

    newColor = rgbToHsl(color.r,color.g,color.g)
    newColor = darken(newColor,CHANGEPERCENT)
    newColor = hslToRgb(newColor.h,newColor.s,newColor.l)
    newColor = rgbToHex(newColor.r/255,newColor.b/255,newColor.g/255)
    userSelectedColors[f"{color}-dark"]=newColor

    newColor = rgbToHsl(color.r,color.g,color.g)
    newColor = lighten(newColor,CHANGEPERCENT)
    newColor = hslToRgb(newColor.h,newColor.s,newColor.l)
    newColor = rgbToHex(newColor.r,newColor.b,newColor.g)
    userSelectedColors[f"{color}-light"]=newColor

  }

  updateStored()
}
```

Diagram showing how these subroutines link

Utility subroutines

These subroutines are called in different parts of the Python files to perform specific actions. This means that it removes duplicate code for procedures that may need to be used many times throughout

generateFolderStructure

This subroutine is called whenever the code needs to generate a list of folders. It makes use of the in-built `os` library in Python. It is called when a new user is created or when a user creates a new site.

```
def generateFolderStructure(folders):
  for folder in folders: # iterate through the list of folders
    if os.path.isdir(folder): # if the folder already exists
      continue
    try:
      os.makedirs(folder)
    except OSError as e: # error catching if required
      raise OSError(e)
```

generateFileStructure

This subroutine is called whenever the code needs to generate a list of files. It makes use of the in-built `os` library in Python. It is called when a user creates a new site.

```
def generateFileStructure(files):
  for file in files: # iterate through the list of files
    if os.path.exists(file): # if the file already exists
      continue
    try:
      with open(file,"w") as f: f.close() # write a new, empty file
    except OSError as e: # error catching if required
      raise OSError(e)
```

Explanation and justification of this process

The initial concept seems large and complicated, but the way it is broken down above into separate parts will make the development easier and faster, and will aid the testing and maintaining of the code due to it's modularity.

The program will be split into three main sections: the multi-user system (including the login system and the creation of sites), the site edit system (including the drag-and-drop editor and the styling system), and the user interface.

The multi-user part of the program will integrate with the SQL database and the server-side file storage. It is mostly made of sequential algorithms that are either called when the user performs a certain interaction, or when another algorithm calls them.

The diagrams above can be used to explain this. The `auth_signup_post` subroutine is called when the user completes the signup form, which then calls the `verifyField`, `isEmailFormat`, and `createUser` subroutines.

Breaking it down like this ensures that the program is modularised in such a way that all of the subroutines can interact with each other. If there is a bug that needs fixing, or a change that needs implementing, it is easy to find the code that needs editing and change it without breaking the rest of the program.

The code will be very modular, which will help with development and any changes that will be made later. This will be achieved by following this design and separating the multi-user system, editing system, and user interface. If another developer were to take over the programming, this design would make it easier to understand and make amendments. The two different programming languages, Python and JavaScript, will communicate via Flask's `session` and `flash` features to make sure that the two languages can interact with each other. The functions will be contained in a class, that will be initialised when ran, to make use of the `self` variable communication so that all of the subroutines can use the same variables. It will also use variables and `return` statements for some subroutines where necessary.

Inputs and Outputs

Input	Process	Output
Login submit button	`auth_login_post` to verify user input.	Log in the user to the session, or provide a suitable error message.
Signup submit button	`auth_login_signup` to verify user input, insert the new user into the database and generating the new folder structure for the user.	Log in the user to the session, or provide a suitable error message.

Input	Process	Output
Create site inputs (when the user goes through the new site creation pages)	Store the inputs in the session, and generate a new site in the database & file structure from the inputs given.	Redirect the user to the site page.
Home button	Redirect the user to the homepage.	Redirect the user to the homepage.
Menu hamburger	Darken the main content and display the menu items over them.	Show the user the menu items.
Hamburger open and anything else selected	Remove the darkening of the main content and hide the menu items.	Hide the menu items.
Site edit option (Add Section, Website Pages, etc)	A modal will open over the main content with the main content being darkened, with the content relating to the selected option.	A modal with relevant options is displayed.
Display size option.	The site container will change width to match the option selected. The elements in the container will have the necessary data tags appended.	The editing window will change size to match the selected option.
Element selected	Fetch element data tags to perform appropriate tasks. Listen for events such as dragging the element, or the resize box, if necessary.	Display appropriate style settings in right hand dock. Display resize box. Display text editor if necessary.
Section selected	Fetch section data tags to perform appropriate tasks. Listen for events such as dragging the section up and down, or dragging on the bottom to resize, if necessary.	Display appropriate style settings in right hand dock. Display resize bar on bottom when hovered.
Style option hovered	Apply the hovered style to the selected element.	Render what the element will look like with the hovered style.
Style option selected	Apply the selected style to the selected element.	Add the selected style option to the element.

Key variables

These are the main variables that the Python program will use:

Name	Data Type	How it is used
`host`	string, in the format `x.x.x.x`	Defines the host the server uses
`port`	integer	Defines the port the server uses
`databaseObject`	`flask_sqlalchemy.SQLAlchemy` object	Access the SQL database
`loginManager`	`flask_login.LoginManager` object	Manage the user login system
`app`	`flask.Flask` object	Manage the site routing

These are the main variables that the JavaSript code will use:

Name	Data Type	How it is used
`requiredChars`	string	A set of characters of which at least one must be in a site name for it to be valid.
`allowedChars`	string	A set of characters that are allowed in the site name when creating a new site.
`defaultColors`	dictionary of strings	The default colour scheme when generating a new site.
`userSelectedColors`	dictionary of strings	The selected colour options when generating a new site.
`colorPickers`	dictionary of lists of elements	The colour preview elements when generating a new site - the first one is the element that changes colour, the second one is the text element that displays the current hex colour.
`CHANGEPERCENT`	integer	The amount to lighten or darken colors when generating.

Name	Data Type	How it is used
`textOptions`	list of elements	A list of the text options when generating a new site - they get given event listeners for when they are selected.
`sectionSelectorNavSelected`	string	A written number referring to the selected section category in the `Add Section` modal.
`sectionSelectorNavSelectedInt`	integer	An integer version of the previous variable. A written number is used to insert it into classes, as integers are not allowed.
`sectionSelectorNavItem`	string	Template for a section link element for the section navigation bar in the `Add Section` modal.
`selectedElement`	element	The currently selected element.
`fonts`	list of dicts	A list of all of the fonts that the style settings use.
`fontDropdownItem`	string	Template for a font dropdown element for the font family dropdown in the style modal.

Validation

To make sure that the program is robust and will not throw critical errors, validation will be used throughout the code to ensure that the data entered by users is accurate, complete, and conforms to specific rules and constraints. Due to the usage of `<input>`s in the HTML code as the vast majority of the input methods, HTML can make sure that the correct data type is being inputted. However, some text that the user input needs to be validated to make sure that it meets certain requirements. Instances of this include:

Login and signup forms

For the login and signup forms, the text inputted need to have specific parameters. For example, the username must have a minimum length of 3 characters, must not contain special characters, and must be unique in the database. To verify most of this, the `verifyFunction` subroutine is used. This subroutine is outlined earlier in the document. For specific things like checking that the username is unique, or that the password matches the given user, SQL queries are used to validate the inputs. The data here is verified on the client-side before being sent, and is then checked server-side to double check that the data is valid.

Website Name when creating a new site

The website name must meet these specific requirements:

- At least four characters
- At least one alphanumeric character
- Illegal characters can be inputted, but will be changed

In the JavaScript, it will take the content of the input and replace any illegal characters into dashes, then display this name to the user. It will also make sure that the form can be submitted until the input matches the given criteria. This process is outlined earlier in the document. This is an example of client-side validation, where the JavaScript checks the data locally before sending it off to the server.

Importing an exported site

When the user attempts to import a zip file containing a website, the zip file will be verified to conform with a specific format that exporting will use. If any malicious files are found, or any extra files that are not supposed to be there, the website will throw an error client-side before sending it to the server. It will also make sure that the HTML files contained are in the correct format.

Uploading data to the CMS

There will be a whitelist for the allowed files that can be uploaded to the CMS, and these will be checked and validated before they are sent to the server.

Data Sanitization

Throughout the code, the user input will be checked and cleaned to remove the risk of potentially dangerous or malicious data before being stored in the database. For example, to negate the possibility of an SQL injection attack, the library used to manage the database removes any usage of SQL queries in the code, meaning the data inputted cannot be used to execute a query. Other attack methods that will be looked into include XSS (cross-site scripting), DDoS (distributed denial of service), and MitM (man in the middle) attacks.

Testing method

When developing the project, a lot of the alpha testing done will be white box testing, done by the developers. Unit testing will be used to ensure that all of the subroutines function as expected and intended. By testing each module of the program individually, this means that when they are all combined together, the program will function correctly. Integration testing will be performed to make sure the program functions as a whole. This will include checking how different modules interact with each other, and how the frontend interacts with the backend of the website.

Different areas of testing when programming will include the input data of the program (for example, the user input on the login form), how the program handles said data, and what the result will be. To ensure that it has suitable error catching throughout, each module should go through destructive testing. For user input, this means using a variety of incorrect entries to see how it handles them. For the editor, this means attempting to perform styling that is invalid, dragging elements outside their boundary region, or making them too large. This will also include security testing; making sure that SQL injection or XSS attacks do not work.

Usability testing will be done both white-box and black-box - the UI design will be tested during and after development and will be tested with some of the stakeholders to make sure that it is easy to understand and navigate, and that it functions as intended on a variety of devices, resolutions, and browsers. Feedback, criticisms, and suggestions from the stakeholders will be taken after these sessions to ensure that the final product is easy to use and meets their requirements.

Any testing will be recorded during the development process, such as different platforms used in the UI design, or invalid inputs entered when checking user input.

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 reogranisations of the page, would be displayed in later sections, but the actual process of orignally 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.

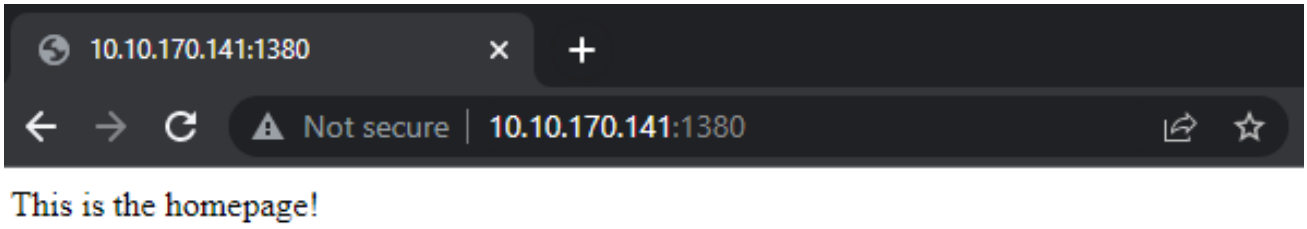
__init__.py

```
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:



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

```python
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
```

```python
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")
```

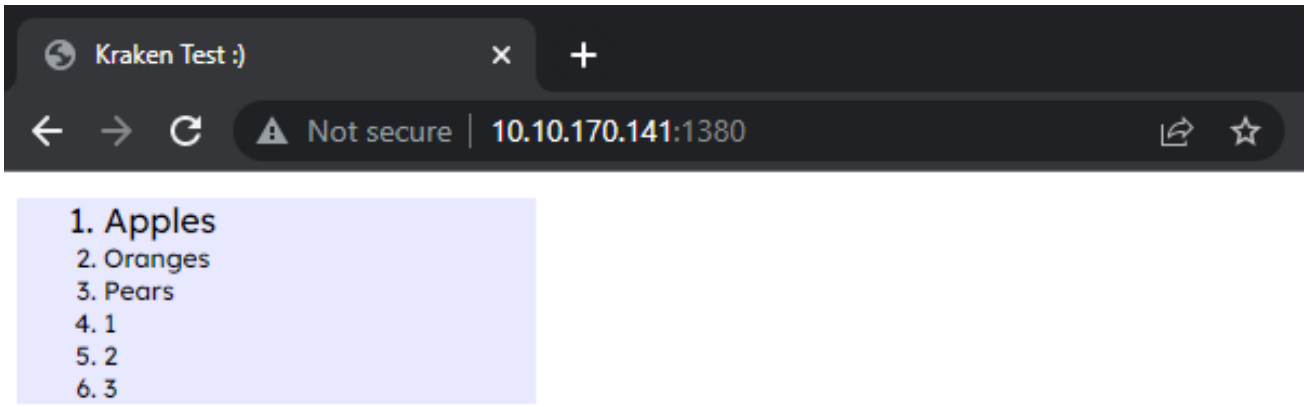
/templates/test.html

```
<!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")
```

/templates/base.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') }}">
              <img class="globalnav-logo-image" alt="Kraken" src="{{url_for(
                'static', filename='img/icon/512-512/kraken-icon-png-primary-
                512-512.png')}}" preserveAspectRatio>
              <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>
```

```
    <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 %}
```

/templates/signup.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 - 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>
```

```
            {% 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%
}
```

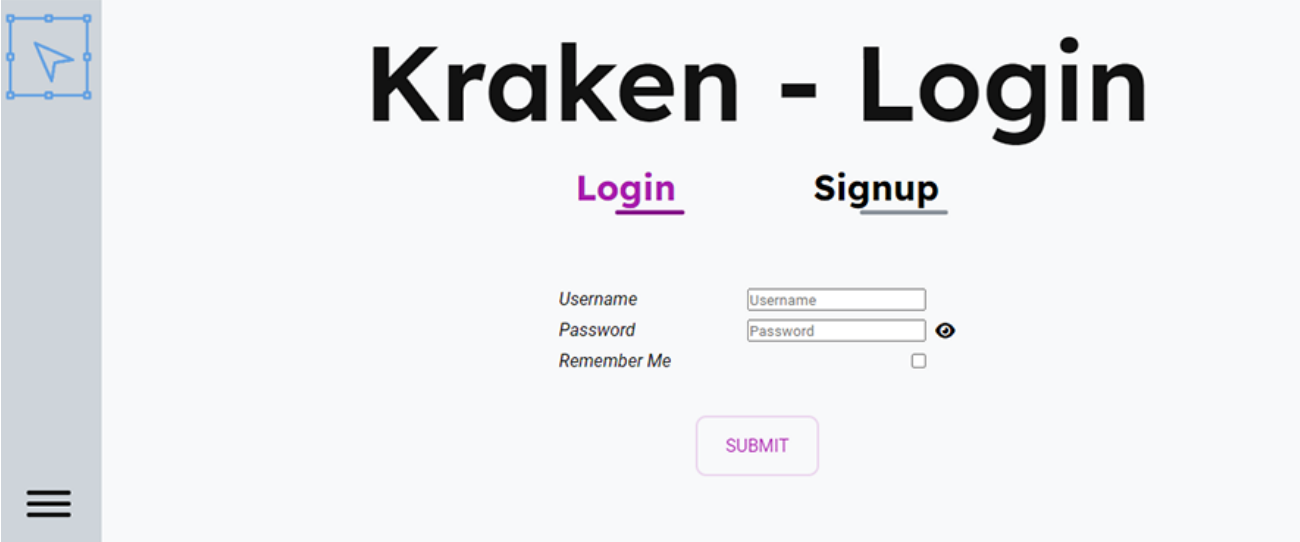
/static/css/auth.css

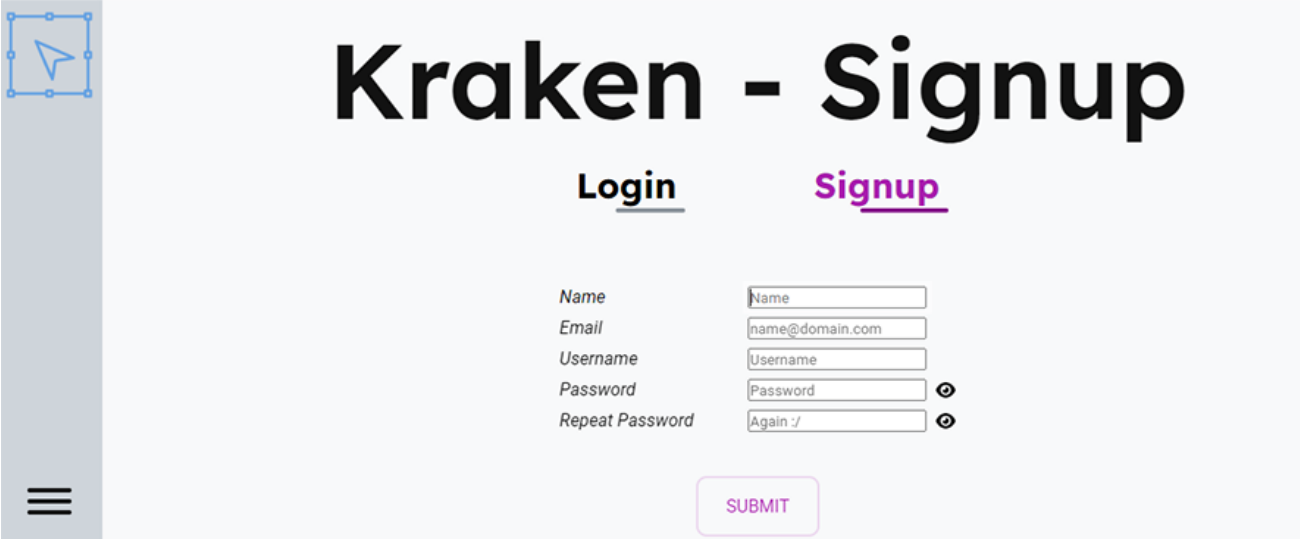
```css
.application-container {
    width:100vw;
    height:100vh;
    display:flex;
    flex-direction:row;
}

.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);
}
```

```css
.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;
}
```

```css
.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:





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.

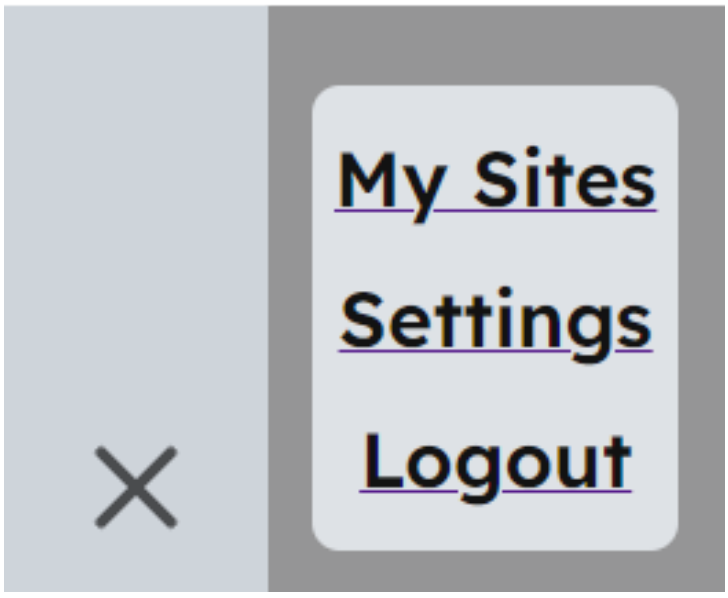
The next step was to set up the frontend programming for the login and signup pages, and the navigation bar. Starting with the navigation bar, I needed to set event listeners for the hamburger and background darkener `div`. These listeners would add or remove the `is-active` class to the hamburger, options 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
  }
```

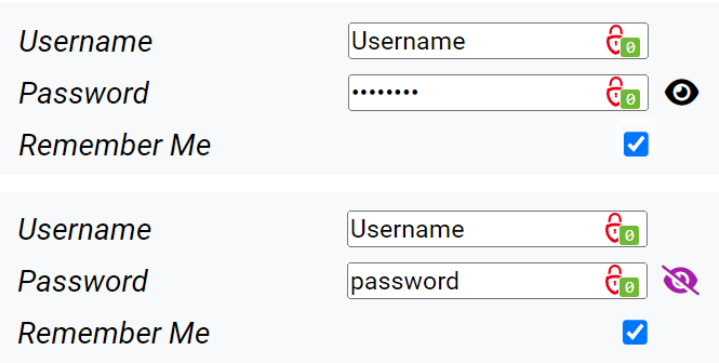
```
  warningSpan.innerText = ""
  document.querySelector(".field-submit").disabled = false
}

// 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:



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')}}">
```

```
<img class="globalnav-logo-image" alt="Kraken" src="{{url_for('static',
filename='img/icon/512-512/kraken-icon-png-'+navbarLogoColor+'-512-512.png')}}"
preserveAspectRatio>
```

```
{% 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 %}
```

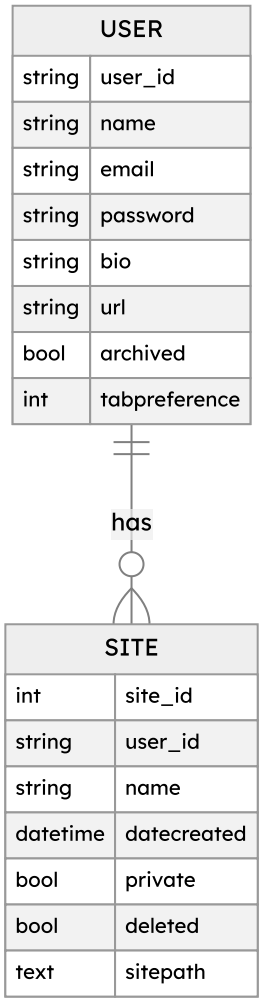
changes to /templates/login.html and /templates/signup.html

```
{% 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 dabase, referring to the entity relationship diagram that I had outlined when planning. The two entities are `User` and `Site`, where `user_id` is a foregin 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)`.

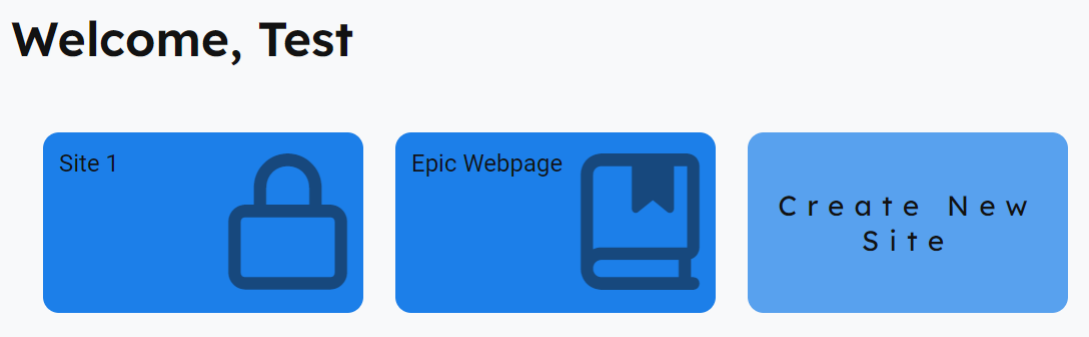
```
{% extends "base.html" %}

{% 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

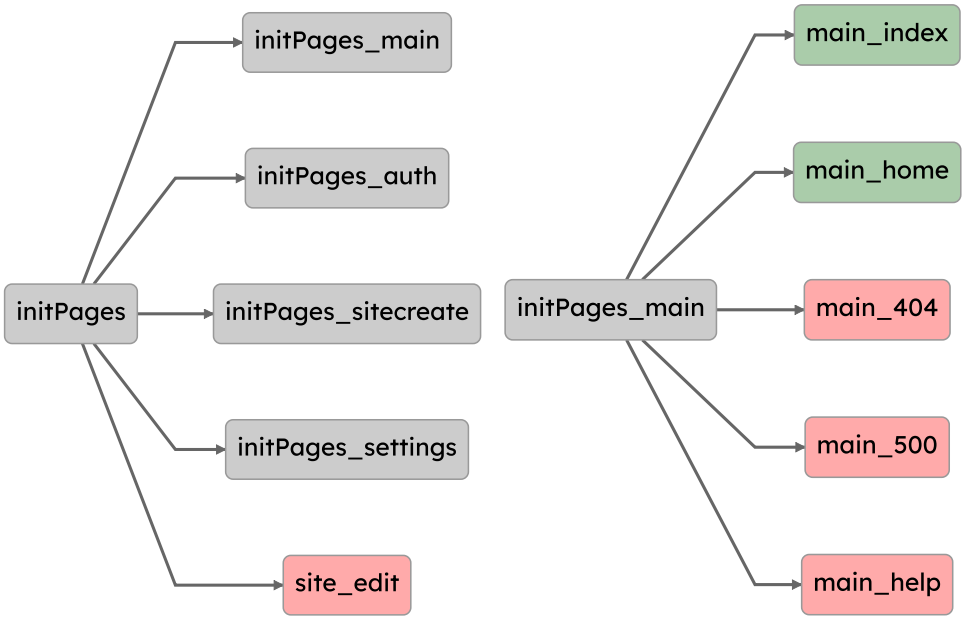
```
{% extends "base.html" %}

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

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

{% block content %}
```

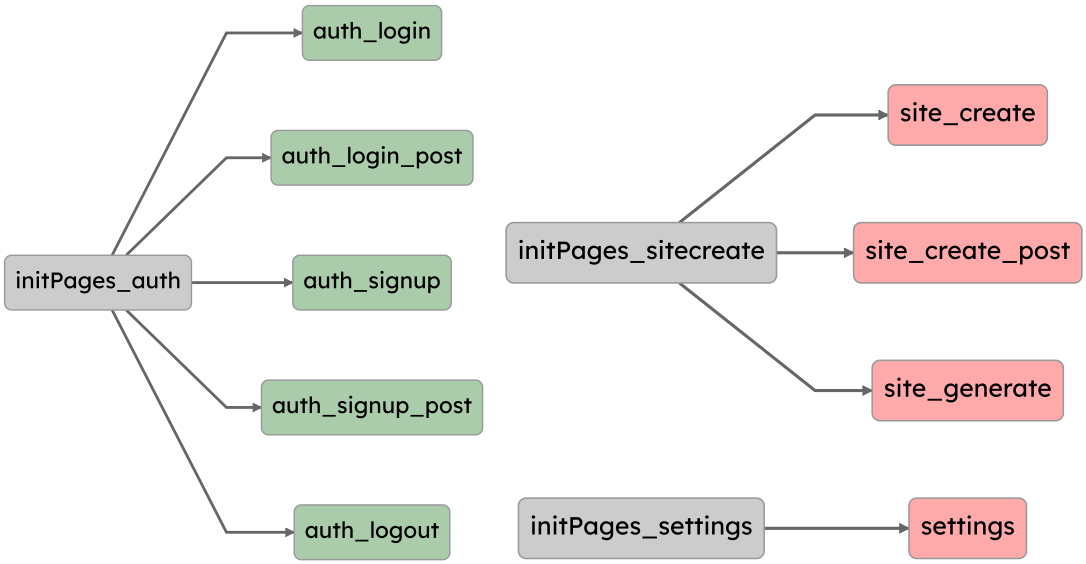
```
<link href="{{url_for('static', filename='css/home.css')}}" rel="stylesheet"
type="text/css" />

<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 interperet. 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.

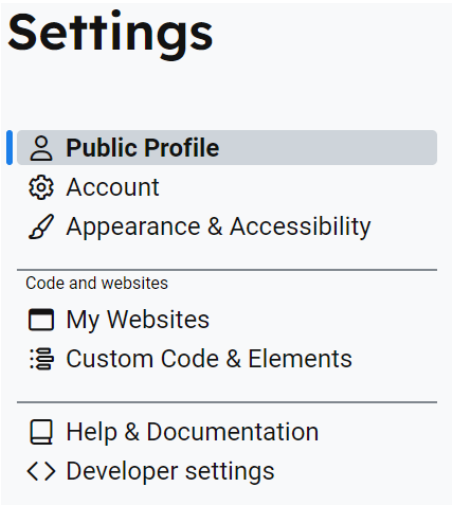




At this point in development, I was able to quickly build the frontend versions of the settings pages for the account system (they are listed in the desirable features section of the success criteria). I didn't add in database functionality to them yet as I knew it would take significantly longer.

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 hilighted 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.



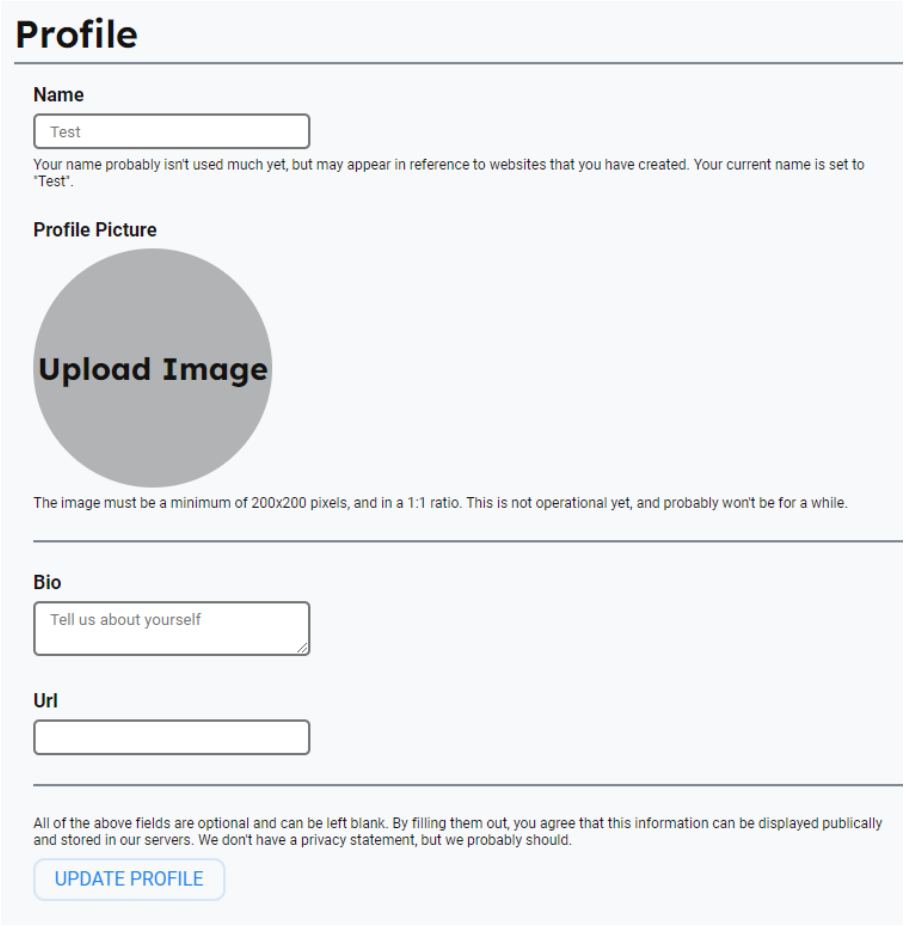
/templates/settings-base.html

```
{% 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>
```

```
    <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.



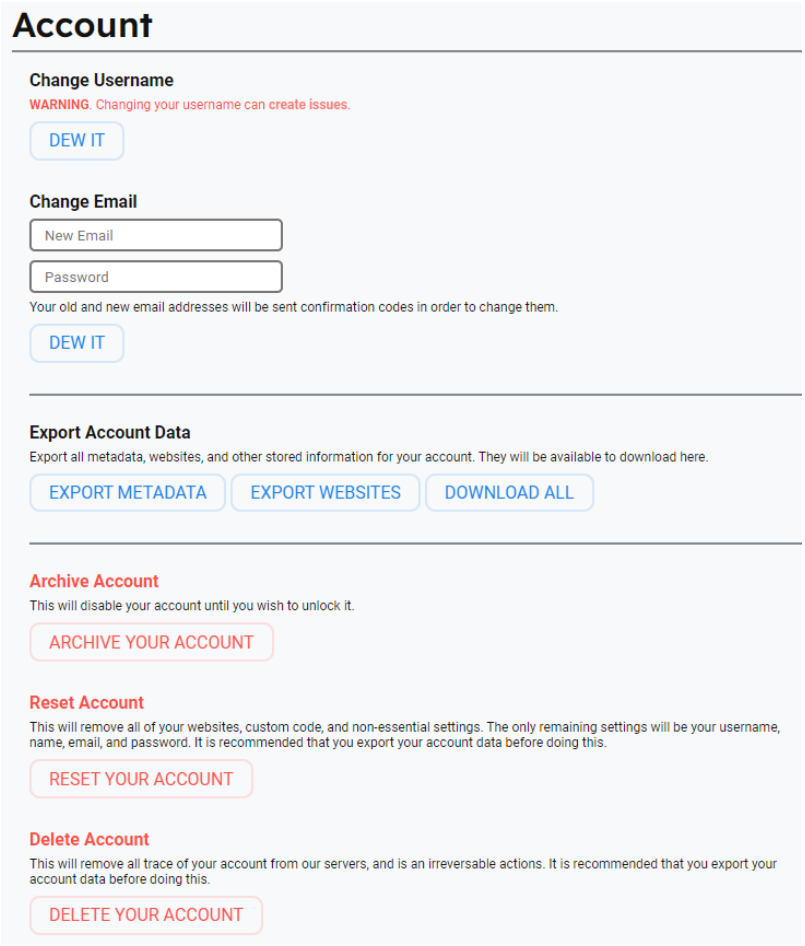
/templates/settings-profile.html

```
{% 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.



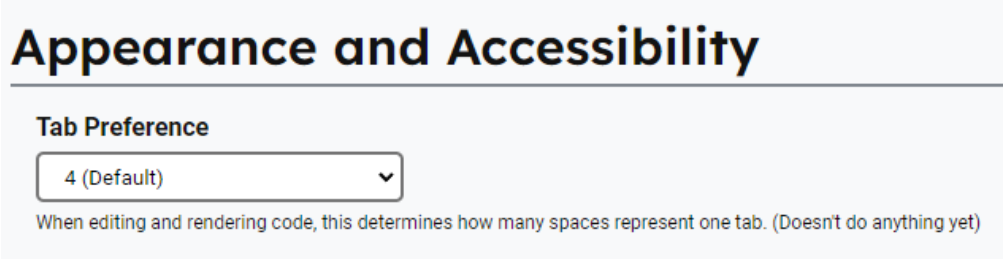
/templates/settings-admin.html

```
{% 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
      irreversable 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.

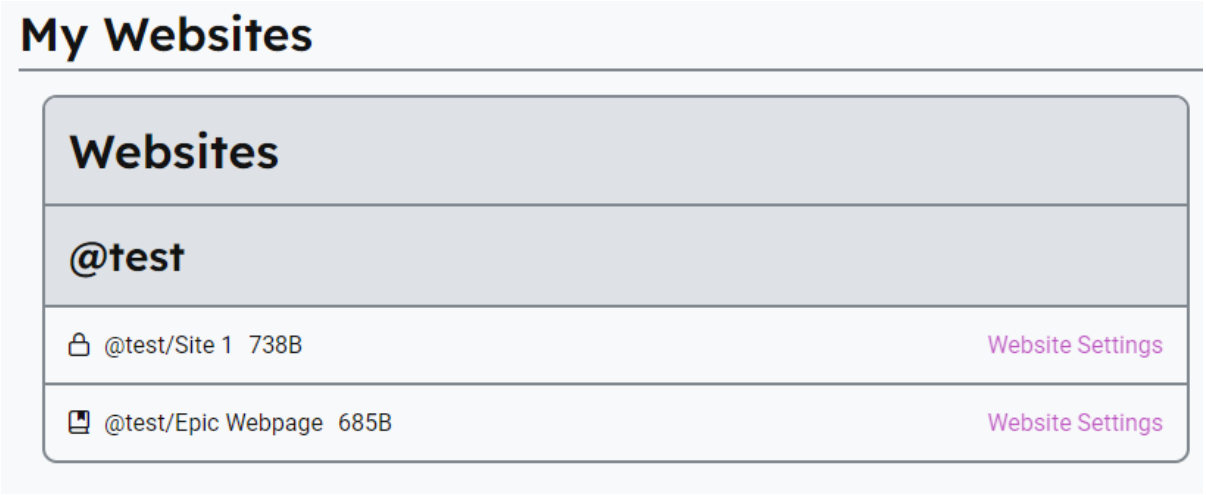


/templates/settings-looks.html

```
{% 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>
```

```
      <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.



/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 interpereted 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

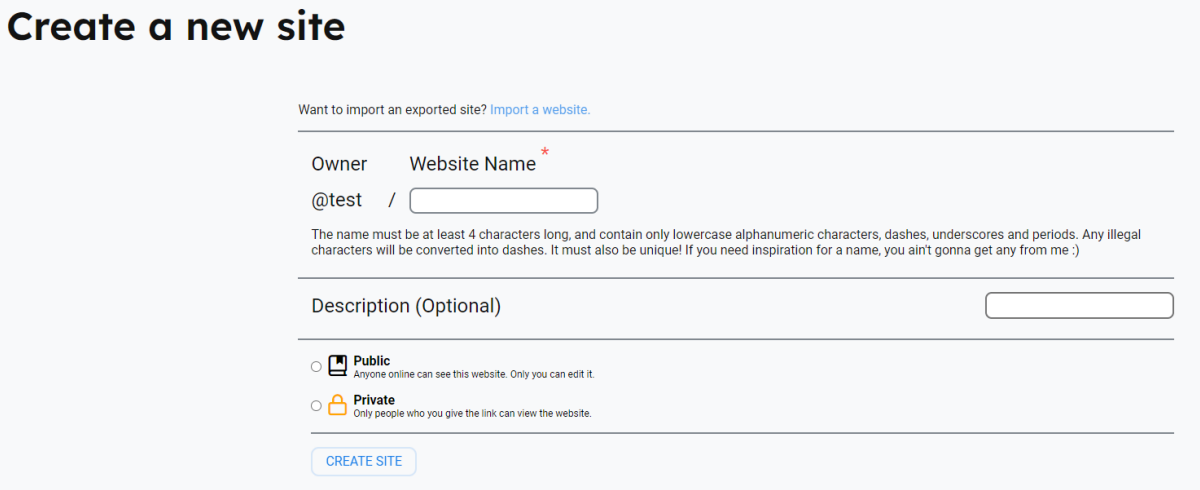
In this stage, I created the system for when a user creates a new site. This includes various subroutines outlined in the design section of the report, such as the website name formatting and storage of colour palettes. Some functionality has not yet been included yet, such as the ability to import a 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:



/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-l 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>
```

```
      <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 wether 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.

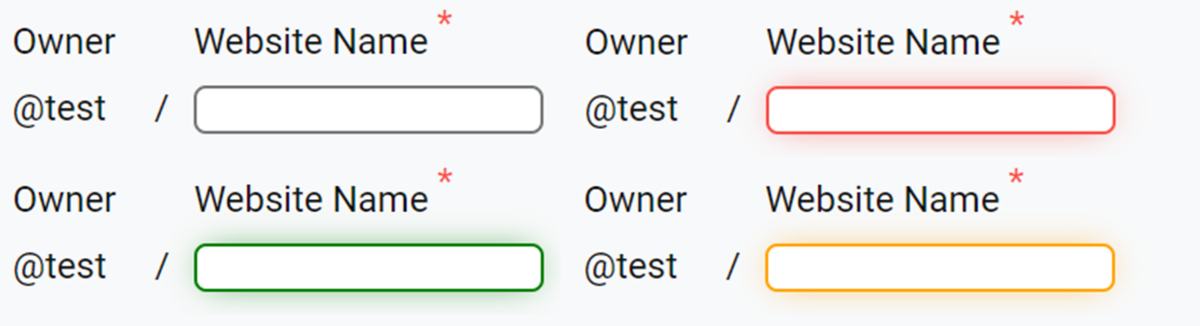
```
function verifyNameField() {
  var nameInput = document.getElementById("new_site_name");
  var val = nameInput.value;

  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`.



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))
```

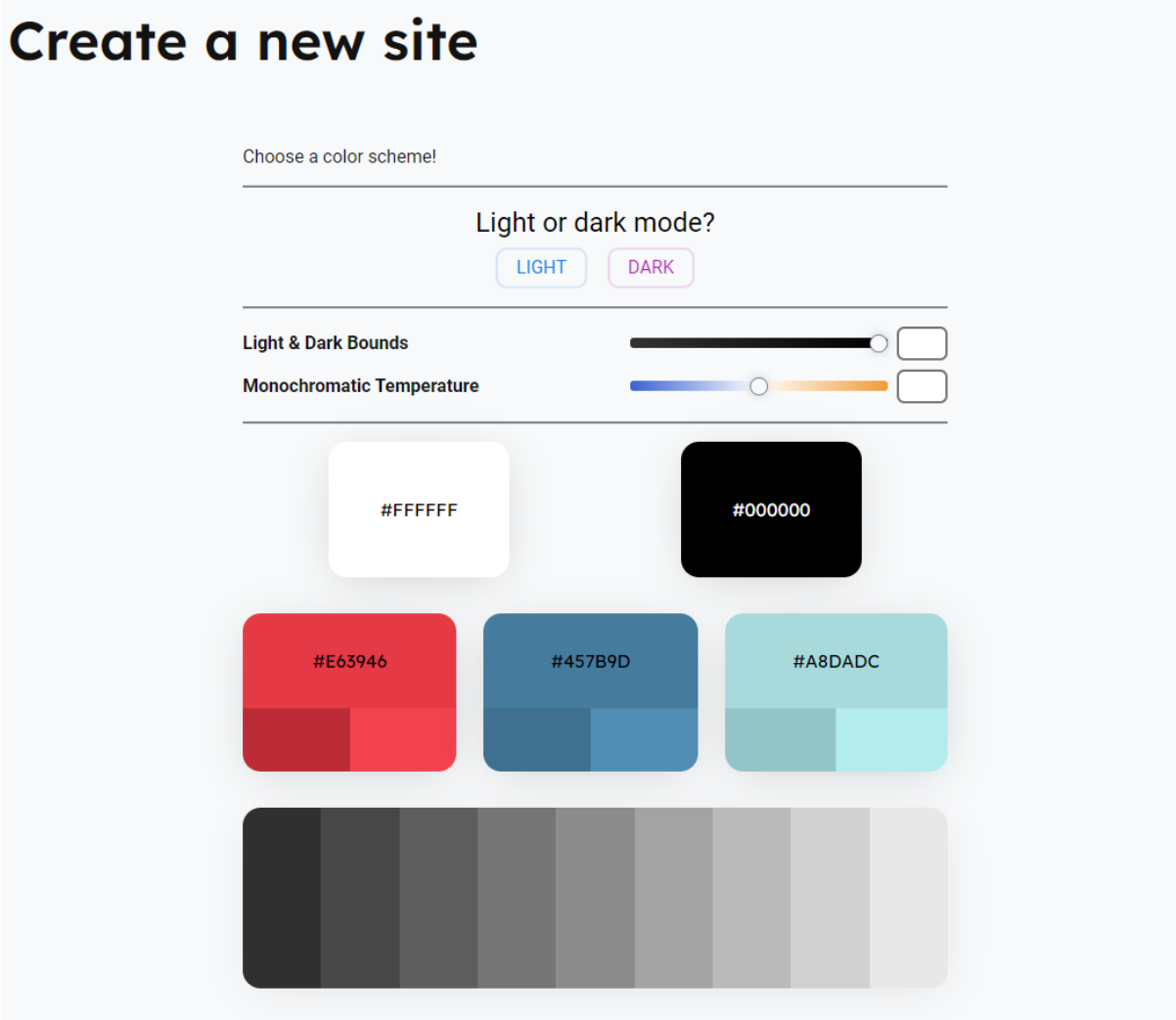
```
        return listToStr(var)

    # 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

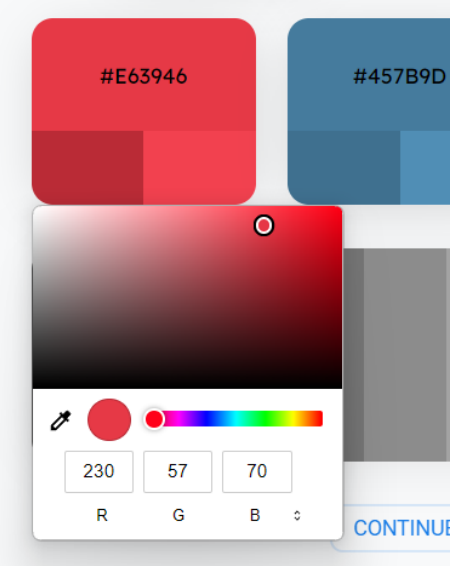
```
@self.app.route("/home/new/1")
@login_required
def site_create_options_1() -> str | Response:
  # stop people from starting halfway through the form i.e. if they didnt come
  # from the previous site_create page, send them to the start
  if not request.referrer == url_for("site_create",_external=True):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



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>

  <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 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 appeneded this element to the end of the form and then wrote the `updateStored` function, which sets the content of the input to the contnet 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:

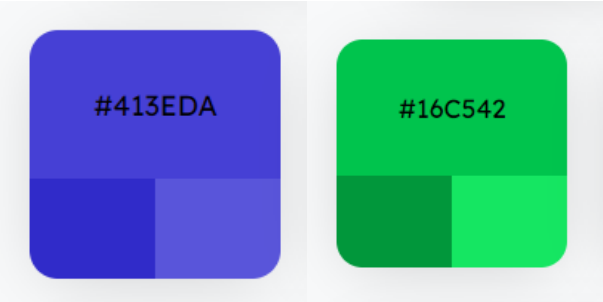
```
function updateColorDisplays() {
  colorDisplay["primary"][0].style.backgroundColor=colors["primary"]
  colorDisplay["primary"][1].innerHTML=colors["primary"]
  colorDisplay["primary-dark"][0].style.backgroundColor=colors["primary-dark"]
  colorDisplay["primary-light"][0].style.backgroundColor=colors["primary-light"]

  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 callede 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.g)

    // 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()

}
```

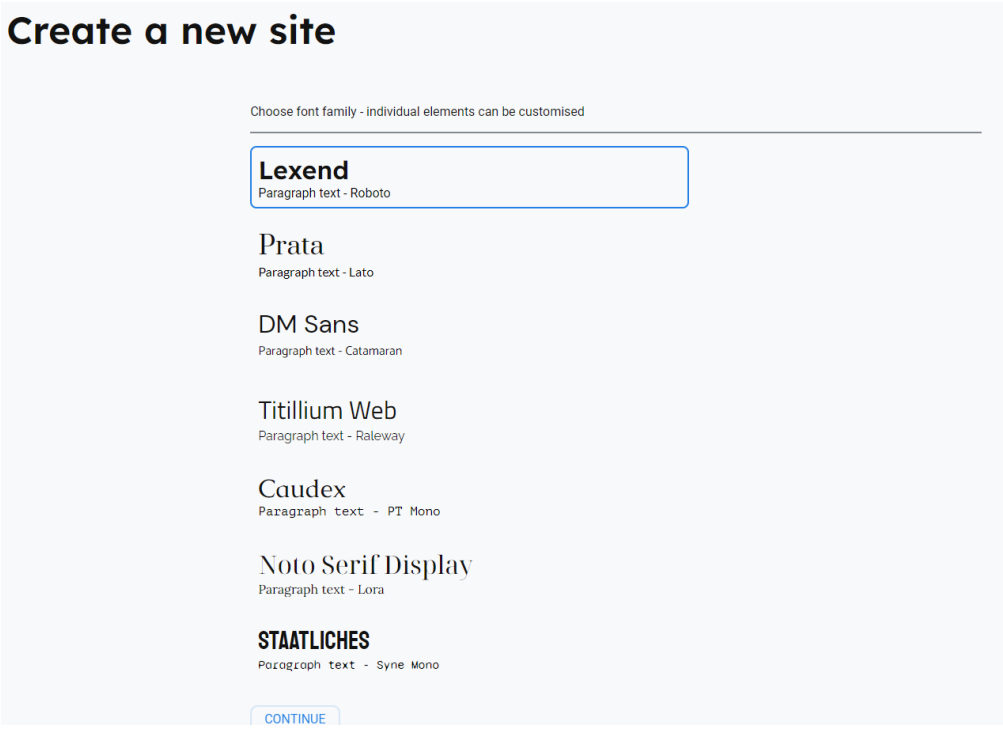
In the backend, the form retrieves the inputted colour variables via the form element `new_site_color_options_dict`. It then interperets the given input into a dictionary, and stores it in the `flask.session` object, meaning it can be accessed at a later date.

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.



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 varaibles
  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) -> 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
```