

Table of Contents

• Analysis	3
○ Problem identification	3
○ Stakeholders	4
○ Why it is suited to a computational approach	5
○ Initial talks with stakeholders	8
■ Analysis from stakeholders	12
○ Research	13
■ Existing solution - Squarespace	13
■ Existing solution - Zyro	17
○ Key features of the solution	20
○ Limitations of the solution	20
○ Hardware and Software Requirements	21
■ Hardware Requirements	21
■ Software Requirements	21
○ Success Criteria	22
■ Essential Features	22
■ Desirable Features	24
• Design	25
○ URL Navigation	25
○ User Interface Design	26
○ Usability	32
■ Accessibility	32
■ ARIA	33
○ Stakeholder input	33
○ Website structure and backend	34
○ Data storage	36
○ Algorithms	39
■ Drag-and-drop editor algorithms	39
■ Multi-user system algorithms	43
■ Diagram showing how the subroutines link	48
○ Subroutines	49
■ Multi-user system - login system	49
■ Multi-user system - creating a new site	52

■ Utility subroutines	57
○ Explanation and justification of this process	58
○ Inputs and Outputs	59
○ Key variables	60
○ Validation	62
○ Testing method	63
• Development	65
○ Stage 1 - Setting up the website	65
○ Stage 2 - Creating and implementing the database	98
○ Stage 3 - Homepage and Settings	105
○ Stage 4 - Creating a New Site	123
○ Stage 5 - Code Documentation	151
○ Stage 6 - Drag-and-Drop Editor	158
■ HTML Syntax	158
■ Importing Sections	164
■ Adding Sections to the Editor	170
■ Resizing Text Elements	173
■ Dragging and Dropping Elements	183
■ Grid Preview System	187
○ Stage 7 - Loading from and Saving to the Server	191
■ Loading from the Server	191
■ Saving to the Server	193
• Evaluation	199
○ Success Criteria	199
○ Usability	206
○ Performance & Maintenance	206
○ Security	206
○ User Satisfaction	206
○ Accessibility	206
○ Overall Comments	206
• Appendix A - Code	207
• Appendix B - Testing	329

Analysis

Problem identification

In today's ever-expanding digital world, the demand for websites has skyrocketed as more and more people rely on the internet. From promoting business portfolios and creating online stores to hosting games, websites cater to several different purposes. No matter who you are or what you do, having a website is now considered a necessity, especially for businesses. However, creating a professional and functional website can be a daunting task for many individuals and small businesses, especially those without technical expertise or resources.

Existing website builders can be challenging to navigate, require extensive coding knowledge, or lack the necessary design flexibility needed to develop a distinctive and successful online presence. Furthermore, it can be challenging for users to locate a platform that suits their particular requirements because many website builders are targeted at particular industries or types of websites. This makes it challenging for users to find a platform that meets their specific needs, and the task of programming a website from scratch can seem daunting.

The main aim of this project is to develop a platform where anyone can build their own website with a simple user interface, without having to deal with the complexities of HTML, CSS, and JavaScript. This will enable them to focus on creating a website that reflects their brand and meets their business goals. Clients will be able to choose from various style and theme options, or create their own. They can upload images, videos, and other media, and interact with a drag-and-drop interface to easily organise and customise their page's styles and elements. 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 pages together, and preview their website on different display sizes. On top of this, they would be able to add pre-made elements and edit the parameters of the elements in the site, however these features would be more geared towards those with some technical expertise.

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 consist of a wide range of individuals/organisations looking to create a website with minimal technical or time requirements.

For example, categories of potential clients could include:

- Individuals: Personal websites can include blogs, portfolios, or resumes.
- Small businesses: Managers or owners of small businesses can create service provider websites and online stores, or simple websites that show off what their business is

capable of.

- Entrepreneurs: Entrepreneurs can often be assisted by a good looking website for their startup or business venture that successfully manages to capture the spirit of their ideas.
- Non-profit organisations: Non-profit organisations that want to create a website that raise awareness of their cause and encourage people to support them.
- Freelancers: Freelancers and independent contractors may wish to create a websites that 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
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 due to it automating 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, the implementation of a drag-and-drop visual interface can allow users to arrange a range of elements on a webpage (including text, images, and videos) without requiring any coding skills. The visual editor can also allow users to customise the design and layout of their website, including the ability to select from various themes and templates, along with applying pre-defined styles and formatting to the website.

Moreover, the use of computational algorithms can enhance website performance across a range of different devices and screen sizes, ensuring that the user's content is not compromised by their client's device. By implementing CSS media queries, the website's layout can be programmatically changed based on screen dimensions and what it is being viewed on. Additionally, JavaScript can modify the layout in accordance to the user's device and screen size.

A computational approach can also be used to ensure speed, security, and reliability for the website. Making use of programming techniques such as minifying (removing redundant elements of code files), compression (reducing file size and quality in exchange for faster loading times), and caching, allows the website to load faster. Furthermore, authentication algorithms can be used both client and server-side to ensure that the website is secure.

The solution will be implemented via a server hosting the website, which will require a computer to use. There is no way around requiring 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, therefore, acts as a bridge between the user and the code. The project would manage any technical details allowing the user to focus on creating a website that suits their requirements, without needing to understand the technical details of coding.

Computational methods that the solution lends itself to:

Problem recognition

The solution is suitable for problem recognition as it addresses the problem that many small businesses or individuals may face: websites, especially professional-looking and functional ones, can be difficult to create without technical expertise or resources. The user-friendly, drag-and-drop interface that this solution proposes will allow users to create and edit their website quickly and easily, without harsh technical requirements. This addresses the problem of users not having the required resources or technical skills to create a website for themselves or their business.

Furthermore, website builders can take advantage of computational algorithms to enable optimisation across a wide range of devices and resolutions, ensuring a response and easy experience for users on any device. This addresses the problem of the rising popularity of mobile devices, and the increasing demand this places on responsive design.

Problem decomposition

The problem can be broken down, or "decomposed," into smaller, more manageable tasks that must be programmed for the program to operate effectively:

- Creating a multi-user system that implements a database for data storage.

This system would allow for many users accessing the server and editing their sites simultaneously. An SQL database would store the user data server-side. The sites would also be stored on the server, but in a separate location as it would not be efficient or effective to store them in the database.

- Creating a menu system that means the users can navigate through the website.
- Creating a variety of templates for users to use in their sites.

The process of storing, previewing, and implementing the elements into a user's site still needs to be decided on.

- 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

I met with each stakeholder to propose the solution and gather feedback on it. This included identifying what features they would want in a website builder, how they'd like it to function, and, if they had some technical experience, whether they had any advice on the implementation of certain aspects of the project. 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:

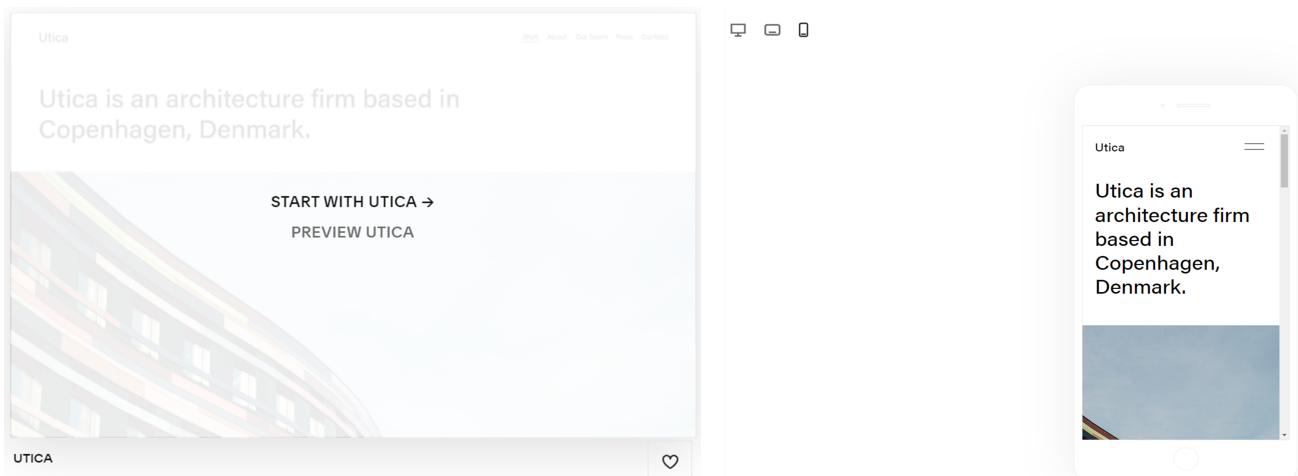
- Easy to use and user-friendly interface
- Ability to edit an existing site quickly
- Ability to invite others to collaborate on the site
- Import and export sites
- Enough customizability to make the website look as desired
- Styling customizability with global and individual element options
- Custom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

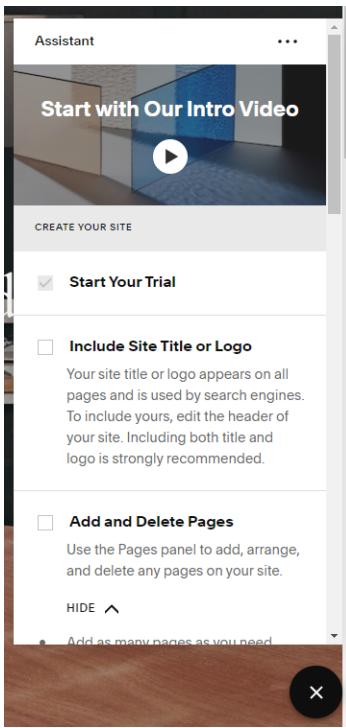
Research

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.

Design

- Site Styles**
- Browser Icon**
- Lock Screen**
- Checkout Page**
- 404 Page**
- Access Denied Screen**
- Social Sharing**
- Custom CSS**

Site Styles

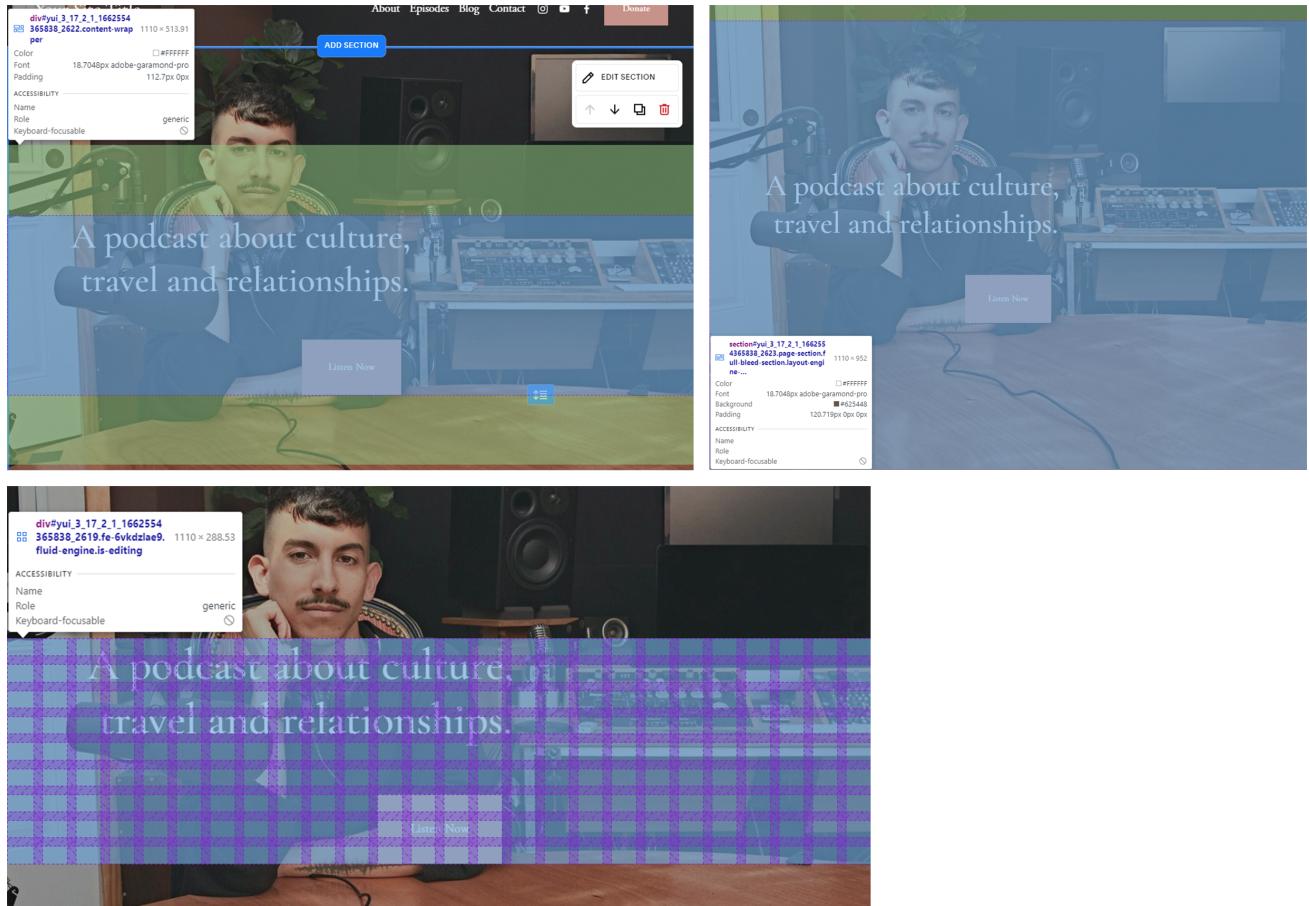
Manage the style settings that appear across your entire site.

- Fonts**
- Colors**
- Animations**
- Spacing**
- Buttons**
- Image Blocks**

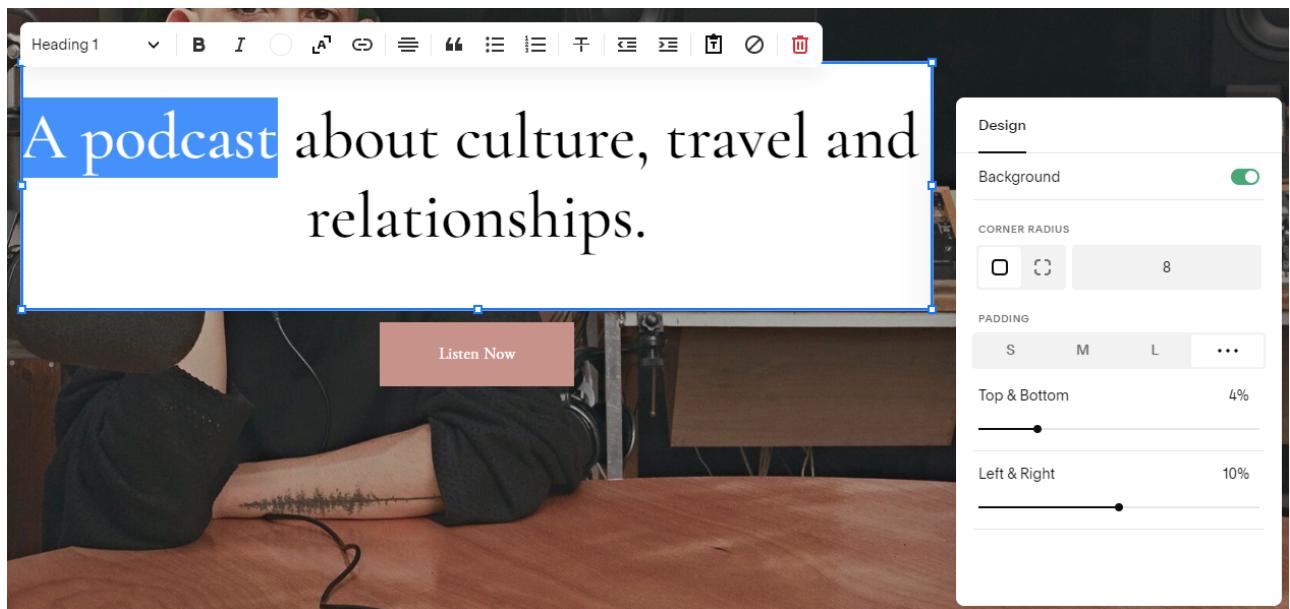
Their editor works in the conventional way of a grid-based system, where the user can place elements anywhere on the grid. It will then assign the item the style property `grid-area:row-start/col-start/row-end/col-end` or `grid-area:y/x/height/width` to define the position of the element. They have different 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.



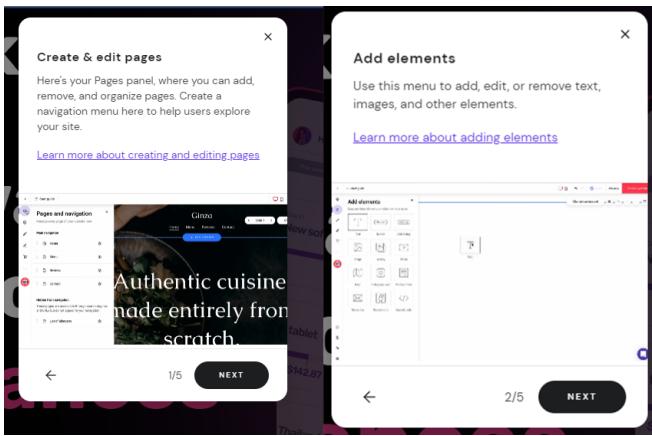
Parts I can apply to my project

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.

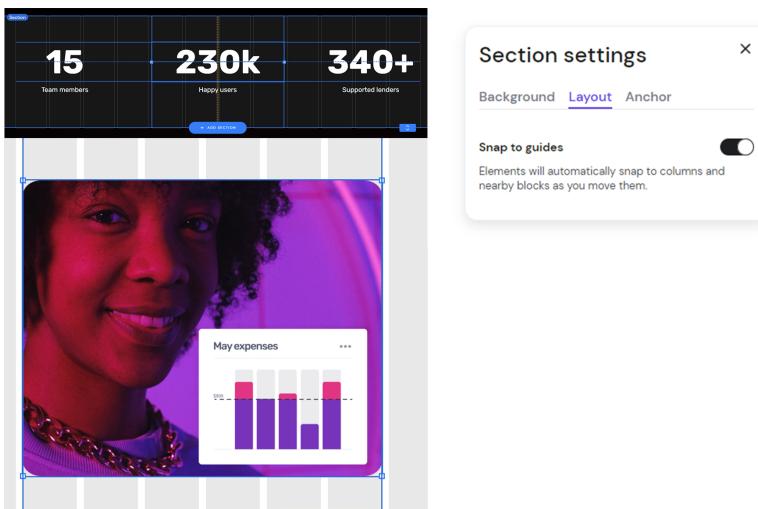
Squarespace also offers a variety of features such as e-commerce integration, SEO tools, analytics, 24/7 customer support, and a Content Management System (CMS) that allows users to update and manage their website's content easily. Although these features are helpful in a website builder, I intend leave 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.

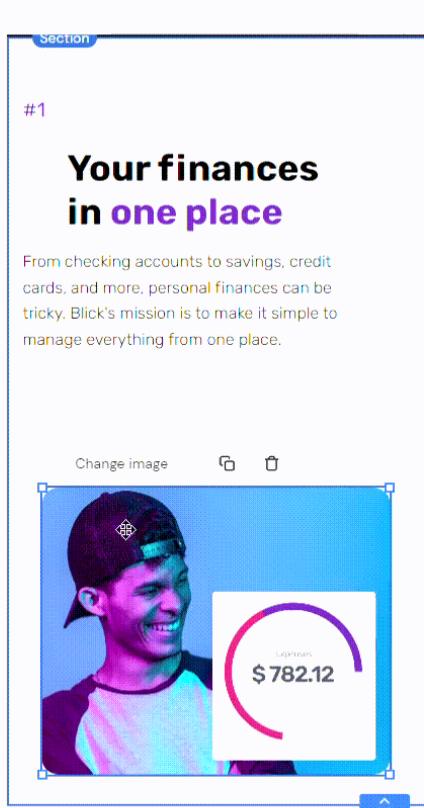
The CSS class with all of the variable references:

```
.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;  
}
```

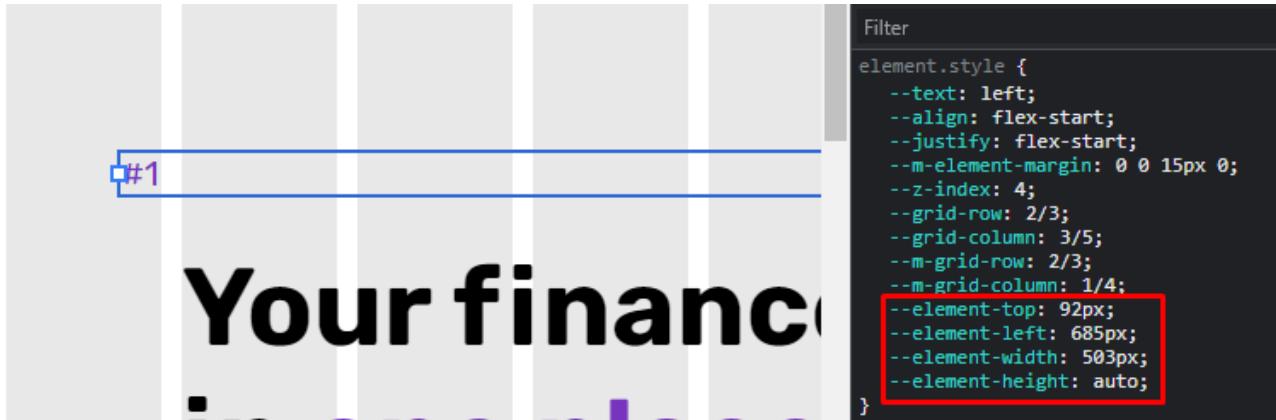
Their image resizing system is good. It uses the `object-fit: cover` property in the style of the image and changes the width and height attributes when being dragged, as explained later.



#2

Your finances

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 will be a multi-user, web-based program. It will be designed to allow for a wide range of users to create a website that meets their requirements. 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. Furthermore, if the website is used by more people, and more websites are created, the server may require more resources, impacting on the performance of the builder and making for a slower experience for individual users. This solution can be mitigated by implementing solutions such as load testing to identify bottlenecks and weakness points, so that they can be fixed. More methods are also brought up when mentioning hardware requirements later in this document.

While the intention for the project is to create an easy-to-use platform for creating professional and functional websites, it may not be the ideal solution for every user. For clients with more complex needs, who require a bespoke website with customised functionality, this builder may fall short of expectations. As the builder prioritises accessibility and a low learning curve, it may lack the level of customisation and responsiveness that certain clients need. This can be reduced by the ability for users to add custom code to their projects, however those with more specific needs may be more inclined towards hiring a developer or using a more advanced builder.

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, the project will need to 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 & grey colours, and primary and secondary fonts.

- 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.

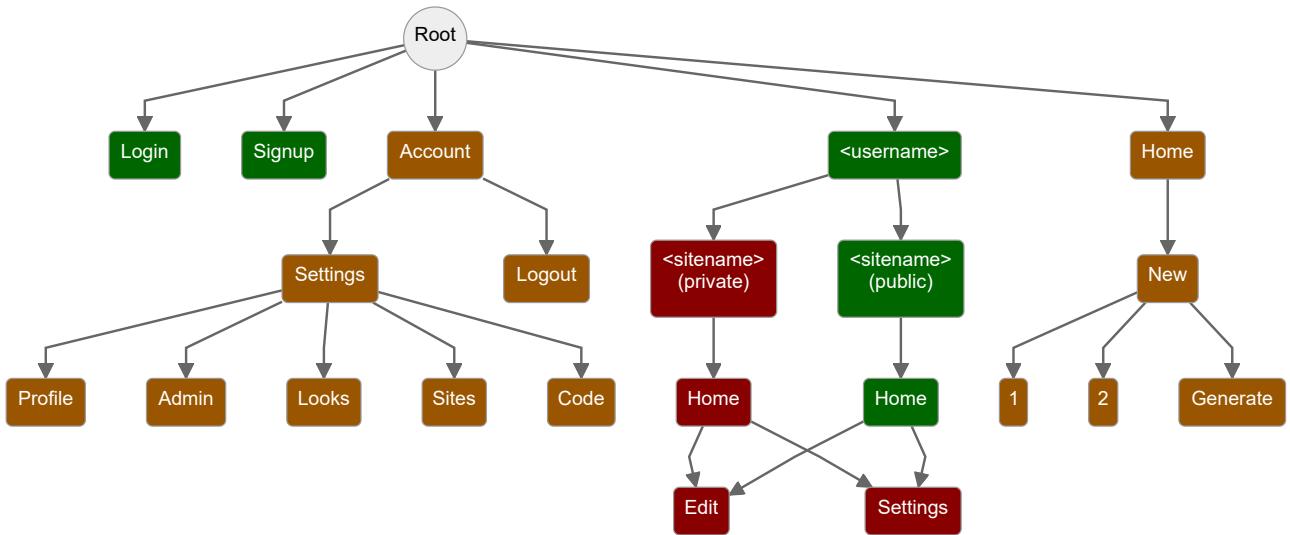
- The site owner can assign other users the ability to edit public or private sites, but there cannot be two people editing simultaneously.

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

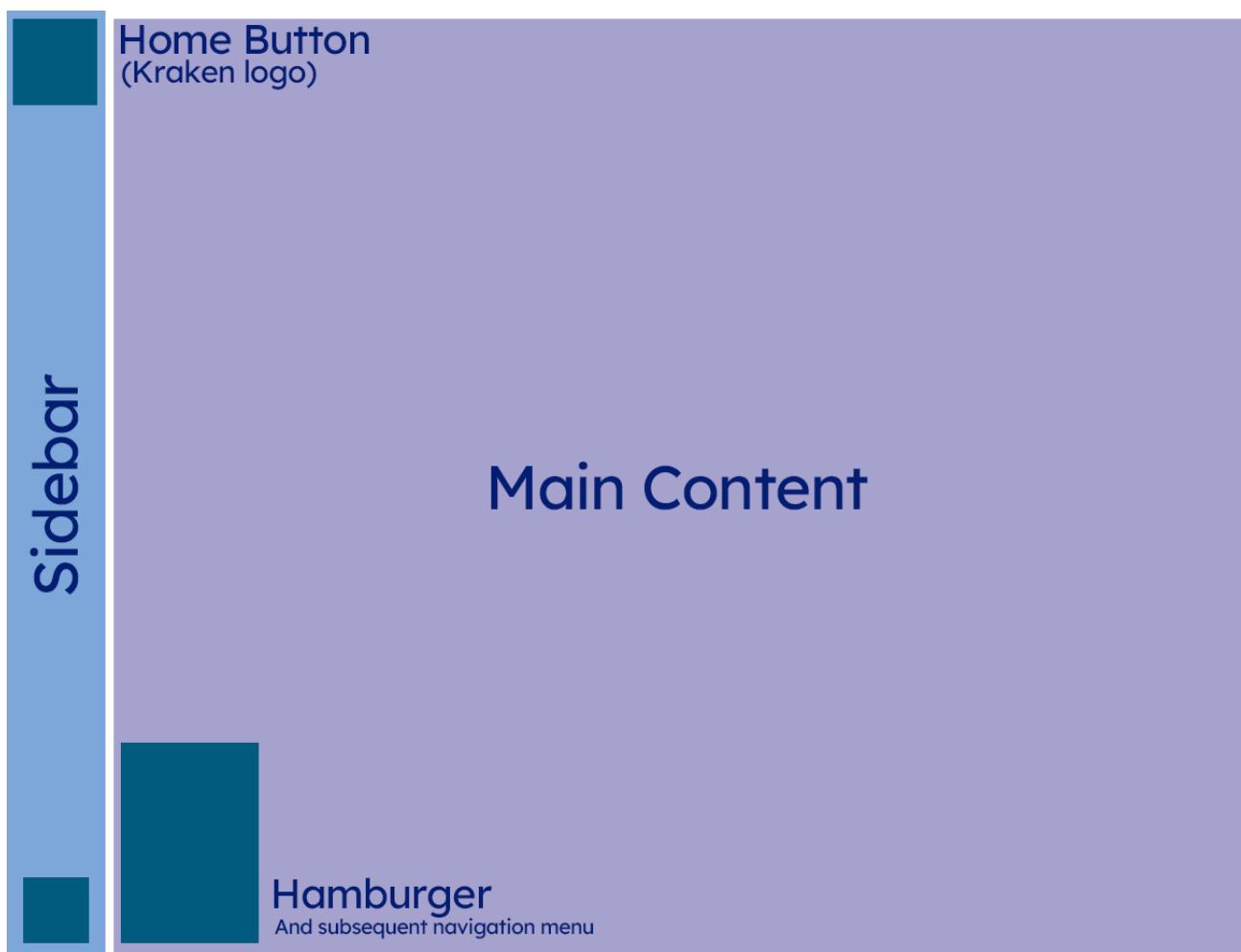
URL Navigation



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.

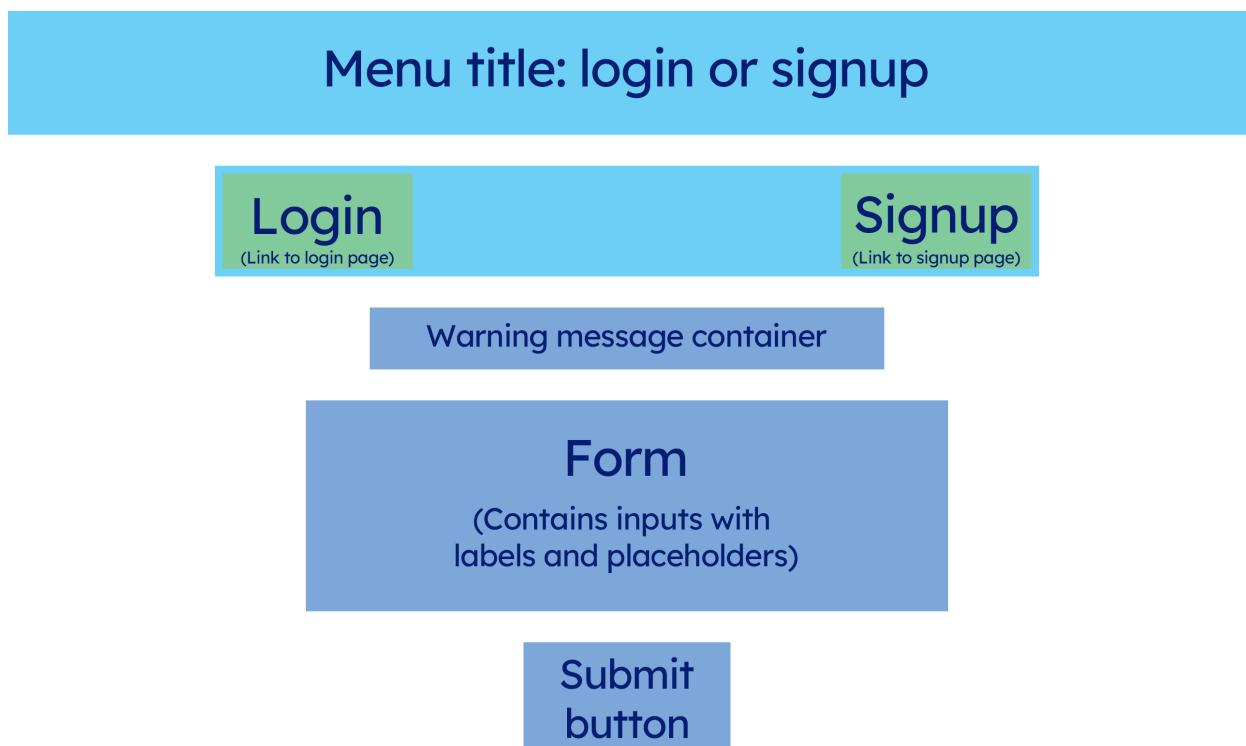
Hamburger

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

“Welcome, <username>”

Grid of user’s websites

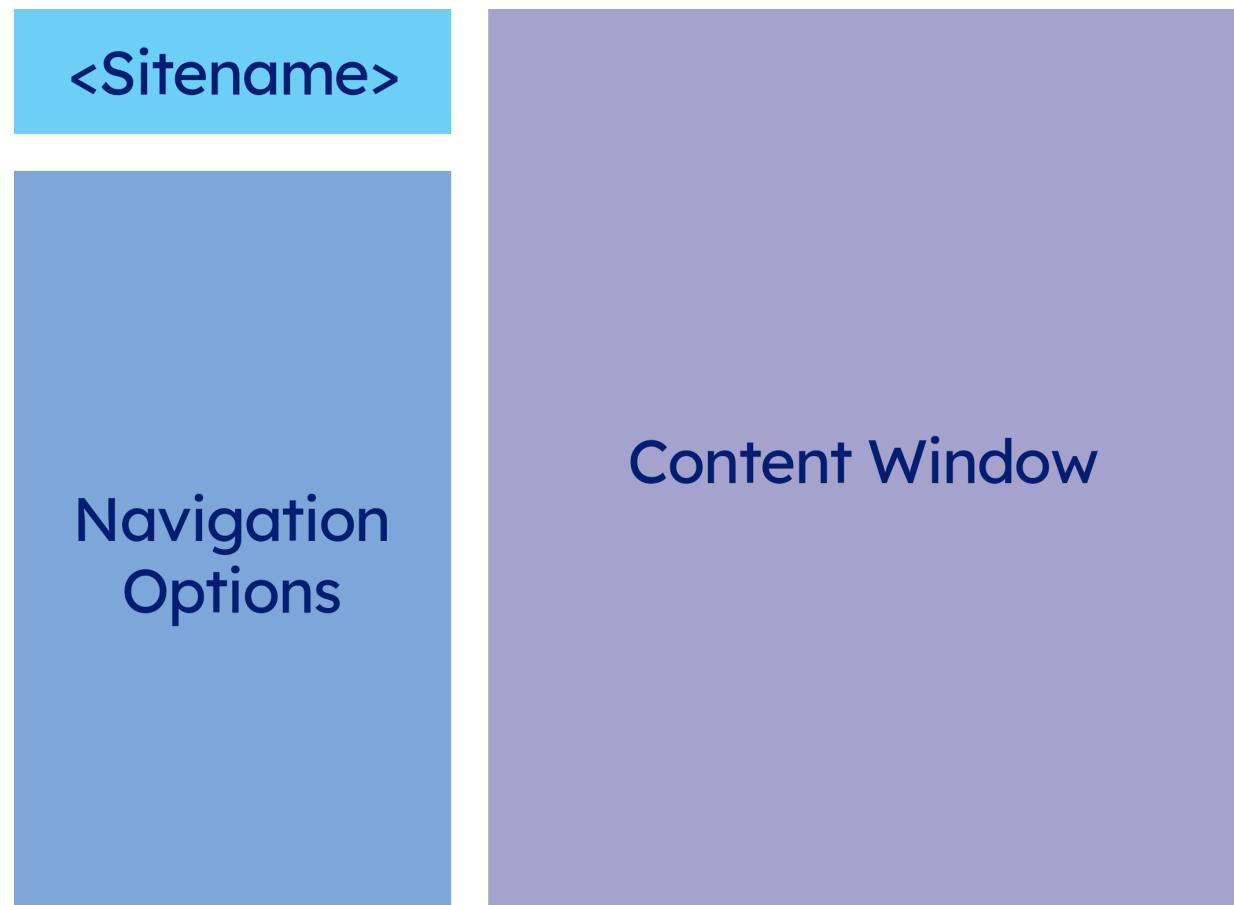
Create new site option

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.



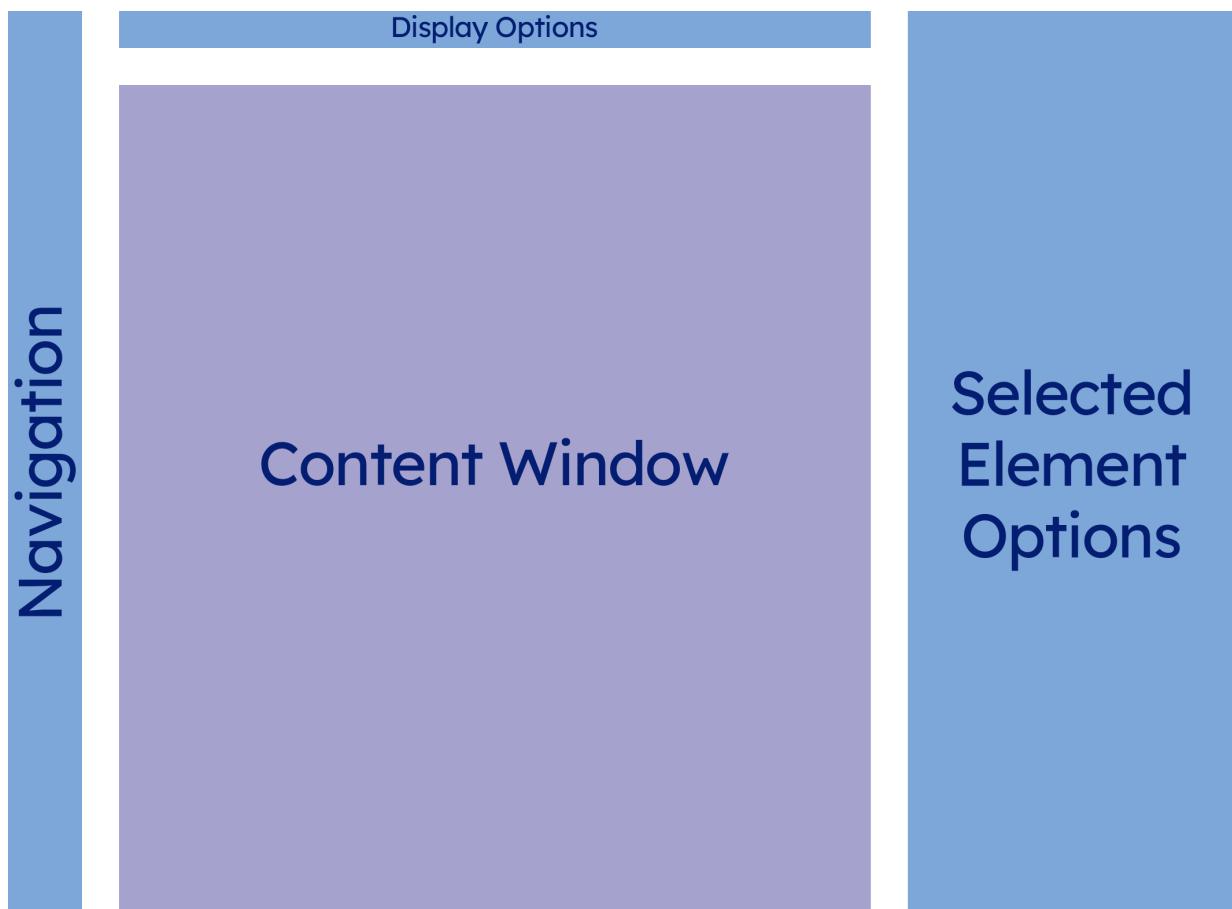
This shows the layout of the site page for the owner (when they visit `/<username>/<sitename>`). Built inside the main template, it contains the following:

- 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.

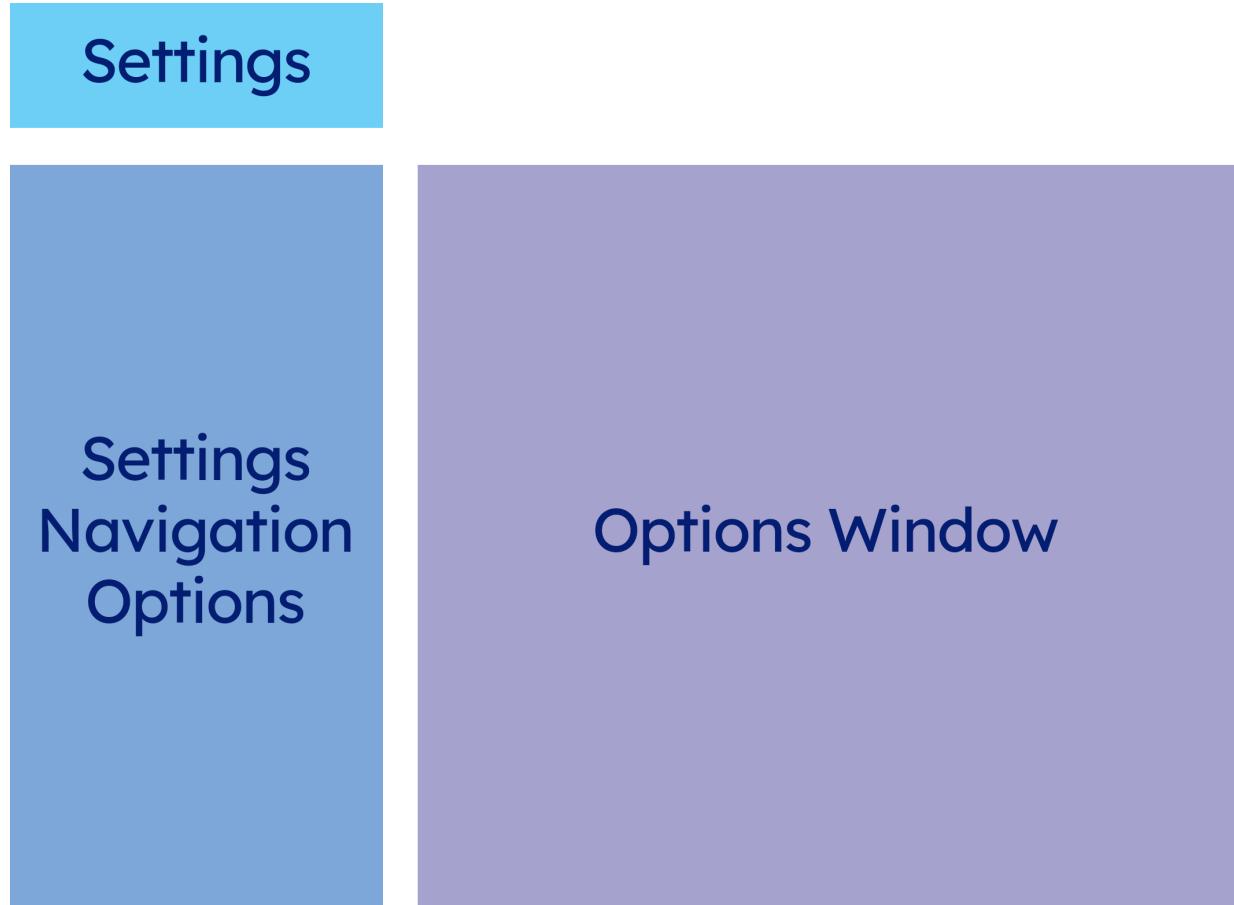
Some of these options will be: Add section, add element, website pages, website styles, and website settings

- 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) is a syntax system developed by the W3C Web Accessibility Initiative to allow for screen readers and other accessibility features when navigating websites on the internet. Quoted from [their overview on w3.org](https://www.w3.org/WAI/ARIA/):

"WAI-ARIA, the Accessible Rich Internet Applications Suite, defines a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with HTML, JavaScript, and related technologies."

ARIA attributes will be used throughout the website to allow screen readers and other accessibility features to use the website, despite the small chance that they would use it. It is also intended to be implemented into the creation of websites via the drag-and-drop editor, using JavaScript to programmatically assign appropriate ARIA tags to elements that require them.

Stakeholder input

Website structure and backend

Flask

I have decided to use the Flask Python library as the backend for this website. This is because I have prior experience in using it, it suits this project, and was suggested by Kevin, one of the stakeholders. It is also appealing due to it being well documented, relatively lightweight, and easy to set up. Despite its lack of built-in features when compared to other libraries (such as Django), there are many other third-party Python libraries (including as `flask-login` and `flask-sqlalchemy`) that add in more functionality that is missing from the framework, and that is required for this project. The one thing that I will have to keep in mind if the project gets significantly larger is that it may not be suited as well to large amounts of traffic, in comparison to other frameworks.

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, as it is the native directory that Flask looks for, making it more intuitive for other potential developers.

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

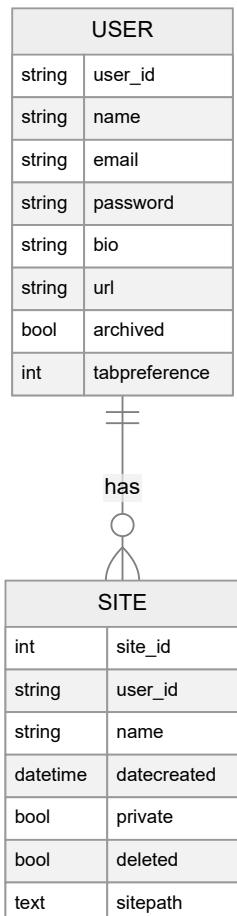
There will be two different methods of storing information for the website:

- 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

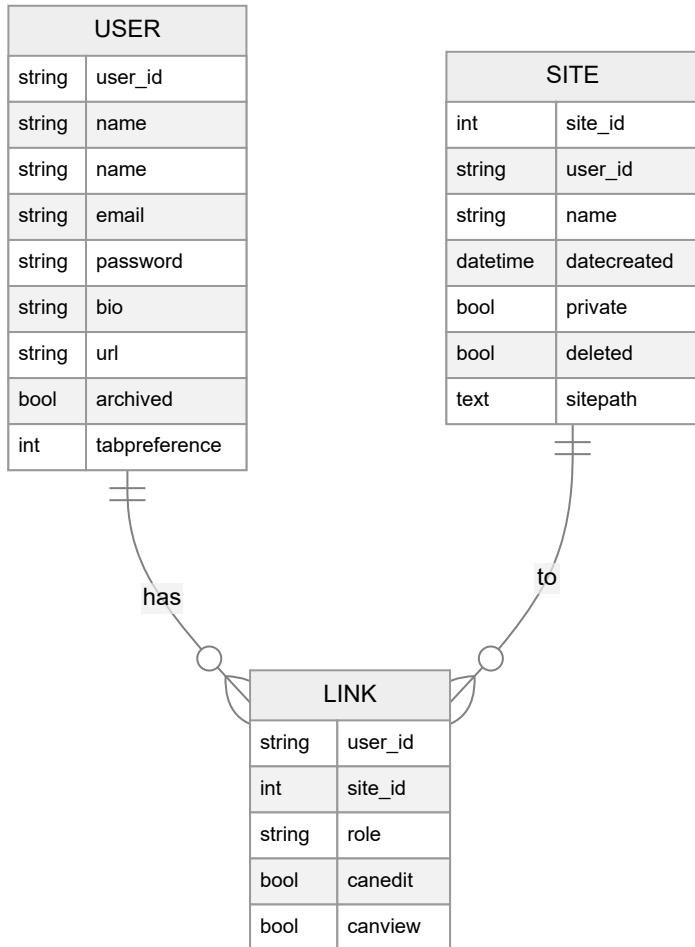
SQL will be used to store data for the multi-user system, including parameters such as the username, email, and password. I have had previous experience with SQL, meaning that it will be easy for me to set up. Its entity-relationship system makes it well suited for storing user and site information, as it can be linked together. The Python library `flask-sqlalchemy` also means that it can integrate into the backend framework that I have chosen. This means that there will be less work for me, as the functionality needed is already built-in and will have been extensively tested. Furthermore, due to the SQLAlchemy library not requiring any queries inputted directly by my code, it alleviates the risk of SQL injection attacks.

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`.



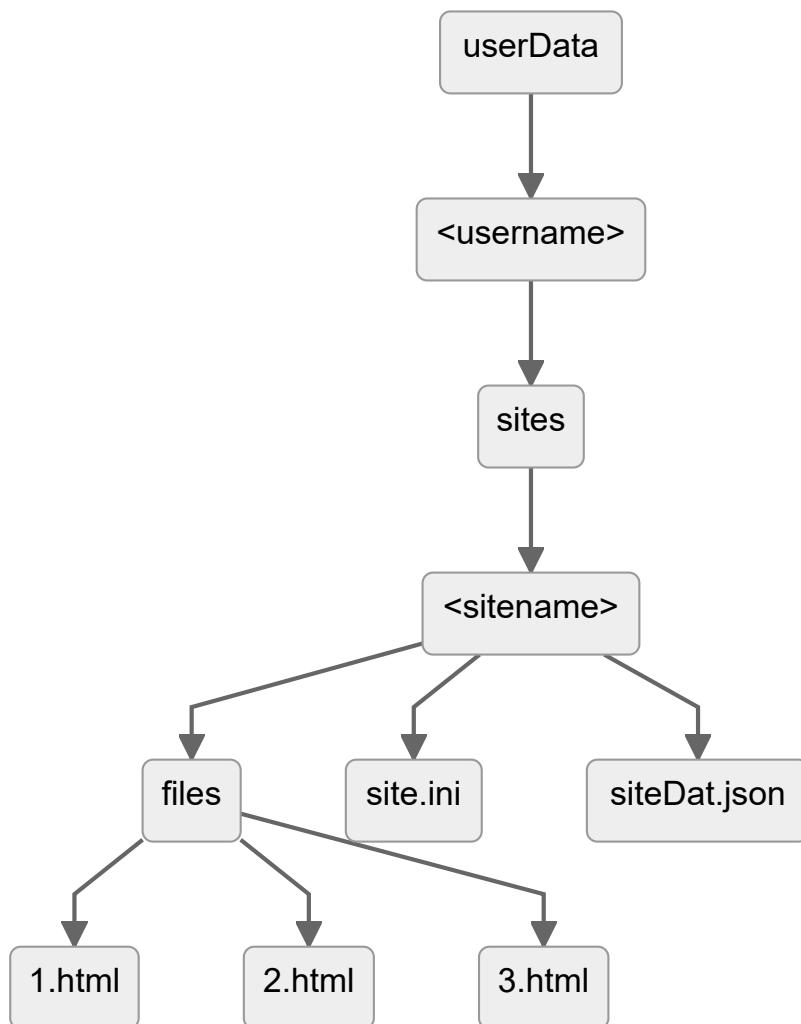
To incorporate a multi-user editing system for certain sites, the entity relationship diagram for the database will look like this. However, this may not be implemented, as it is only a desirable feature and is not required for the project to function.

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

The user's website files will be impractical to store in SQL, so will instead be stored in the server-side storage, and accessed by the Python `os` module. The server-side storage means that a user will be able to access their websites from any location, assuming that they have an internet connection. The syntax for storing 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 elements of this solution are: using SQL to store user data for the multi-user system, the UI design and interactivity, and the JavaScript for the drag-and-drop editor, with the latter being the most complex. To ensure security is maintained when user data is being sent to the server, a lot of algorithms, such as `verifyField`, will be developed and tested in both languages. This will alleviate the risk of an attacker bypassing the client validation, whilst also meaning that the load on the server is reduced if a user inputs invalid data.

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:

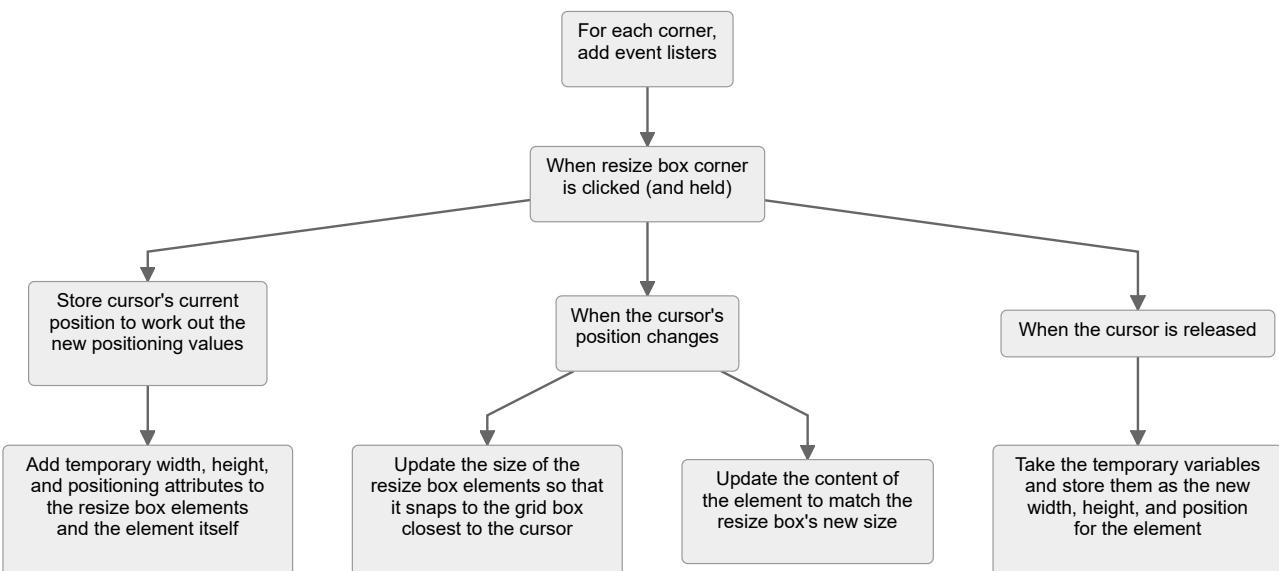
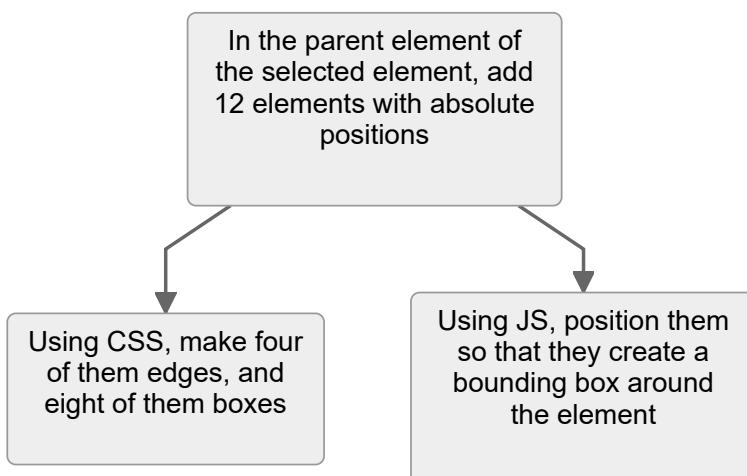
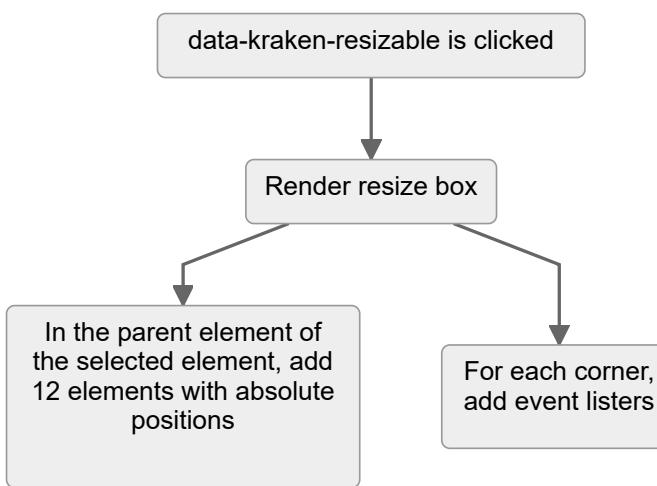
- `data-kraken-resizable`
- `data-kraken-draggable`
- `data-kraken-editable-text`
- `data-kraken-editable-style`
- `data-kraken-locked`

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.

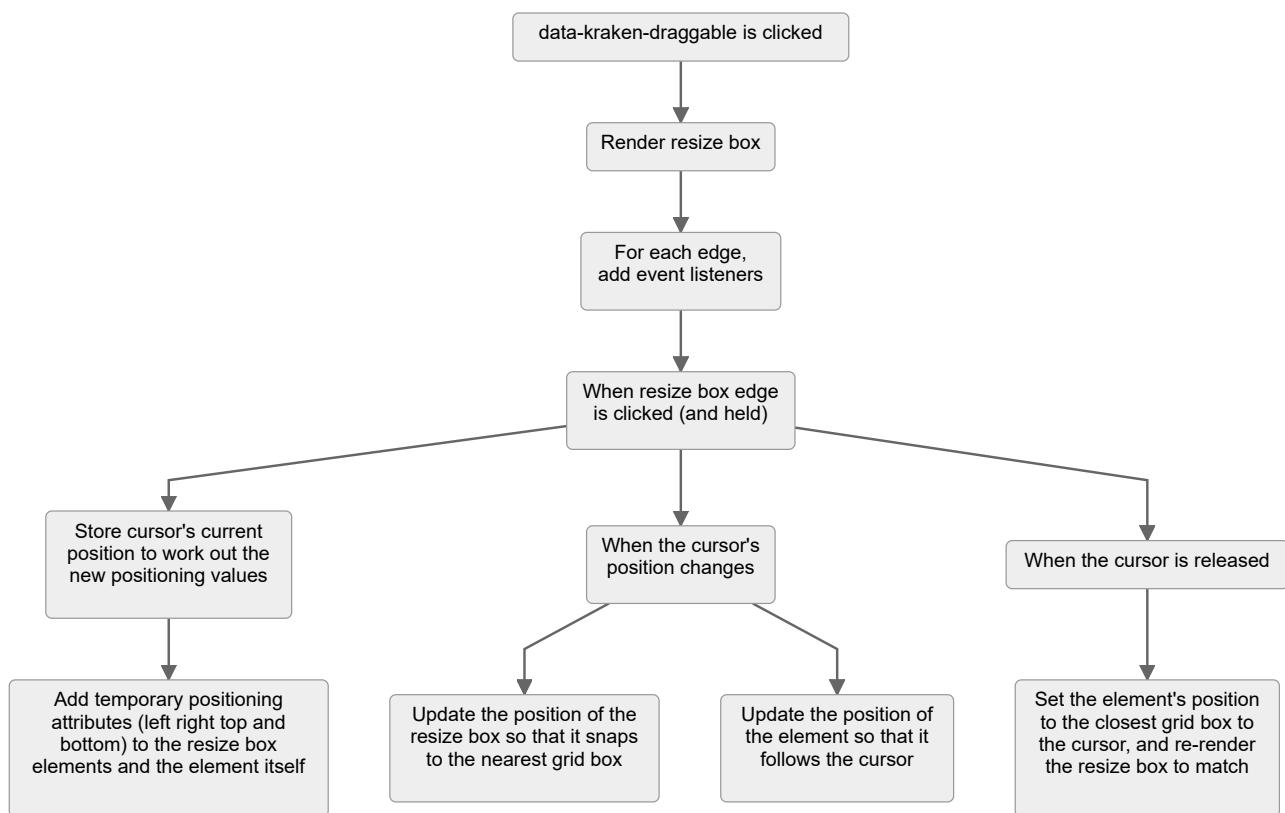
When an element is selected, depending on its function, it will be tagged with one of these attributes so that the JavaScript can easily edit it (more may be added during development):

- `data-kraken-selected-text`
- `data-kraken-selected-style`

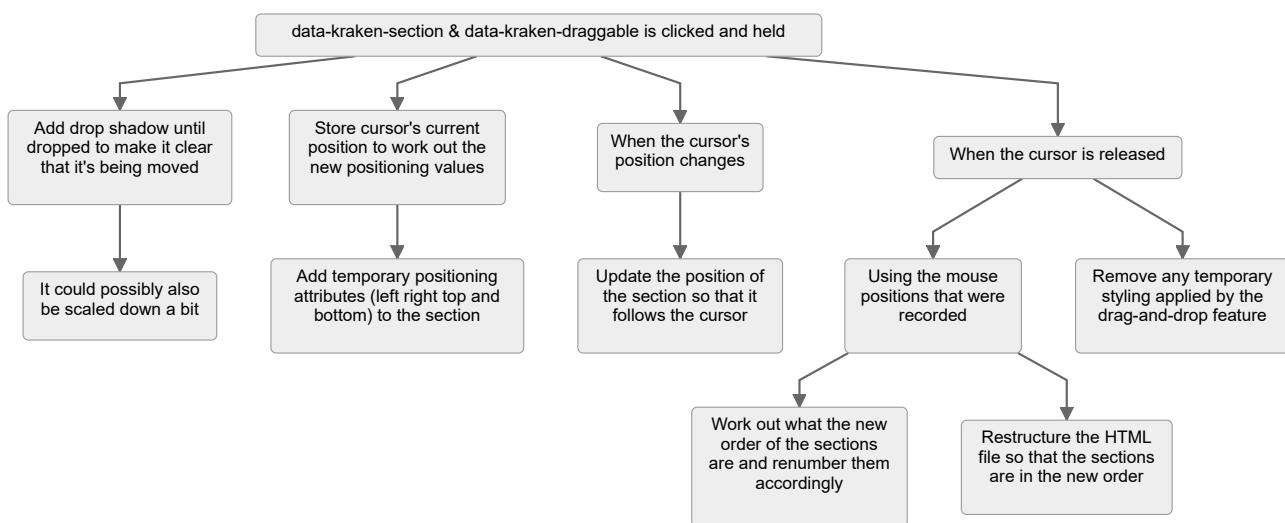
Resize box



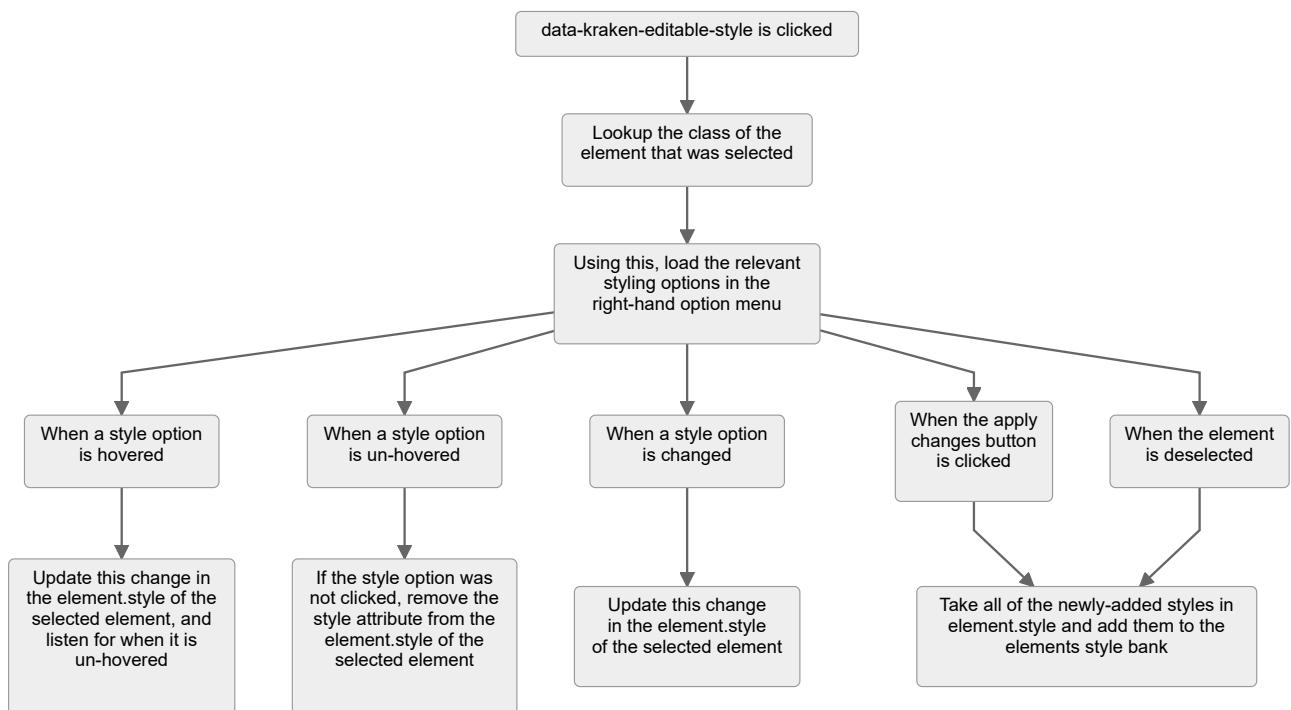
Dragging and dropping elements



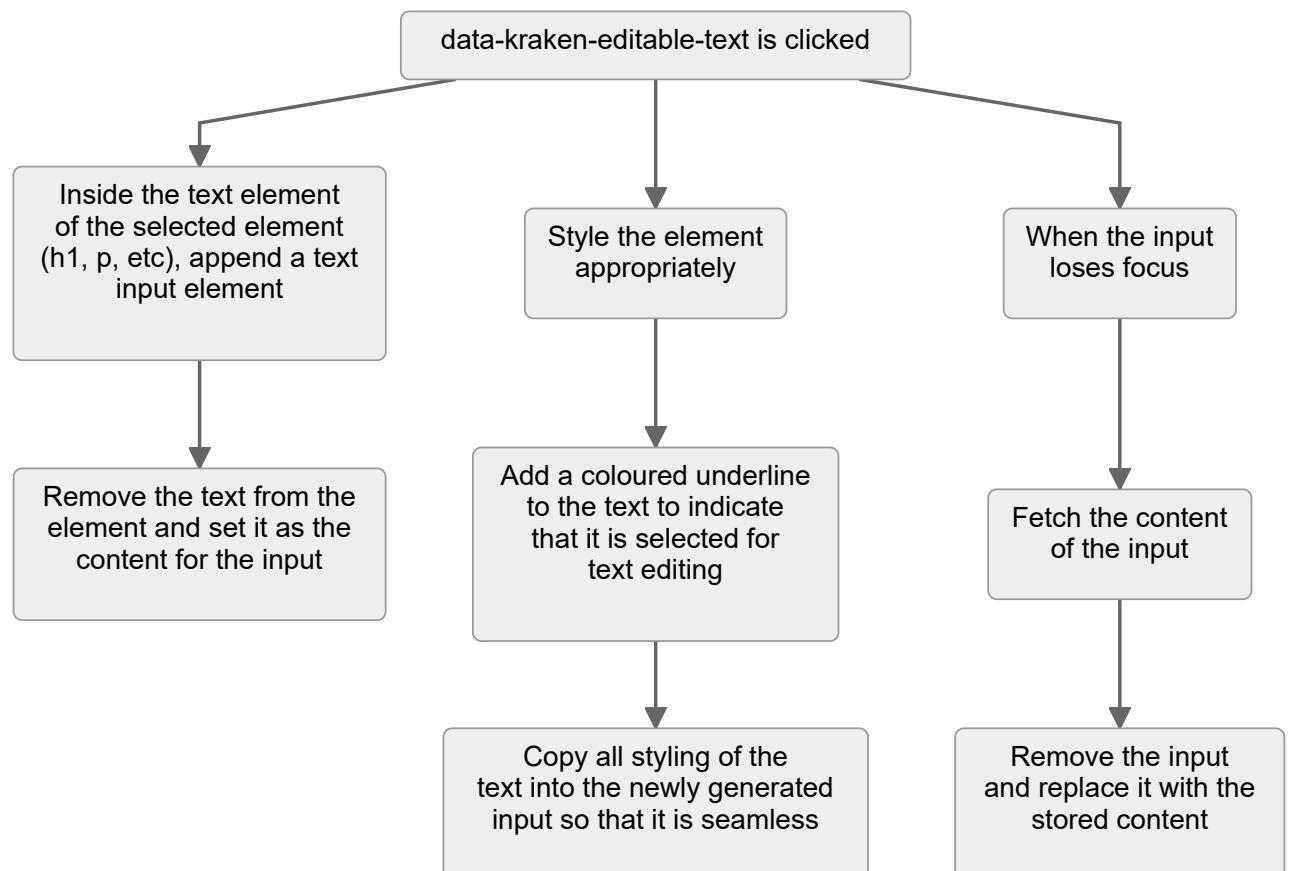
Dragging and dropping sections



Displaying element and section options



Displaying text editors



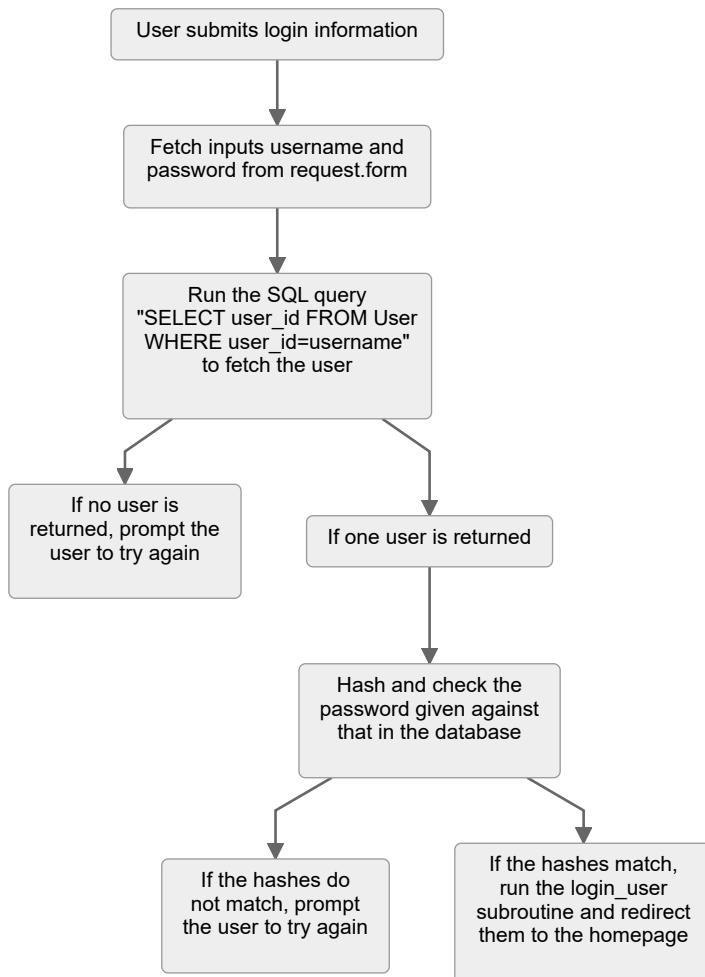
Multi-user system algorithms

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.

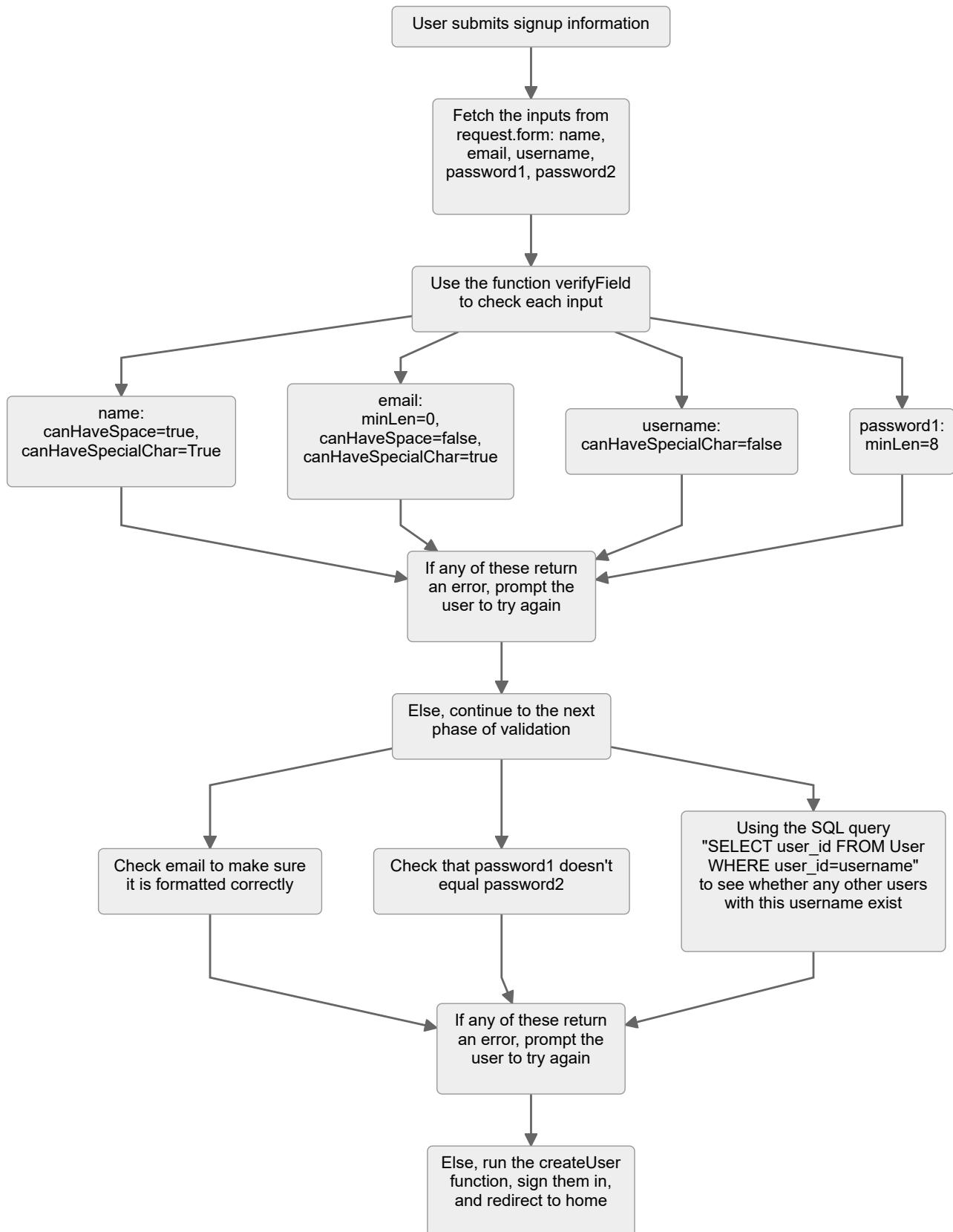
The main things that the multi-user system should be able to do are:

- 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

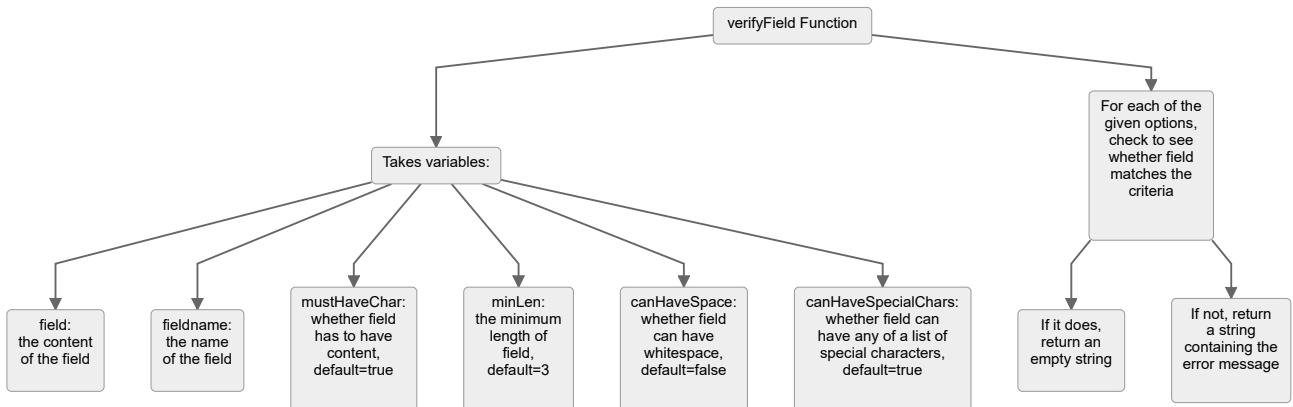
Login page



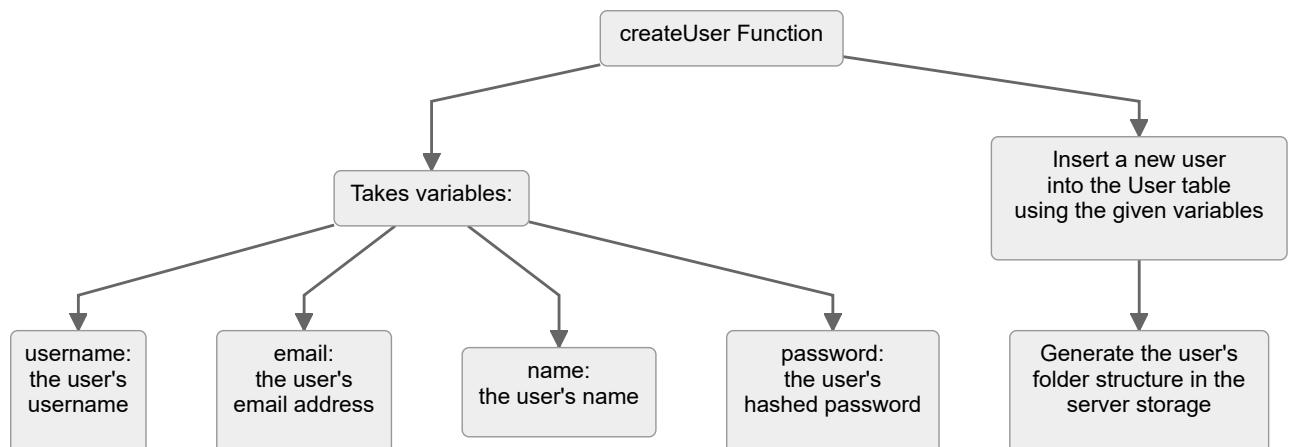
Signup page



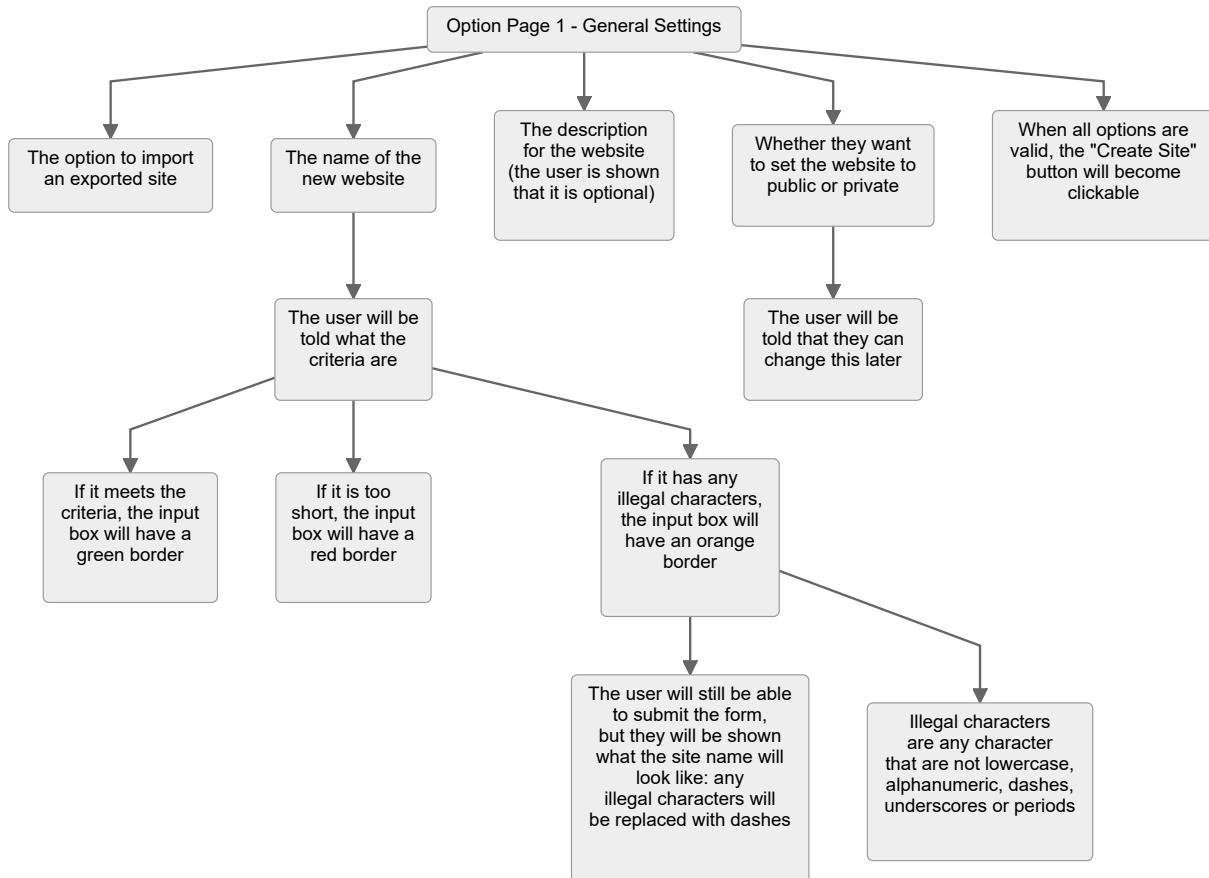
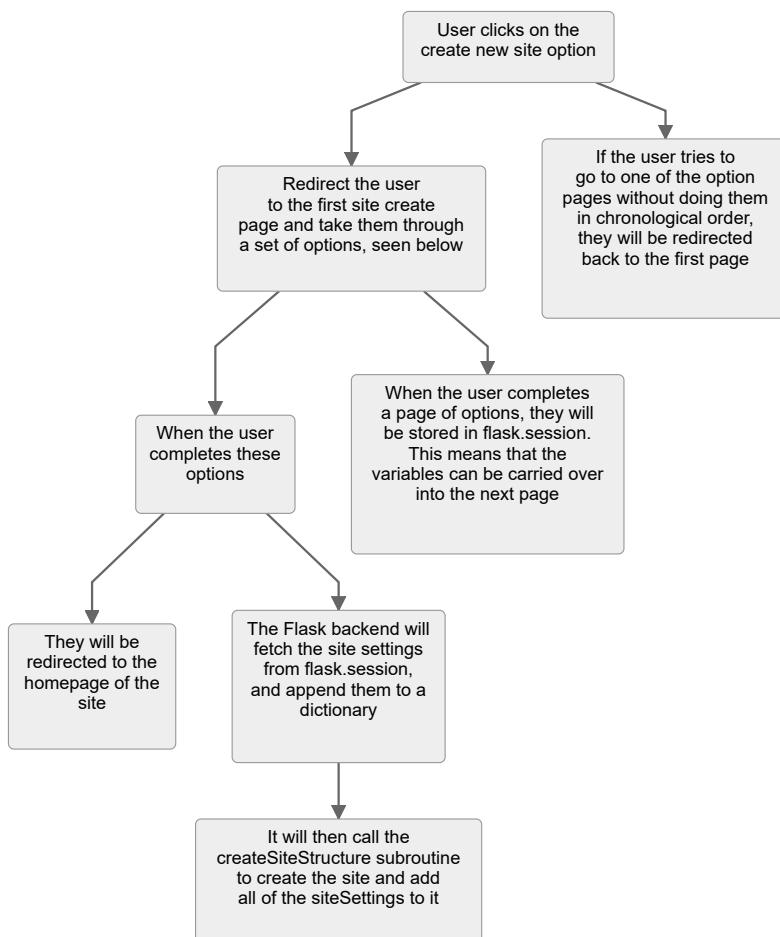
Field verification



Creating a new user



Creating a new site



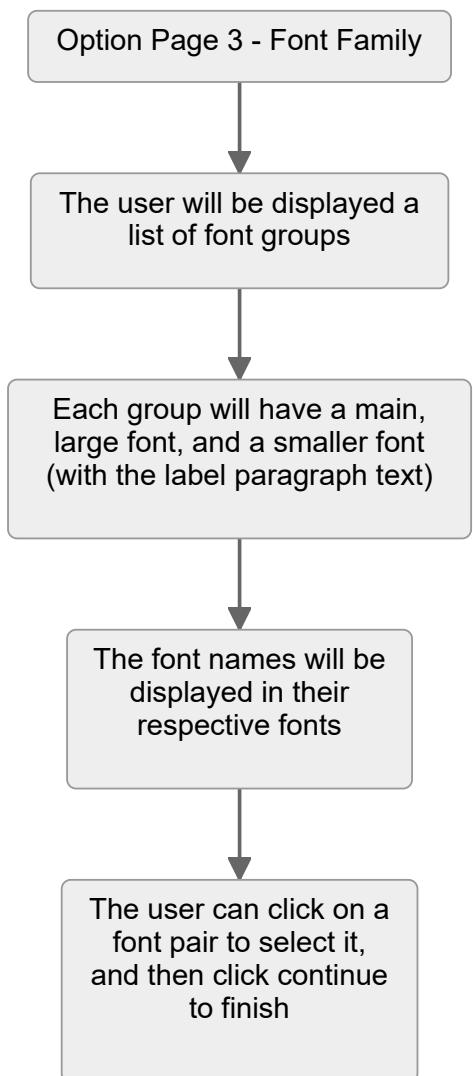
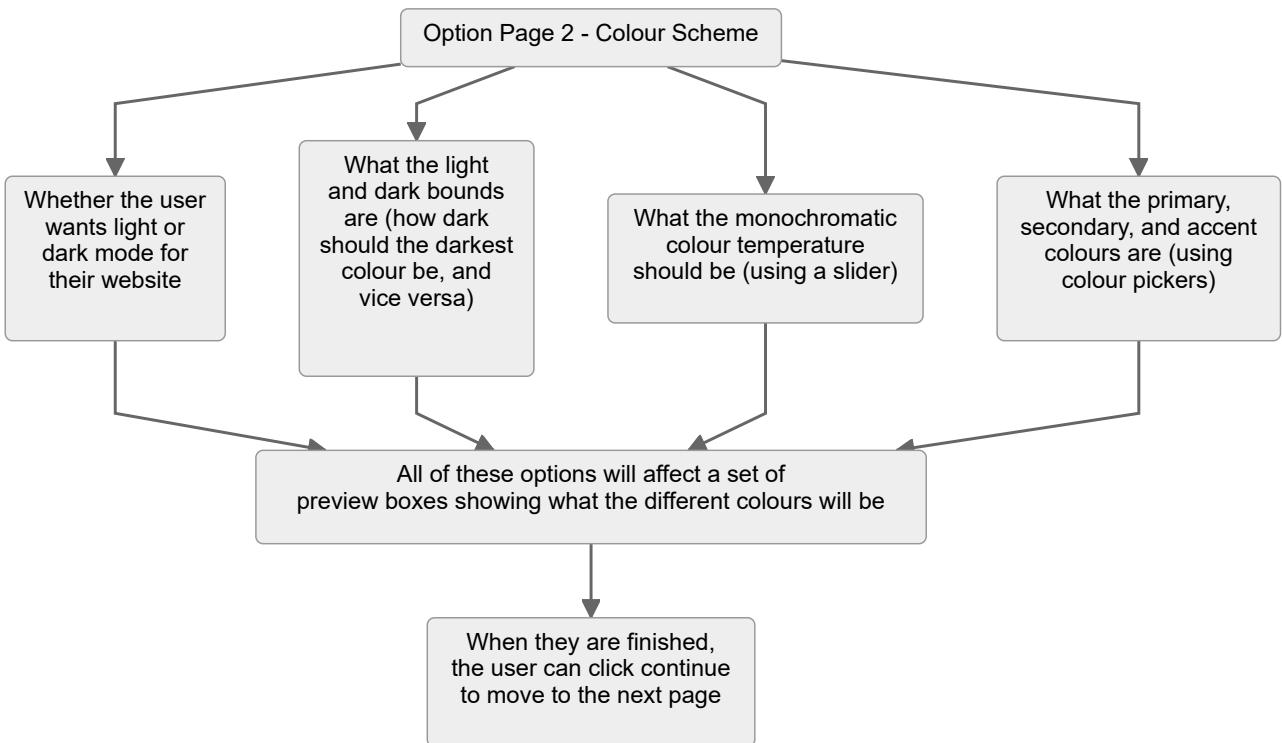


Diagram showing how the subroutines link

Subroutines

Now that I have an idea of the requirements for the subroutines and how the different modules will fit together, I can start writing pseudocode to plan them. The multi-user subroutines will be written in Python, and in some cases JavaScript, whereas the subroutines for the website builder will be written primarily in JavaScript. For now, all of the subroutines are written in pseudocode

I anticipate the JavaScript files to be quite large after development has finish, so I may also separate the subroutines out into separate files to make navigating the code easier. Each page will have it's own JavaScript file, so that only the required code is being imported

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(
        "SELECT sitename from SITE where userid={session.user.id}")
}
```

```

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

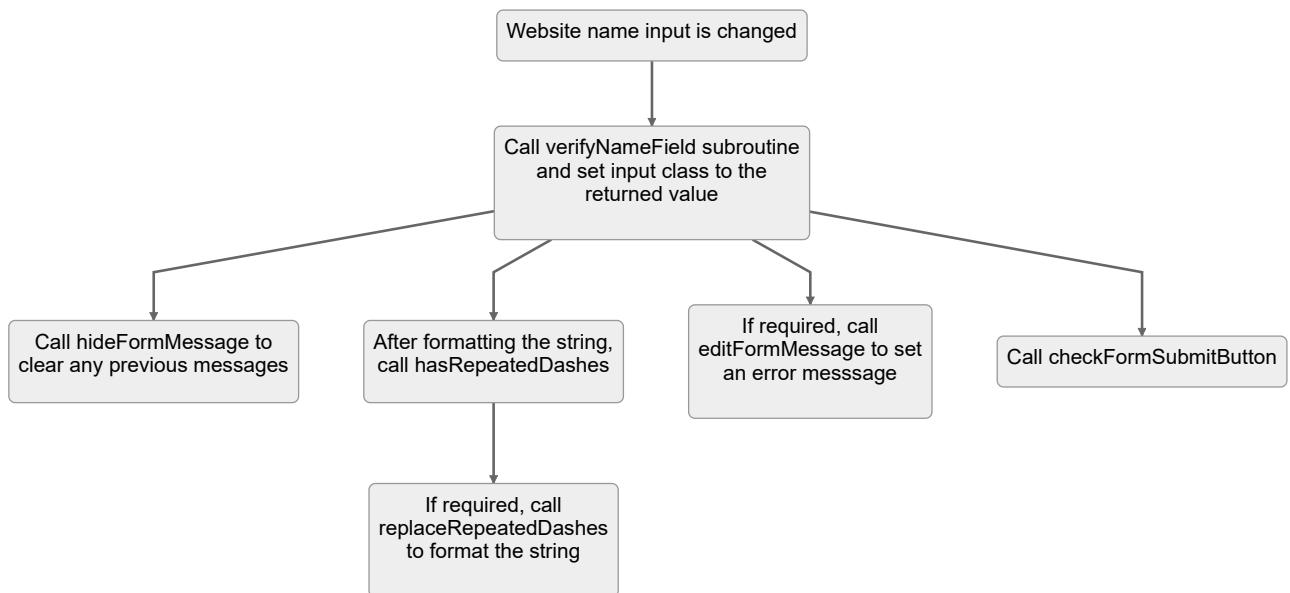
for (letter of val) {
    if (letter not in "qwertyuiopasdfghjklzxcvbnm-.1234567890") {
        editFormMessage(val)
        return "warning"
    }
}

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.b)
        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.b)
        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

Although the initial concept for this solution appears large and complicated, the decomposition into separate parts will make the development easier and quicker, and will also aid the testing and maintenance of the code, due to its 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 demonstrate this implementation. 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 in a clear and linear fashion 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, as most of the code doesn't rely inherently on what came before.

The code will be designed to be modular, so that development and modifications down the line are easier. This will be achieved by following the principles set out in this design. For example, separating the multi-user system, editing system, and user interface. The overall goal is that, if another developer were to take over the development, they would be able to easily comprehend how the code works and fits together. To further aid this, the code will be well documented with clear and concise comments throughout the code, along with function definitions.

The two main programming languages, Python and JavaScript, will communicate via Flask's `session` and `flash` functionality, ensuring the languages can interact as expected. The Python functions will be contained in an object representing the full project. This means that it can make use of the `self` communication to easily pass frequently used variables between different functions. It will also use variables and `return` statements for some subroutines, where required.

Inputs and Outputs

Input	Process	Output
Login submit button	<code>auth_login_post</code> to verify user input.	Log in the user to the session, or provide a suitable error message.
Signup submit button	<code>auth_login_signup</code> 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.
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	Display appropriate style settings in right hand dock. Display

Input	Process	Output
	down, or dragging on the bottom to resize, if necessary.	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 <code>x.x.x.x</code>	Defines the host the server uses
port	integer	Defines the port the server uses
databaseObject	<code>flask_sqlalchemy.SQLAlchemy</code> object	Access the SQL database
loginManager	<code>flask_login.LoginManager</code> object	Manage the user login system
app	<code>flask.Flask</code> object	Manage the site routing

These are the main variables that the JavaScript 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	The selected colour options when

Name	Data Type	How it is used
	strings	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.
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

There is always the possibility that the code will break at some point, and where it does can have a large impact on the effect of this. For example, JavaScript errors will simply halt any further processes, meaning the user will have to refresh to resume functionality. However, this can be unintuitive and frustrating for the user. In contrast, flask will catch most errors, and throw a `500 Internal Error` that the user will see, causing a similar annoyance to the user. In the rare chance that something very bad breaks, such as the database being corrupted, the entire process will be stopped and the server will go down. As such, these, and many other possibilities, need to be accounted for and the risk mitigated as much as possible.

One way of reducing said risk is via thorough validation throughout the program. At any point when the user inputs information, both the client-side and server-side programs should ensure that the data is accurate, complete, and conforming to any requirements set. Some of this is aided by the usage of HTML `<input>`s for user input. This means that HTML can make sure that the correct data type and format is being inputted. However, a lot of data requires specific validation to ensure 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

The JavaScript will take the content of the input and replace any invalid characters with dashes, and then display the final result to the user, so that they know what the name of their site will be. The code will also insure that the form can only be submitted when the user has filled out all required information. This process is outlined earlier in the document.

This is an example of client-side validation, where JavaScript ensures that the data is valid before sending it off to the server, reducing the load on the server. This validation will still be executed server-side as well.

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 Sanitisation

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.

A good example of this, that will require a lot of validation and checking, is the ability to import sites and user content into the server. This, if unchecked, is a good opportunity for attackers to insert malicious code into the server, and, as such, any content uploaded like this must be sanitised thoroughly to ensure that the integrity of the server is maintained.

Testing method

During development, the vast majority of alpha testing will be white box testing, performed by the developers as the project is being coded. Unit testing will be used to make sure that all of the individual modules function as expected and intended, before combining them together. By testing each module separately, it ensures that, when all of them are combined together, the program will function correctly. Integration testing will make sure that the program functions as a whole, including checking how different modules interact with each other, and how the front-end interacts with the back-end.

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 either be recorded during the development process or recorded in Appendix B, 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 reorganisations of the page, would be displayed in later sections, but the actual process of originally designing the website is not documented to make the document easier to read. A lot of the design process involved tweaking classes, re-ordering HTML elements, and adding new or modifying existing CSS blocks.

Stage 1 - Setting up the website

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

`_init_.py`

```
from flask import Flask

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

        # Initialise the website pages
        self.initPages()

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

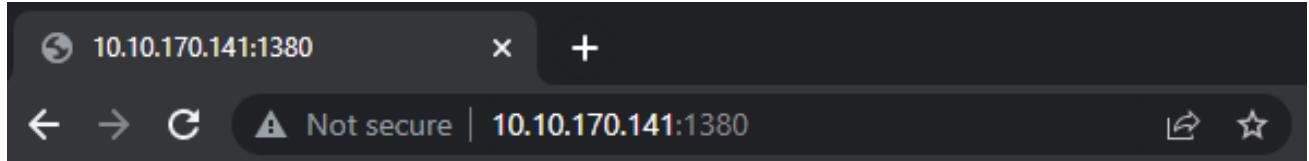
    def initPages(self) -> None:
        # Home page route
        @self.app.route("/")
        def main_index() -> str:
            # Display the returned text
            return "This is the homepage!"

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

When run, it would output this to the console:

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

And look like this in the website:



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

changes to `__init__.py`

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

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

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

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

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

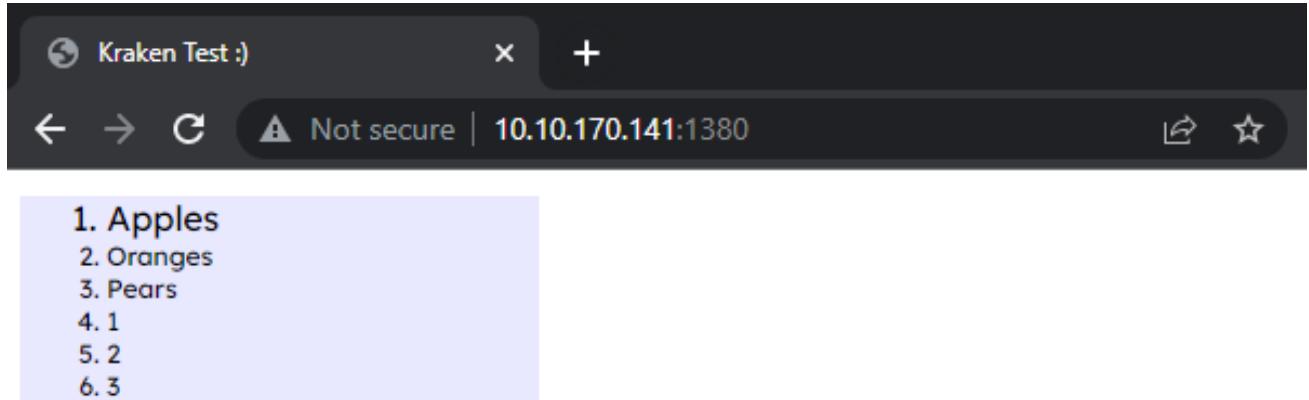
/templates/test.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Kraken Test :)</title>  
  
    <link href="{{url_for('static', filename='css/test.css')}}" rel="stylesheet"  
          type="text/css" />  
  </head>  
  
  <body>  
    <ol>  
  
      {# Iterate through the list in the first flashed message, and create a  
         list element for each one #}  
  
      {% for element in get_flashed_messages()[0] %}  
        <li>{{element}}</li>  
      {% endfor %}  
  
    </ol>  
  </body>  
</html>
```

/static/css/test.css

```
ol{  
  background-color:#e8e8ff;  
  font-size:12px;  
  font-family:Lexend,sans-serif;  
  width:200px;  
}  
  
li:first-of-type{  
  font-size:16px;  
}
```

When run, the website looked like this:



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

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

changes to `__init__.py`

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

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

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

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

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

    <title>Kraken</title>

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

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

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

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

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

  </head>
  <body>

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

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

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

```

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

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

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

    &lt;a class="globalnav-floating-option one" href=""&gt;
        &lt;span class="globalnav-floating-option-content text header small"&gt;
            My Sites&lt;/span&gt;
    &lt;/a&gt;

    &lt;a class="globalnav-floating-option two" href=""&gt;
        &lt;span class="globalnav-floating-option-content text header small"&gt;
            Settings&lt;/span&gt;
    &lt;/a&gt;

    &lt;a class="globalnav-floating-option three" href=""&gt;
        &lt;span class="globalnav-floating-option-content text header small"&gt;
            Logout&lt;/span&gt;
    &lt;/a&gt;
&lt;/div&gt;

<!-- Backdrop behind nav bar modal to apply a darkness filter behind
the modal --&gt;

&lt;div class="globalnav-floating-options-backdrop"&gt;&lt;/div&gt;
</pre>

```

```

<!-- 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 %}

```

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

{% block content %}

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

    <div class="field-container active">
        <!-- Warning area for the form that uses the flashed warning message -->
        <span class="field-warning text italic">
            <!-- TODO: add code for flashed warning msg -->
        </span>

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

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

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

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

```

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

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

</form>

</div>
</div>

{% endblock %}

```

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

For example, the URL for the home button image in `base.html` is defined by

```

{{url_for('static', filename='img/icon/512-512/kraken-icon-png-
'+navbarLogoColor+'-512-512.png')}} . The variable navbarLogoColor is declared in child
templates to define which colour should be used.

```

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

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

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

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

Syntax for `/templates/main.css`

```
Global classes:  
.notextselect  
.nopointerevents  
.visibly-hidden  
.fake  
.box  
.no-inversion  
  
Positional Classes:  
.relative  
.sticky  
.fixed  
.abs  
.static  
  
Text classes:  
.text <light|dark|primary|secondary|accent|grey-100 => grey-800> [italic]  
[bold|thin] [ellipsis] [xl|large|default-size|small] [header|jumbo]  
[lowercase|uppercase] [notextselect] [left|center|right|justify]  
  
Link classes:  
.text .link [classes for text] [disabled] [link-slide] [notformatted]  
(link-slide class requires the css variable --link-slide-width)  
  
Btn classes:  
.btn <light|dark|primary|secondary|accent> <square|rounded|pill> [slide]  
.btn.slide [from-left|from-right]  
  
If slide is set, the btn must contain a span element with syntax:  
    span .btn-content .text (all text classes apply here)  
which contains the text of the button  
  
A span like this is recommended even if the .slide class is not present so  
you can format the text inside separately  
  
em classes:  
[classes for text]
```

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

/static/css/build.css

```
.globalnav {  
  background: var(--colors-grey-200);  
  position:fixed;  
}  
  
.globalnav-floating-options {  
  transform:scale(0);  
  transform-origin:bottom left;  
  opacity:0;  
  margin:0;  
  transition:transform,opacity,margin,visibility;  
  transition-delay:130ms;  
  transition-duration: 260ms;  
  transition-timing-function: ease-in-out;  
  background-color:var(--colors-grey-100);  
  position:fixed;  
  bottom:0;  
  left:96px;  
  z-index:8;  
  padding:16px 8px;  
  display:flex;  
  flex-direction: column;  
  align-items: center;  
  border-radius: 10px;  
  visibility:hidden;  
}  
  
.globalnav-floating-options.is-active {  
  margin-bottom:16px;  
  margin-left:16px;  
  transform:scale(1);  
  opacity:1;  
  visibility:visible;  
}  
  
.globalnav-floating-option:not(:first-of-type) {  
  margin-top:16px;  
}  
  
.globalnav-floating-options-backdrop {  
  z-index:7;  
  position:fixed;  
  width:calc(100vw - 96px);  
  height:100vh;  
  background-color: #000;
```

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

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

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

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

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

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

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

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

/static/css/auth.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,right,opacity;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The image contains two screenshots of the Kraken website. The top screenshot shows the 'Kraken - Login' page. It features a sidebar with a logo and a hamburger menu icon. The main area has a large title 'Kraken - Login' and two buttons: 'Login' (underlined in purple) and 'Signup'. Below these are input fields for 'Username' and 'Password', a 'Remember Me' checkbox, and a 'SUBMIT' button. The bottom screenshot shows the 'Kraken - Signup' page, which has a similar layout with a sidebar, a large title 'Kraken - Signup', and two buttons: 'Login' (underlined in purple) and 'Signup' (underlined in pink). It includes input fields for 'Name', 'Email', 'Username', 'Password', and 'Repeat Password', along with a 'SUBMIT' button.

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.

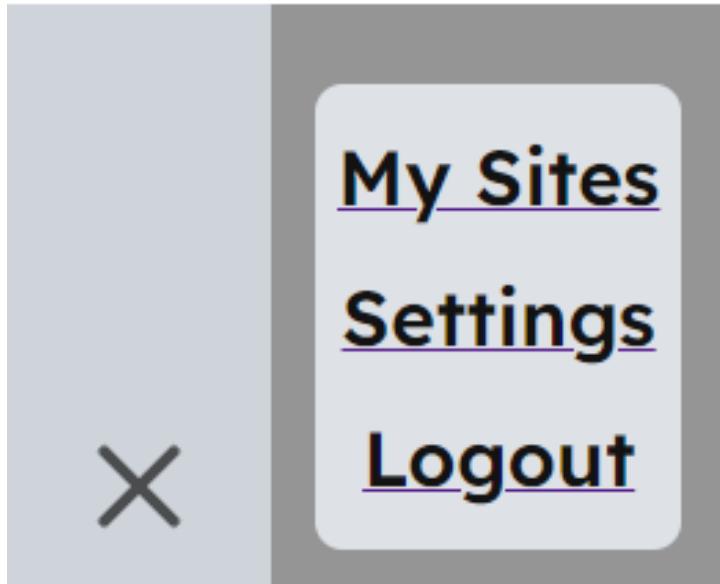
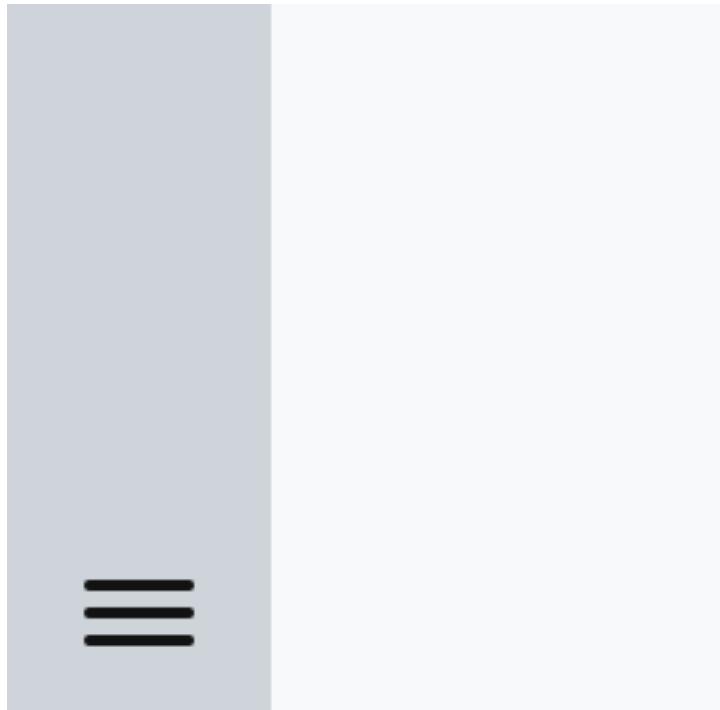
The next step was to set up the frontend programming for the login and signup pages, and the navigation bar. Starting with the navigation bar, I needed to set event listeners for the hamburger and background darkener `div`. These listeners would add or remove the `is-active` class to the hamburger, options 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 doesn't 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:

The image contains two side-by-side screenshots of a login interface. Both screenshots show three fields: 'Username' (containing '6e'), 'Password' (containing 'password'), and 'Remember Me' (with a checked checkbox). To the right of each field is a small green square icon with a red eye symbol, indicating that the input has been checked for validity.

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

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

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

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

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

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

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

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

changes to /static/js/auth.js

```
function verifyField(...)

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

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

changes to `__init__.py`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# TODO: check whether this email already has an account

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

```

# TODO: check whether this username already exists

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

# TODO: create a new user in the database

return redirect(url_for("auth_login"))

```

```

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

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

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

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

```

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

```

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

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

```

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

changes to `__init__.py`

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

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

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

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

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

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

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

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

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

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

changes to /templates/login.html

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

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

changes to /templates/signup.html

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

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

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

changes to `/templates/base.html`

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

```

```

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

```
{% if navbarOptionsEnabled %}  
  
    <!-- Floating option modal for the navbar, which is opened and closed via the  
    hamburger in the navigation bar -->  
  
    <div class="globalnav-floating-options">  
        <a class="globalnav-floating-option one" href="">  
            <span class="globalnav-floating-option-content text header small dark">  
                My Sites  
            </span>  
        </a>
```

```

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

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

<!-- Backdrop behind nav bar modal to apply a darkness filter behind the modal --&gt;

&lt;div class="globalnav-floating-options-backdrop"&gt;&lt;/div&gt;

&lt;script src="{{url_for('static', filename='js/globalnav-floating-options.js')}}"&gt;
&lt;/script&gt;

{% endif %}
</pre>

```

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

```

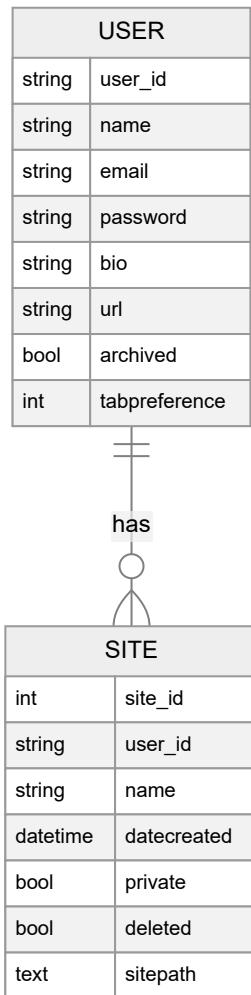
{% set navbarLogoColor = "secondary" %}
{% set navbarOptionsEnabled = False %}

```

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

Stage 2 - Creating and implementing the database

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



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

changes to __init__.py

```
from flask_sqlalchemy import SQLAlchemy

# SQLAlchemy manages the SQL database

# Create the database object
databaseObject = SQLAlchemy()

# The main class of the application
class Kraken():

    # Global reference to database object
    global databaseObject

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

        # Initialise the flask application
        self.initFlask()

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

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

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

models.py

```
from flask_login import UserMixin

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

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

```

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

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

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

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

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

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

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

```

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

db.sqlite commands

```
CREATE TABLE user (
    user_id TEXT PRIMARY KEY,
    name TEXT,
    email TEXT NOT NULL,
    password TEXT NOT NULL,
    bio TEXT,
    url TEXT,
    archived BOOLEAN NOT NULL,
    tabpreference INT NOT NULL
)

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

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

changes to `__init__.py`

```
from flask_login import LoginManager, login_user

# LoginManager is the object that manages signed in users
# login_user logs in a give user

from werkzeug.security import generate_password_hash, check_password_hash

# generate_password_hash and check_password_hash are used when generating
# and authenticating users
```

```
def __init__(self,host:str,port:int) -> None:  
  
    # Initialise the login manager  
    self.loginManager=LoginManager()  
    # set which function routes to the login page  
    self.loginManager.login_view="auth_login"  
    self.loginManager.init_app(self.app)  
  
    # Fetches a row from the User table in the database  
    @self.loginManager.user_loader  
    def loadUser(user_id): return self.User.query.get(user_id)
```

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

```
# Fetch the user from the database. if there's no user it returns none  
user = self.User.query.filter_by(user_id=username).first()  
  
if user is None:  
    # Flashes true to signify an error, the error message, the username  
    # given, and the remember flag given  
    flash([True,'Please check your login details and try again.',  
          username,remember])  
    return redirect(url_for('auth_login'))  
  
# TODO: check for correct password  
if not check_password_hash(user.password,password):  
    flash([True,'Please check your login details and try again.',  
          username,remember])  
    return redirect(url_for('auth_login'))  
  
# Log in the user and redirect them to the homepage  
login_user(user,remember=remember)  
return redirect(url_for("main_home"))
```

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

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

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

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

changes to `__init__.py`

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

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

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

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

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

    # Server-side folder generation

    prefix="static/data/userData/"

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

    self.generateFolderStructure(folderStructure)
```

```

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

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

```

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

changes to `__init__.py`

```

def auth_signup_post() -> Response:

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

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

```

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

changes to `__init__.py`

```

from flask_login import LoginManager, login_user, login_required, logout_user

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

```

Stage 3 - Homepage and Settings

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

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

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

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

 would never work.

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

changes to `_init_.py`

```
# Index route, redirects to auth_login, which will redirect to main_home if  
# logged in  
@self.app.route("/")  
def main_index() -> Response: return redirect(url_for("auth_login"))  
  
# Home Page Route  
@self.app.route("/home/")  
@login_required # User must be logged in to access this page  
def main_home() -> str:  
    # check to see if user has any sites  
    if len(self.Site.query.filter_by(user_id=current_user.user_id).all()) > 0:  
        # and False:  
  
            # For each site, flash its userid, name, and privacy flag  
            # This will not yet do anything as there are no sites in the server  
  
            # flash([[x.user_id,x.name,x.private] for x in self.Site.query.filter_by(  
            # user_id=current_user.user_id).all()])  
  
            # For testing and design purposes, the flash command above was commented out  
            # and replaced with this command  
            flash([["user1","Site 1",True],["user1","Epic Webpage",False]])  
  
    return render_template("home-sites.html")  
return render_template("home-nosite.html")
```

/templates/home-nosite.html

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

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

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

{% block content %}

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

<div class="application-content">

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

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

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

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

        </div>
    </div>

</div>

{% endblock %}
```

/templates/home-sites.html

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

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

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

{% block content %}
```

```

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

<div class="application-content">

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

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

        {% for site in sites %}

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

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

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

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

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

        {% endfor %}

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

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

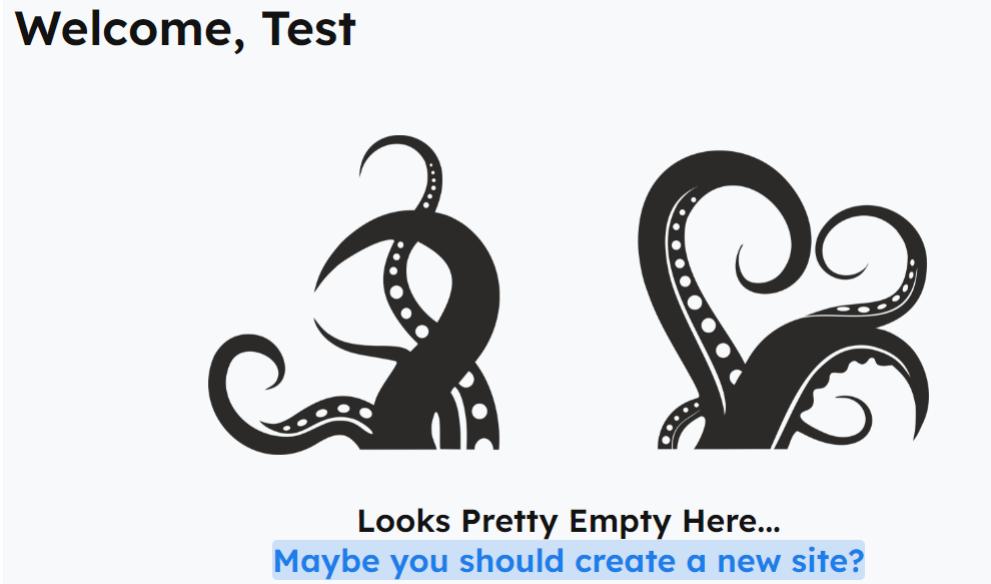
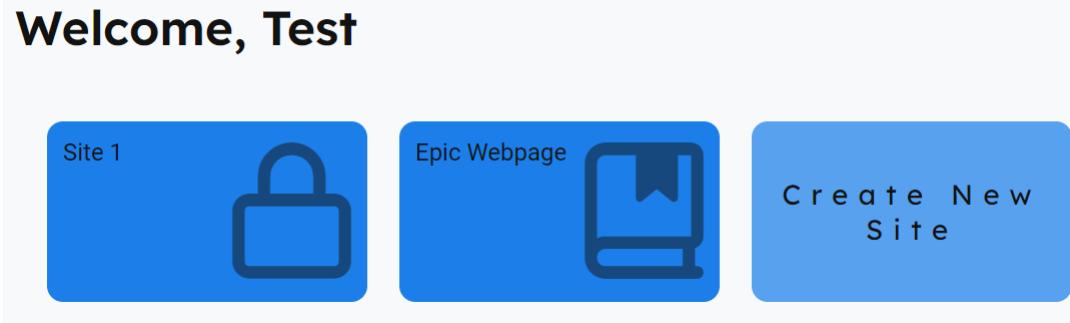
    </div>

</div>

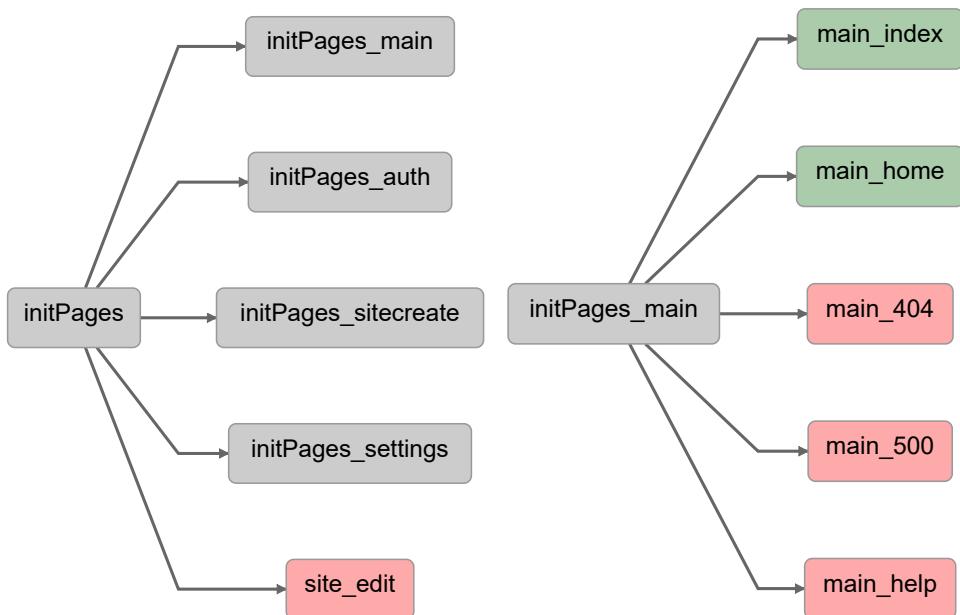
{% endblock %}

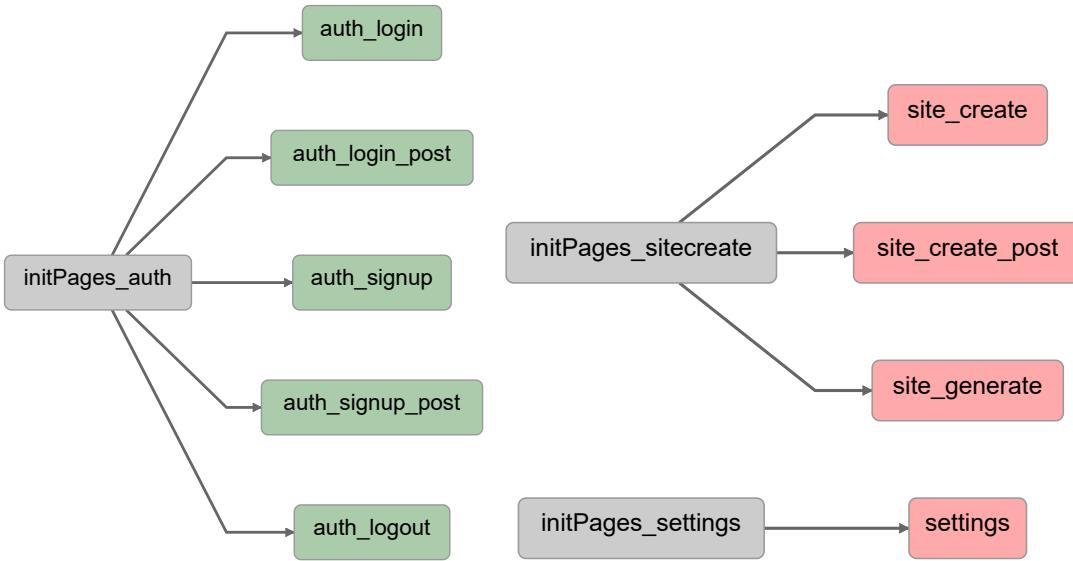
```

The homepage looked like this:



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



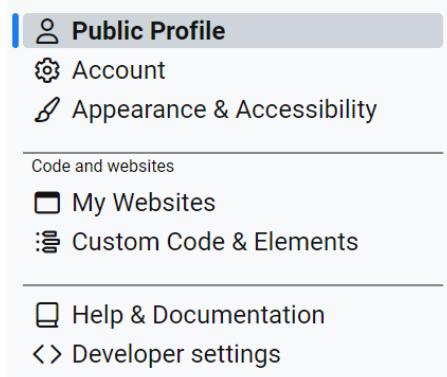


At this point in development, I was able to quickly build the frontend versions of the settings pages for the account system (they are listed in the desirable features section of the success criteria). I didn't add in database functionality to them yet as I knew it would take significantly longer.

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

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

Settings



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

```

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

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

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

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

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

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

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

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

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

</div>

<div class="settings-content">

  {% block settings_content %}
  {% endblock %}

```

```

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

{% endblock %}

```

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

Profile

Name

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

Profile Picture



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

Bio

Url

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

[UPDATE PROFILE](#)

/templates/settings-profile.html

```

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

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

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

{% block settings_content %}

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

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

```

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

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

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

</div>

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

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

</div>

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

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

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

</div>

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

</div>

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

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

```

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

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

</div>

</form>

{% endblock %}

```

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

Account

Change Username

WARNING: Changing your username can create issues.

[DEW IT](#)

Change Email

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

[DEW IT](#)

Export Account Data

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

[EXPORT METADATA](#)

[EXPORT WEBSITES](#)

[DOWNLOAD ALL](#)

Archive Account

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

[ARCHIVE YOUR ACCOUNT](#)

Reset Account

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

[RESET YOUR ACCOUNT](#)

Delete Account

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

[DELETE YOUR ACCOUNT](#)

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

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

{% set settingsSidebarActivated = 2 %}

{% block settings_content %}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

</div>

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

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

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

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

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

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

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

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

```

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

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

</form>

{% endblock %}

```

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

Appearance and Accessibility

Tab Preference

4 (Default) ▾

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

/templates/settings-looks.html

```

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

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

{% set settingsSidebarActivated = 3 %}

{% block settings_content %}

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

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

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

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

```

```

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

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

</form>

{% endblock %}

```

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

My Websites

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

/templates/settings-sites.html

```

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

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

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

{% block settings_content %}
<h3 class="settings-content-header text header dark">My Websites</h3>

```

```

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

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

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

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

            {% for site in flashedSiteNames %}

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

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

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

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

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

                </div>

            {% endfor %}

        </div>
    </div>

</form>

{% endblock %}

```

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

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

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

This will:

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

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

changes to `__init__.py`

```
def getFolderSize(self,path:str) -> int:  
    # Get the size of the base directory, should return 0  
    size=self.os.path.getsize(path)  
  
    # for all directories and files under the path  
    for sub in self.os.listdir(path):  
        subPath=self.os.path.join(path,sub) # get the path of the directory / file  
  
        # get the size if it is a file  
        if self.os.path.isfile(subPath): size+=self.os.path.getsize(subPath)  
  
        # get the size if it is a directory by calling this function  
        elif self.os.path.isdir(subPath): size+=self.getFolderSize(subPath)  
  
    # return the size, in bytes  
    return size
```

```
def convertByteSize(self,bytes:int) -> str:  
    # Zero check  
    if bytes==0: return "0B"  
  
    # All possible sizes  
    sizes=("B","KB","MB","GB","TB","PB","EB","ZB","YB")
```

```

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

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

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

```

```

def initPages_settings(self) -> None:

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

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

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

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

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

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

```

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

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

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

Stage 4 - Creating a New Site

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

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

/templates/site-create-base.html

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

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

{% block content %}

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

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

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

{% endblock %}
```

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

Create a new site

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

Owner Website Name *

@test /

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

Description (Optional)

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

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

/templates/site-create.html

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

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

{% block site_create_base %}

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

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

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

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

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

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

    </div>
</form>
```

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

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

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

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

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

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

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

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

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

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

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

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

```


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


<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 programmatically enabling and disabling the submit button

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

changes to `__init__.py`

```

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

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

```

```

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

flash(out[:-1])

return render_template("site-create.html")

```

changes to /templates/site-create.html

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

/static/site-create.js

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

```

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

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

    formSubmit.removeAttribute("disabled")
    return
}

```

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

```

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

    hideFormMessage()

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

```

```

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

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

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

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

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

hideFormMessage()
return "success"
}

```

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

```

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

```

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

changes to `_init_.py`

```

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

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

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

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

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

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

```

```

    return listToStr(var)

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

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

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

return redirect(url_for("site_create_options_1"))

```

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

changes to `__init__.py`

```

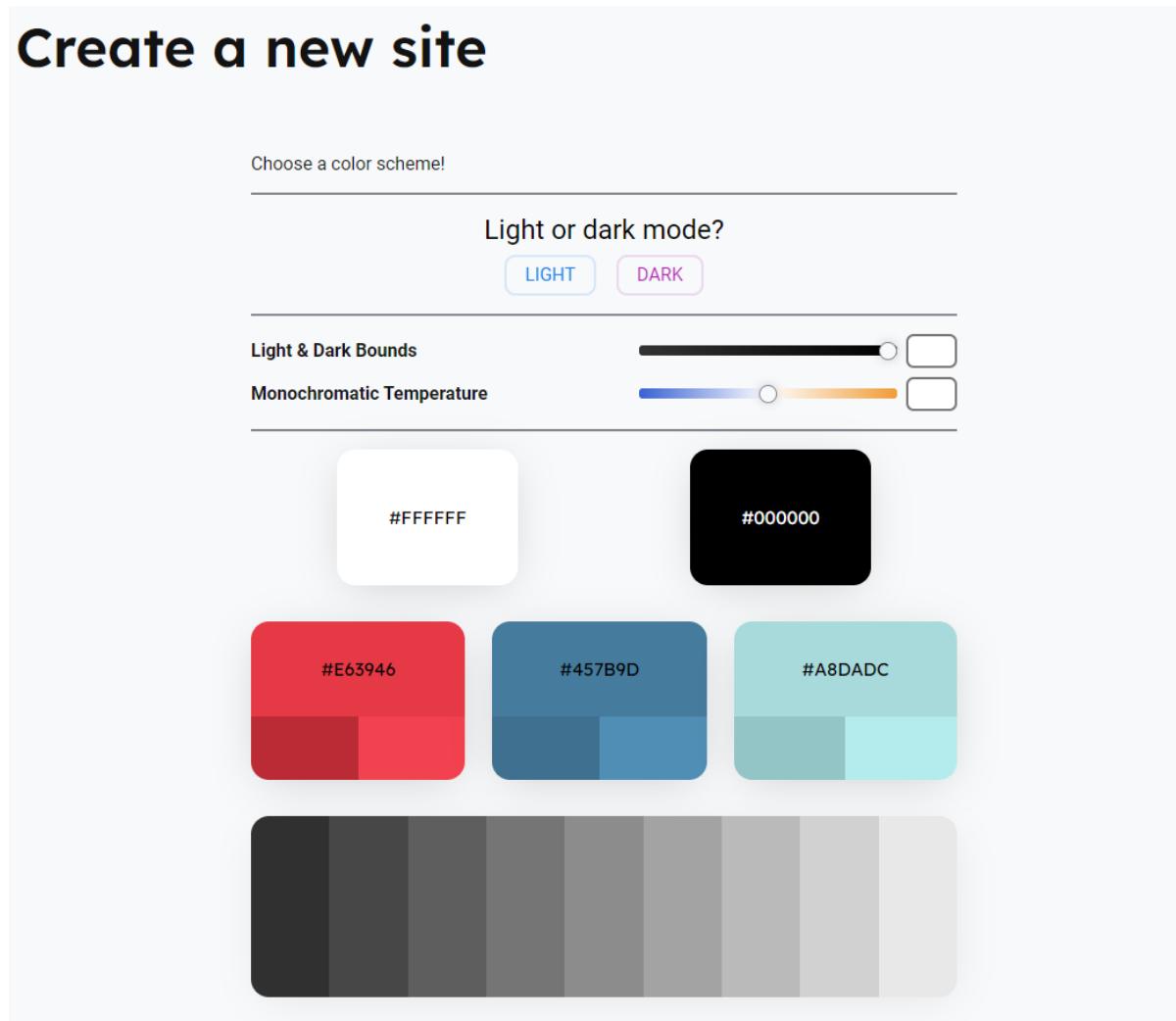
@self.app.route("/home/new/1")
@login_required
def site_create_options_1() -> str | Response:
    # stop people from starting halfway through the form i.e. if they didnt come
    # from the previous site_create page, send them to the start
    if not request.referrer == url_for("site_create",_external=True):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

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

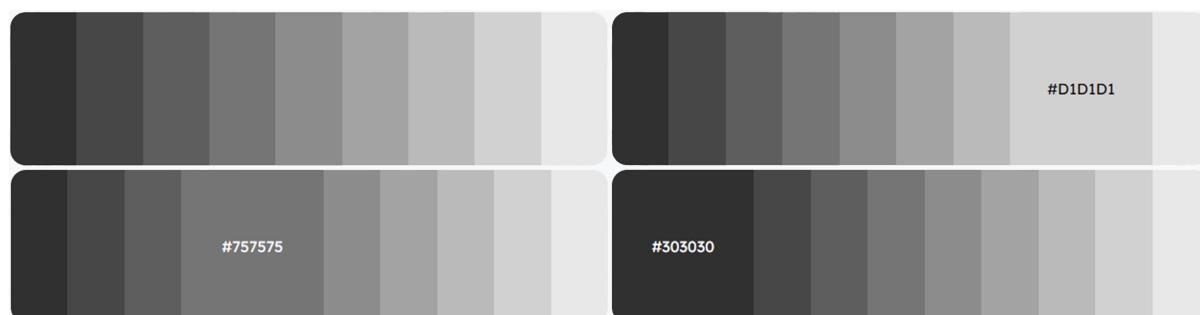
```

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

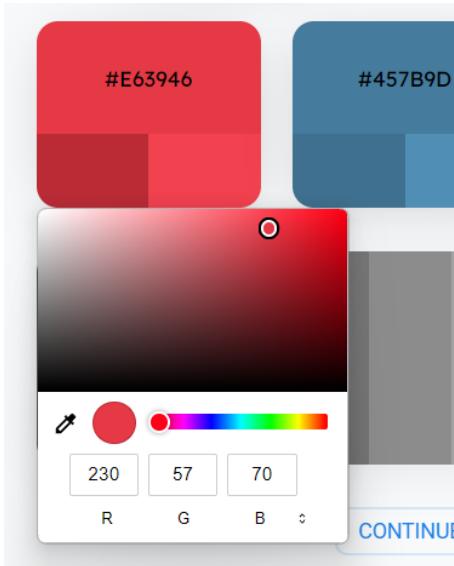
Create a new site



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 %}



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

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

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

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

            <div class="btn secondary thin rounded slide from-right m-xs-t"
                id="new_site_darkModeToggle">
                <span class="btn-content text uppercase secondary notextselect">
                    Dark
                </span>
            </div>
        </div>
    </div>
</div>

</form>


```

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

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

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

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

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

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

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

        </div>
    </div>

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

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

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

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

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

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

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

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

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

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

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

</div>

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

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

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

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

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

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

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

  <div class="color-triple-card accent-color">
    <input type="color" class="color-card-picker-input"
      id="new_site_colors_accent_picker" value="#a8dadcc">

    <div class="color-triple-card-main">
      <span class="color-code text uppercase center">#a8dadcc</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": "#a8dadcc", "accent-dark": "", "accent-light": "",
    "grey-100": "#303030", ... , "grey-900": "#e8e8e8"
}
var colors = defaultColors

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

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

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

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

    updateStored()
}

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

    updateStored()
}

```

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

```

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

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

```

```

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

updateStored()
}

```

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



```

function updateColorVariables() {
  var changePercent = 20

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

    var newColor = hexToRgb(colors[color])

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

    newColor = 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 interprets the given input into a dictionary, and stores it in the `flask.session` object, meaning it can be accessed at a later date.

changes to `__init__.py`

```

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

```

```

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

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

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

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

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

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

```

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

Create a new site

The screenshot shows a web-based font selection interface. At the top, a header reads "Choose font family - individual elements can be customised". Below this, a legend box contains the text "Lexend" and "Paragraph text - Roboto". The main area lists several typefaces with their corresponding paragraph text examples and font families:

- Prata**
Paragraph text - Lato
- DM Sans**
Paragraph text - Catamaran
- Titillium Web**
Paragraph text - Raleway
- Caudex**
Paragraph text - PT Mono
- Noto Serif Display**
Paragraph text - Lora
- STAATLICHES**
Paragraph text - Syne Mono

A blue "CONTINUE" button is located at the bottom left of the interface.

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

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

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

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

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

] %}

{% block site_create_base %}

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

</div>

{endfor%}
</div>

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

</form>

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

{% endblock %}

```

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

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

```

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

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

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

```

```

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

})
}

```

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

changes to `_init_.py`

```

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

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

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

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

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

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

```

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

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

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

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

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

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

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

changes to __init__.py

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

    cfgContent.read(siteConfigFile)

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

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

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

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

```

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

# Create a new Site object using the variables given
newSite=self.Site(
    name=siteSettings["name"],
    datecreated=siteSettings["created"],
    private=not siteSettings["isPublic"],
    deleted=False,
    user_id=siteSettings["user"],
    sitePath=sitePath,
)
# Add and commit the new site to the database
self.db.session.add(newSite)
self.db.session.commit()
```

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

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

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

Stage 5 - Code Documentation

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

Sphinx Docstring Format

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

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

[Summary]

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

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

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

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

```
class Kraken():

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

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

    :returns: Void
    :rtype: None
    """

```

```
class Kraken(
    host: str,
    port: int
)
```

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

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

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

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

    :returns: Void
    :rtype: None
    """

```

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

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

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

```
def main_home() -> str:  
    """Page for ``/home``.  
  
    This page is the homepage, containing links to all of the current user's sites.  
  
    Requires a user to be logged in.  
  
    Flashes a list of the users sites, in the format  
    ``[<userid>,<sitename>,<isprivate>]`` , if the user has any sites.  
  
    :returns: The HTML content of the page, generated from the Jinja template syntax  
    :rtype: str  
    """
```

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

```
Page for /home/ .
```

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

```
Requires a user to be logged in.
```

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

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

```
:rtype: str
```

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

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

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

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

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

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

```
Page for /login/ .
```

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

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

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

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

```
:rtype: str | Response
```

```
def auth_login_post() -> Response:  
    """POST method of ``/login``.  
  
    Fetches the given user information from the form via ``flask.request``, and  
    validates it. The user is logged in if the data is valid,  
    and shown an error message if it is not.  
  
    Flashes an appropriate error message, if required.  
  
    :returns: A redirect to the ``main_home`` or ``auth_login`` depending on the  
             success of the login attempt  
    :rtype: Response  
    """
```

```
(function) def auth_login_post() -> Response  
POST method of /login/ .  
Fetches the given user information from the form via flask.request , and  
validates it. The user is logged in if the data is valid, and shown an error  
message if it is not.  
Flashes an appropriate error message, if required.  
.returns: A redirect to the main_home or auth_login depending on the  
success of the login attempt  
.rtype: Response
```

```
def site_create_generate() -> Response:  
    """Method for ``/home/new/generate``.  
  
    This page is the second third creation page. It contains the button styling  
    selection system.  
    Requires a user to be logged in.  
    Requires the referrer to be ``site_create_options_2``  
  
    :returns: A redirect to the new site homepage (``site_edit_home``) if  
             successful OR a redirect to ``site_create`` if  
             ``site_create_options_2`` was not the referrer.  
    :rtype: Response  
    """
```

```
(function) def site_create_generate() -> Response  
Method for /home/new/generate .  
This page is the second third creation page. It contains the button styling  
selection system.  
Requires a user to be logged in. Requires the referrer to be  
 site_create_options_3  
.returns: A redirect to the new site homepage ( site_edit_home ) if successful  
OR a redirect to site_create if site_create_options_3 was not the  
referrer.  
.rtype: Response
```

```

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

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

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

```

```

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

```

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

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

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

:type canHaveSpecialChar: bool, optional
:returns: An empty string if the field is valid OR an error message if the field is
invalid
:rtype: str

```
def createSiteStructure(self,siteSettings:dict[str,str | any]) -> None:  
    """Creates a new site in the database and in local storage.  
  
    :param siteSettings: a dictionary containing the information about the site,  
        with the keys: ``name`` ``user`` ``desc`` ``created``  
        ``isPublic`` ``colorOptions`` ``fontOptions``  
        ``buttonOptions``  
    :type siteSettings: dict  
  
    :returns: Void  
    :rtype: None  
    """
```

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

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

:param siteSettings: a dictionary containing the information about the site, with
the keys: name user desc created isPublic colorOptions
fontOptions buttonOptions
:type siteSettings: dict
:returns: Void
:rtype: None

```
def convertByteSize(self,bytes:int) -> str:  
    """Converts a given byte value into a human readable version  
  
    :param bytes: The amount of bytes to convert  
    :type bytes: int  
  
    :returns: A human readable version of the value  
    :rtype: str  
    """
```

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

Converts a given byte value into a human readable version

:param bytes: The amount of bytes to convert
:type bytes: int
:returns: A human readable version of the value
:rtype: str

Variable Type Declarations

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

Variable type declaration in Python is done with the syntax:

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

This can also be done in functions:

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

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

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

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

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

Stage 6 - Drag-and-Drop Editor

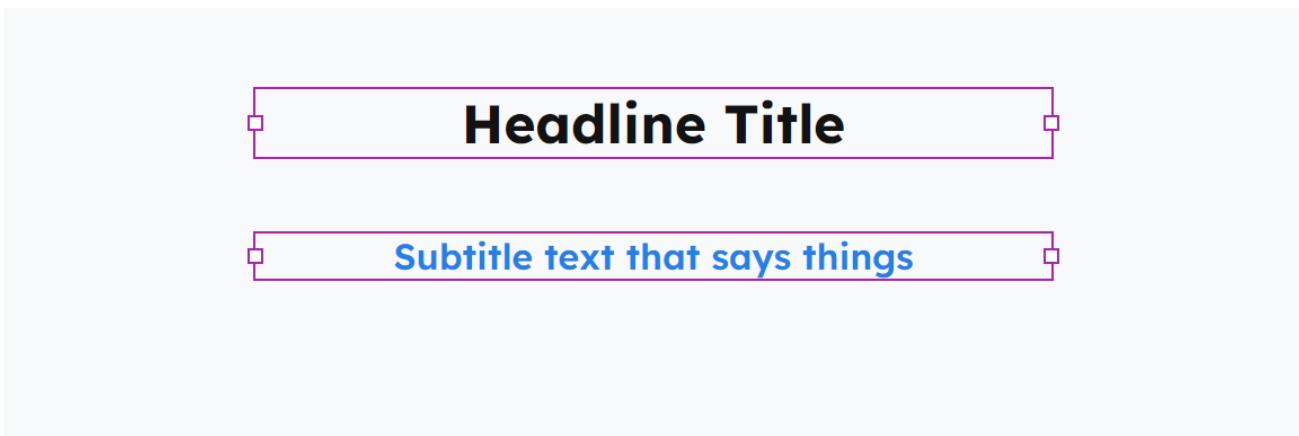
HTML Syntax

The first part of creating the drag-and-drop editor was creating the syntax used for the sections. I created a basic headline section to use as a template when testing. It includes the `section-wrapper`, which has the relevant data tags. The `section` element inside contains all of the required css variables for the section parameters, and then the `grid-item`s with their own individual parameters.

The `position` parameter is used in the grid positioning system, with the syntax `<start-row>/<start-column>/<end-row>/<end-column>`. This is what will be changed when the element is being moved or resized by JavaScript.

Each `grid-item` has an `element-resize-container`, which contains the `resize-corner`s for resizing, and the `element-outline` to show that the element is selected. These will not be included in the final product, and instead will be added by the JavaScript so that it doesn't appear in the final site. They are included at the moment for design purposes.

This is what the example section looked like when I opened the HTML file on its own:



`/static/data/html/headline/html_section_headline_1.html`

```
<div class="section-wrapper" data-kraken-section>

<section class="--headline --type-1"
        style="--content-width: 1600px;
              --grid-width: 1224px;
              --column-count: 12;
              --content-padding: 16px;
              --cols: 12;
              --rows: 6;
              --width: 1224px;
              --m-rows: 1;
              --col-gap: 24px;
              --row-gap: 16px;
              --row-size: 48px;>
```

```
--column-gap: 24px;
--block-padding-top: 16px;

--block-padding: 16px 0 16px 0;
--block-padding-right: 0;
--block-padding-bottom: 16px;
--block-padding-left: 0;">
<div class="section-background"></div>

<div class="section-grid">

<div class="grid-item" data-kraken-element data-kraken-resizable
data-kraken-draggable data-kraken-editable-style data-kraken-editable-text
style="--text: center;
--align: flex-start;
--justify: center;
--position: 2/5/2/13;
--element-z-index: 1;
--element-height: 0px;">

    <div class="text-box grid-item-content">
        <h1 style class="text-content text dark bold large header center">
            Headline Title
        </h1>
    </div>

    <div class="element-resize-container">

        <div class="resize-corners">
            <div class="resize-corner resize-corner-horizontal-left">
                <div class="resize-corner-visual-element"></div>
            </div>
            <div class="resize-corner resize-corner-horizontal-right">
                <div class="resize-corner-visual-element"></div>
            </div>
        </div>
        <div class="element-outline" style="height: 40px;"></div>
    </div>
</div>

<div class="grid-item" data-kraken-element data-kraken-resizable
data-kraken-draggable data-kraken-editable-style data-kraken-editable-text
style="--text: center;
--align: flex-start;
--justify: center;
--position: 4/5/4/13;
--element-z-index: 1;
--element-height: 0px;">

    <div class="text-box grid-item-content">
        <h2 style class="text-content text primary header">
            Subtitle text that says things
        </h2>
    </div>
</div>
```

```

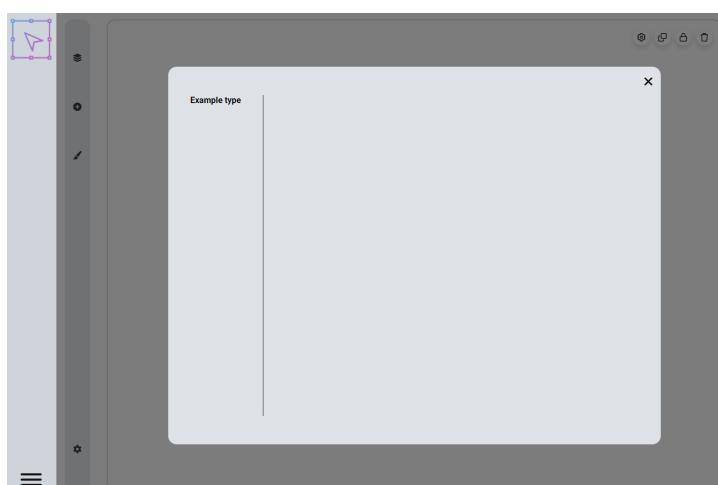
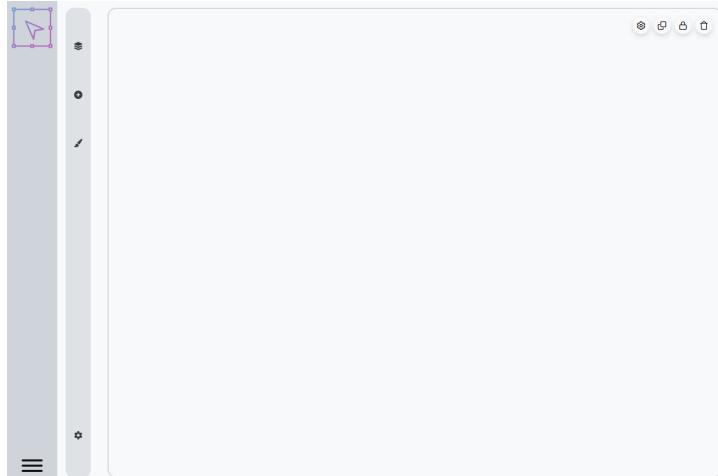
</div>

<div class="element-resize-container">
  <div class="resize-corners">
    <div class="resize-corner resize-corner-horizontal-left">
      <div class="resize-corner-visual-element"></div>
    </div>
    <div class="resize-corner resize-corner-horizontal-right">
      <div class="resize-corner-visual-element"></div>
    </div>
  </div>
  <div class="element-outline" style="height: 40px;"></div>
</div>

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

```

I next created the HTML for the editor. This includes a local navigation bar containing links for the website page organisation modal, section selection modal, style editor, and settings page, which, when hovered, will display a label showing what they do. I also created the website styles modal with the associated JavaScript to open it. It also includes the section options buttons, which will be rendered when a section is selected.



/templates/site-edit.html

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

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

{% set settingsSidebarActivated = get_flashed_messages()[0] %}
{% set currentName = get_flashed_messages()[1] %}
{% set currentSite = get_flashed_messages()[2] %}

{% block content %}

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

<div class="lightbox-mask"></div>

<div class="application-content">

<div class="localnav one localnav-vertical localnav-collapsible">
    <div class="localnav-content">

        <ul class="localnav-list">

            <ul class="localnav-list" style="height:30%">

                <li class="localnav-item one">
                    <a class="link unformatted">
                        <div class="localnav-item-collapsible-icon">
                            <i class="faicon fa-solid fa-layer-group"></i>
                        </div>
                        <div class="localnav-item-collapsible-text text bold primary">
                            Website Pages
                        </div>
                    </a>
                </li>

                <li class="localnav-item two">
                    <a class="link unformatted" id="localnav_add_section_btn">
                        <div class="localnav-item-collapsible-icon">
                            <i class="faicon fa-solid fa-circle-plus"></i>
                        </div>
                        <div class="localnav-item-collapsible-text text bold primary">
                            Add Section
                        </div>
                    </a>
                </li>

                <li class="localnav-item three">
                    <a class="link unformatted">
                        <div class="localnav-item-collapsible-icon">
```

```
        <i class="faicon fa-solid fa-paintbrush"></i>
    </div>
    <div class="localnav-item-collapsible-text text bold primary">
        Website Styles
    </div>
    </a>
</li>

</ul>

<li class="localnav-item four">
    <a class="link unformatted">
        <div class="localnav-item-collapsible-icon">
            <i class="faicon fa-solid fa-gear"></i>
        </div>
        <div class="localnav-item-collapsible-text text bold primary">
            Website Settings
        </div>
    </a>
</li>

</ul>
</div>

</div>

<div class="section-selector-container">
    <div class="section-selector">
        <div class="section-selector-exit-btn">
            <i class="faicon fa-solid fa-xmark"></i>
        </div>

        <div class="section-selector-sidebar">
            <div class="section-selector-nav" id="section_selector_nav">
                <div class="section-selector-nav-content">
                    <ul class="section-selector-nav-list">
                        <ul class="section-selector-nav-list two" style="height:30%">
                            <!-- Template header - will be replaced by JS -->
                            <li class="section-selector-item one">
                                <a class="link unformatted" id="section_selector_nav_headline" style="opacity:1">
                                    <span class="text bold">Example type</span>
                                </a>
                            </li>
                        </ul>
                    </ul>
                </div>
            </div>
        </div>
    </div>
</div>
```

```

<div class="vertical-separator" style="height:85%"></div>

<div class="section-selector-content">
    <div class="section-selector-list" id="section_selector_list">
        <style id="section_selector_style"></style>
    </div>
</div>
</div>

<div class="site-builder">

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

    <div class="site-builder-preview" id="contains_site">
    </div>

    <!-- will be set to hidden by js code -->
    <div class="section-edit-actions shown">
        <div class="actions-list">
            <div class="edit-action" id="section_edit_action_settings">
                <i class="fa-regular fa-gear"></i>
            </div>

            <div class="edit-action" id="section_edit_action_duplicate">
                <i class="fa-regular fa-clone"></i>
            </div>

            <div class="edit-action" id="section_edit_action_lock">
                <i class="fa-regular fa-lock"></i>
            </div>

            <div class="edit-action hilight-danger" id="section_edit_action_delete">
                <i class="fa-regular fa-trash"></i>
            </div>
        </div>
    </div>

    </div>
</div>

{% endblock %}

```

The JavaScript for the add section modal included adding event listeners to the local navigation bar button, the background of the modal that darkens the rest of the page, and the "X" in the top right of the modal.

/static/js/site-edit.js

```
var addSectionBtn = document.getElementById("localnav_add_section_btn")
var addSectionModal_container = document.querySelector(
  ".application-content .section-selector-container")

function addSectionModal_show() {
  addSectionModal_container.classList.add("shown");
  document.querySelector(".lightbox-mask").classList.add("shown")
}

function addSectionModal_hide() {
  document.querySelector(".application-content .section-selector-container")
    .classList.remove("shown")
  document.querySelector(".lightbox-mask").classList.remove("shown")
}

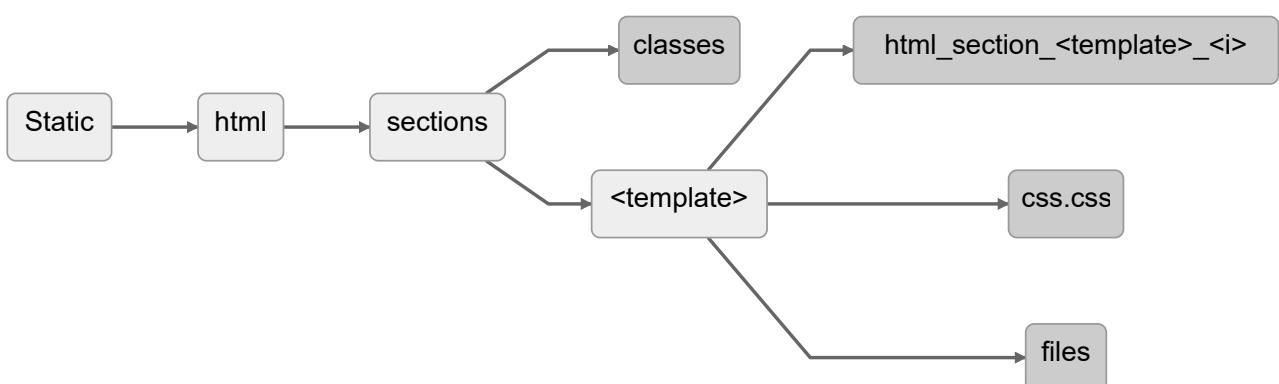
// event listener for the add section button
addSectionBtn.addEventListener("click",() => {
  // if the add section modal is not already shown
  if (!(addSectionModal_container.classList.contains("shown"))){
    addSectionModal_show()
  }
})

// event listener for the close x in the add section modal
document.querySelector(".application-content .section-selector-exit-btn")
  .addEventListener("click",addSectionModal_hide)

// event listener for the dark background of the modals
document.querySelector(".lightbox-mask")
  .addEventListener("click",addSectionModal_hide)
```

Importing Sections

The next step was to be able to import the template sections into the site so that they can be shown in the add section modal. The format of the directories for the template directories looks like this (where file nodes are a darker shade):



The `classes` file will contain all of the section type names (in effect, every folder in the directory).

/static/html/classes example

```
headline
about
gallery
header
footer
image
```

The `files` file will contain all of the section templates in the given directory.

/static/html/<template>/files example

```
html_section_headline_1.html
html_section_headline_2.html
html_section_headline_3.html
html_section_headline_4.html
html_section_headline_5.html
html_section_headline_6.html
html_section_headline_7.html
html_section_headline_8.html
html_section_headline_9.html
html_section_headline_10.html
html_section_headline_11.html
html_section_headline_12.html
```

The `css.css` file will contain any and all CSS code that the templates contained in the directory will require.

Given that the syntax for the `/static/html/` folder is now down, I can write the code that imports the templates to be previewed. This involved using the `fetch()` function in JavaScript, which is a function that needs to be `await`ed to get the required response. `async` and `await` are utilities that I have used before, but not very much. Essentially, calling an `async` function without `await`ing it will return a `Promise` object that is updated when the function is finished. If, instead, you call the function and `await` it, it will hold the program on that line until its finished, and it will return the expected value. However, JavaScript doesn't allow you to `await` functions at the top-level, so I had to create an `async` function (in this case, `loadsections()`) that used `await`, and called said function at top-level without `await`ing it.

changes to /static/js/site-edit.js

The `loadsections` function goes through every template and adds it to the `addSectionModal_data` dictionary to make sure that all of the files are loaded. This is done when the webpage has finished loading to ensure that the user can access all of the section templates immediately, without having to wait for them to load. Due to the nature of this being an `async` function that isn't `await`ed, it also means that this can happen in "parallel" to the rest of the JavaScript, meaning that it can load the files in the background without affecting the usability of the interface.

```
// loads all section templates from the /static/html/ directory and stores them
// to the dictionary addSectionModal_data, along with the types of templates
// being stored in the list addSectionModal_headers
// The function is asynced to allow the use of await functions to fetch the
// content of the file
async function loadsections() {

    // load the classes file, which contains all of the template types, and store
    // them in the headers list
    await fetch("../..../static/html/sections/classes")
        .then( response => { // throw an error if there is an issue with the response
            if (!response.ok) throw new Error(`HTTP error: ${response.status}`)
            return response.text()
        })
        .then( text => {
            // add each line to the DOM and headers list
            for (line of text.split("\n")) {
                addSectionModal_setNavbar(line)
                addSectionModal_headers.push(line)
            }
        })

    var path
    var files

    // iterate through all the section types
    for (var header of addSectionModal_headers) {

        // add the section type to the data
        addSectionModal_data[header] = {css:"",sections:[]}

        // set the root URL for the section type
        path=`..../..../static/html/sections/${header}`

        // fetch the custom CSS file for this type of headers
        await fetch(path+"/css.css")
            // throw an error if there is an issue with the response
            .then( response => {
                if (!response.ok) throw new Error(`HTTP error: ${response.status}`)
                return response.text()
            })
    }
}
```

```

// add to the data dictionary
.then( text => addSectionModal_data[header].css=text )

// fetch the file list
await fetch(path+"/files")
// throw an error if there is an issue with the response
.then( response => {
  if (!response.ok) throw new Error(`HTTP error: ${response.status}`)
  return response.text()
})
.then( text => {
  // split the file into lines and filter out any empty lines
  files=text.split(/[\r\n]+/g).filter(
    function(value, index, arr) { return value != "" }
  )
})
}

// iterate through all given files
for (var file of files) {

  // fetch the template HTML
  await fetch(path+"/"+file)
  // throw an error if there is an issue with the response
  .then( response => {
    if (!response.ok) throw new Error(`HTTP error: ${response.status}`)
    return response.text()
  })
  // add the template to the sections list
  .then( text => addSectionModal_data[header].sections.push(text))

}
}
}

```

The `addSectionModal_setNavbar` function is called by the `loadsections` function to append a given title to the local navigation bar of the modal. It uses the `addSectionModal_navItem` variable, which is essentially a template string defined later in the project. The string includes `[i]`, `[n1]`, and `[n2]`, which indicate areas that will be replaced in this function, before appending it to the `section-selector-nav-list` DOM element.

```

function addSectionModal_setNavbar(text) {
  // format the string based on the content of the string
  out=addSectionModal_navItem
  .replace("[i]",number2words(1)) // TODO work out the correct number
  .replace("[n1]",text.replace(" ",""))
  .replace("[n2]",capitalizeWords(text))

```

```
// append to the local navbar
addSectionModal_nav.querySelector("ul.section-selector-nav-list").innerHTML=out
}
```

The `addSectionModal_clearCurrentPreviews` function is called whenever a new navigation item is selected - i.e. a different set of templates needs to be displayed. It iterates through all currently visible templates (fetched by the `addSectionModal_getAllTemplates` function defined later) and removes them from the DOM.

```
function addSectionModal_clearCurrentPreviews() {
  addSectionModal_getAllTemplates().forEach((e) => {
    // remove the section if it is not the style element
    if (!(e == addSectionModal_styleElement)) e.remove()
  })
}
```

The `addSectionModal_populate` function is also called when a new navigation item is selected. It calls the `addSectionModal_clearCurrentPreviews` to clear the current sections, iterates through all of the sections in the passed `data` argument, and adds each one to the section. It will also add the event listeners to add the section to the content once the functionality has been built. I had to experiment with different ways of adding the section to the modal, as `innerHTML += section` wasn't able to do it correctly in this instance.

```
function addSectionModal_populate(data) {
  addSectionModal_styleElement.innerHTML = data.css

  addSectionModal_clearCurrentPreviews()

  var i=1

  for (var section of data.sections) {
    //addSectionModal_list.innerHTML += section
    addSectionModal_list.insertAdjacentHTML("beforeend",section)

    var e = document.querySelector(".--headline.--type-"+i)
    e.addEventListener("click",function() {}) // TODO add section to content

    i++
  }
}
```

The top-level of `site-edit.js` defines all of the variables used for the modal, which can be easily identified by the `addSectionModal_` prefix. It also includes simple functions such as `addSectionModal_getAllTemplates`.

```

var addSectionModal_selectedNavItem="one"
var addSectionModal_selectedNavItemInt=1
var addSectionModal_container=document.querySelector(
  ".application-content .section-selector-container")
var addSectionModal_list=document.getElementById("section_selector_list")
var addSectionModal_styleElement=document.getElementById("section_selector_style")
var addSectionModal_nav=document.getElementById("section_selector_nav")
var addSectionModal_data = {}
var addSectionModal_headers = []

var addSectionModalNavItem=<li class="section-selector-nav-item [i]">
<a class="link unformatted" id="section_selector_nav_[n1]">
<span class="text bold">[n2]</span></a></li>

var addSectionModal_getAllTemplates = function() {
  return addSectionModal_list.childNodes }

loadsections()

```

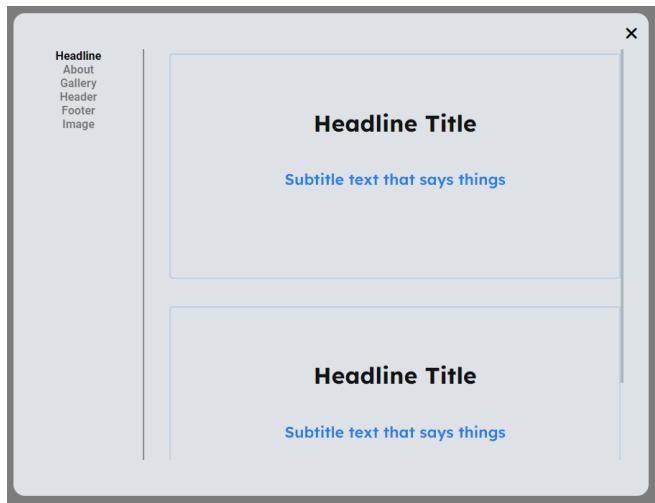
Having tested the functionality to ensure that it successfully imports all of the required sections, this is what the modal looks like when it is asked to import, using the test data outlined earlier.



The important thing that I had forgotten was to consider that the sections would not be scaled down from their original sizes, so, with use of the `data-preview` tag, I created the `preview_override.css` file and went through every style that could affect an element in the section from both `main.css` and `edit.css`, and copied them over. I then wrapped any variables in `calc(<value> * 2 / 3)` or simply put in the calculated value for explicit values. This was only necessary for measurements using the `px` units. The full contents of `preview_override.css` can be found in Appendix A.

To make a clearer distinction between sections, I also added a light border (that gets more noticeable when hovered) to each section.

After doing that, the preview now looked much better.



Adding Sections to the Editor

The next step was to be able to add a selected section to the editor. When the modal is being populated with elements, they are each given click event listeners. These will use the functions `stripPreviewTags` and `addSection`, along with the `addSectionModal_hide` function to close the modal.

After completing this, I added the event listener functionality to sections and elements in the editor.

changes to `/static/js/site-edit.js`

```
function addSectionModal_populate(data)

e.parentElement.addEventListener("click", (e) => {
  addSection(stripPreviewTags(e.currentTarget.outerHTML))
  addSectionModal_hide()
})
```

The `stripPreviewTags` function removes all `data-preview` tags from the element so that it renders normally in the editor. To do this, the event listener gets the `currentTarget` (the element that was assigned the event listener) and fetches its `outerHTML`, which returns a string containing the HTML the element and its contents.

```
function stripPreviewTags(html) {
  return html.replaceAll("data-preview", " ")
}
```

The `addSection` function appends the section to the end of the editor, and then calls `sectionEventListeners` and `elementEventListeners`, which will be used to add the required event listeners for the drag-and-drop system to work, along with `createResizeBoxes` to draw in the resize boxes to the elements.

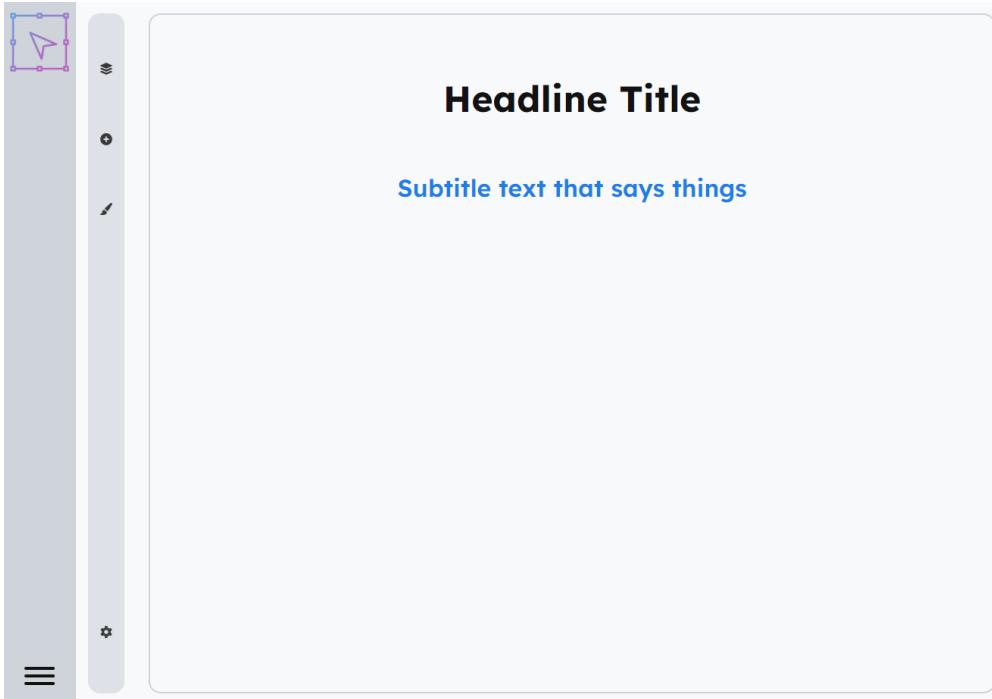
```

function addSection(html) {
    try {siteroot().insertAdjacentHTML("beforeend",html)}
    catch {siteroot().innerHTML += html}

    // sectionEventListeners()
    // elementEventListeners()

    // createResizeBoxes()
}

```



The `sectionEventListeners()` function goes through each section and adds an `onclick` event, which will clear all selected sections and set the current one as selected. It will also display the section action buttons, once they have been implemented.

```

function sectionEventListeners() {
    clearSelectedSections()

    for (var section of sections()) {
        section.addEventListener("click",((e)=>{
            var x = e.currentTarget.parentElement
            while (!(x.hasAttribute("data-kraken-section")))
                x = x.parentElement

            clearSelectedSections()
            x.setAttribute("data-kraken-section-selected","",)
            selectedSection=x

            // TODO show the section action buttons
        }))
    }
}

```

```
    }
}
```

The `clearSelectedSections` function goes through each section and removes the `data-kraken-section-selected` attribute from them. It will also hide the section action buttons once they have been implemented.

```
function clearSelectedSections() {
  for (var section of sections())
    section.removeAttribute("data-kraken-section-selected")

  // TODO hide the section action buttons
}
```

To allow for easier selection of elements, I created the `siteroot`, `elements`, and `sections` functions, that all return `NodeList`s or DOM elements, so that all of the query definition for elements is in the same place.

```
function siteroot() { return document.querySelector("[data-content-parentview])}

function sections() {
  return document.querySelectorAll(`[data-kraken-section]:not([data-preview])`)
}

function elements() {
  return document.querySelectorAll(`[data-kraken-section]:not([data-preview])
  [data-kraken-element]`)
}
```

When a section in the editor is hovered, it will display a light border. When it is then selected, it will display a more prominent border to indicate that it has been selected. It will also display the list of section action buttons when a section is selected.

```
var sectionEditActions = document.querySelector(".section-edit-actions")
```

```
function sectionEventListeners() {
```

```
  sectionEditActions.classList.add("shown")
```

```
  function clearSelectedSections() {
```

```
sectionEditActions.classList.remove("shown")
```

To add functionality to the section edit actions, I added event listeners to the four of them in the top-level of the JavaScript code. As the settings page is yet to be implemented, I have not added any functionality to it yet

```
// section action event listeners
document.getElementById("section_edit_action_settings")
    .addEventListener("click",e=>{})

document.getElementById("section_edit_action_duplicate")
    .addEventListener("click",e=>{ addSection(selectedSection.outerHTML) })

document.getElementById("section_edit_action_lock")
    .addEventListener("click",e=>{
        selectedSection.setAttribute("data-kraken-locked","");
    })

document.getElementById("section_edit_action_delete")
    .addEventListener("click",e=>{ selectedSection.remove() })
```

This means that after adding and selecting a section, the editor looks like this:



It can also successfully duplicate and delete the section. Whilst it can also add the `data-kraken-locked` data tag to it, at the moment this does nothing.

Resizing Text Elements

Next up, I implemented the resizing functionality for text elements. There are two different versions of it: one for horizontal resizing (for elements such as text), and one for 2D resizing (for elements such as images). The horizontal resizing is useful for elements such as text as it means that the text will automatically add linebreaks where necessary, so two resize points are all that is required. As such, there is a selector in the JavaScript code that checks for the `data-kraken-editable-text` tag to see whether the element needs four corners or two.

The system of generating the element control box is broke up into a lot of functions that each generate individual parts of the element or combine them together.

The code makes use of the functions `resizable` and `draggable`, which are defined at the start of the code, similarly to `siteroot`, `elements`, and `sections`. They return a list of all the resizable and draggable elements respectively, based on criteria that considers tags such as `data-preview` and `data-locked`. Functions are used instead of repeatedly calling the `document.querySelectorAll` function as it means that the query used is only written once, making it easy to edit should some aspect of the syntax change.

changes to `/static/js/site-edit.js`

```
// return all unlocked resizable elements in the editor
function resizable() {
    return document.querySelectorAll(
        `#contains_site
        [data-kraken-section]:not([data-preview]):not([data-kraken-locked])
        [data-kraken-element][data-kraken-resizable]:not([data-kraken-locked])`)
}

// returns all unlocked draggable elements in the editor
function draggable() {
    return document.querySelectorAll(
        `#contains_site
        [data-kraken-section]:not([data-preview]):not([data-kraken-locked])
        [data-kraken-element][data-kraken-draggable]:not([data-kraken-locked])`)
}
```

```
// generates resize boxes for every resizable element
function createResizeBoxes() {
    var r = resizable() // get all resizable elements
    for (var element of r) {
        // remove the current resize box
        element.querySelectorAll(".element-resize-container").forEach((e) => {
            e.parentElement.removeChild(e)
        })

        // set the resize type based on the element tags
        type = "corners"
        if (element.hasAttribute("data-kraken-editable-text")) type = "horizontal"

        // create the resize box for resizable and draggable elements
        element.appendChild(createResizeBoxContainerElement(type,element,true))
    }

    // iterate through all draggable elements
    for (var element in draggable()) {
        // if the element is not resizable
        if (!(element in r)) {
```

```
// create the resize box for non-resizable elements
element.appendChild(createResizeBoxContainerElement("",element,false))
}
}
}
```

```
// returns a resize box container element with all required things inside
function createResizeBoxContainerElement(type,parent,corners) {
    // create the container element and add required classes
    container = document.createElement("div")
    container.classList.add("element-resize-container")

    if (corners) { // if it requires corners, append a corner container child
        if (type == "corners")
            container.appendChild(createResizeBoxFourCornersElement())
        if (type == "horizontal")
            container.appendChild(createResizeBoxHorizontalCornersElement())
    }

    // append an outline child
    container.appendChild(createResizeBoxEdgesElement(parent))

    return container
}
```

```
// returns the resize box edges element
function createResizeBoxEdgesElement(parent) {
    // create the outline element and add required classes
    outline = document.createElement("div")
    outline.classList.add("element-outline")

    // set the correct height
    outline.style.height=parent.querySelector("div").offsetHeight+"px"

    return outline
}
```

```
// returns an element containing four resize corners
function createResizeBoxFourCornersElement() {
    // create the container element and add required classes
    root = document.createElement("div")
    root.classList.add("resize-corners")

    // append each required corner to the element
    root.appendChild(createResizeBoxCornerElement("top-left"))
    root.appendChild(createResizeBoxCornerElement("top-right"))
    root.appendChild(createResizeBoxCornerElement("bottom-left"))
    root.appendChild(createResizeBoxCornerElement("bottom-right"))
```

```
    return root
}
```

```
// returns an element containing two horizontal resize corners
function createResizeBoxHorizontalCornersElement() {
    // create the container element and add required classes
    root = document.createElement("div")
    root.classList.add("resize-corners")

    // append each required corner to the element
    root.appendChild(createResizeBoxCornerElement("horizontal-left"))
    root.appendChild(createResizeBoxCornerElement("horizontal-right"))
    return root
}
```

```
// returns a resize corner
function createResizeBoxCornerElement(loc) {
    // create the corner element and add required classes
    corner = document.createElement("div")
    corner.classList.add("resize-corner")
    corner.classList.add(`resize-corner-${loc}`)

    // create the visual element and add required classes
    trigger = document.createElement("div")
    trigger.classList.add("resize-corner-visual-element")

    // add the visual element as a child of the corner element
    corner.appendChild(trigger)

    return corner
}
```

For these to be rendered properly, I now needed to define the `elementEventListeners` function that was commented out in `addSection`. The hover functionality is added in via CSS, and the `elementEventListeners` function adds the selected element functionality so that the resize box can be held on visible. Similarly to the `sectionEventListeners` function, it uses a `clearSelectedElements` function, like the `clearSelectedSections` function, to remove the `data-kraken-element-selected` tag from each element. To do this, it uses the `elements` function that returns a `NodeList` of all elements.

```
// creates the event listeners for every element
function elementEventListeners() {
    for (var element of elements()) {

        element.addEventListener("click", (e) => {
            clearSelectedElements()
```

```

        setSelectedElement(e.currentTarget)
        createResizeCornerEventListeners()

        createDragEventListeners()
    })

}

}

```

```

// removes the data-kraken-element-selected tag from every element
function clearSelectedElements() {
    for (var element of elements())
        element.removeAttribute("data-kraken-element-selected")
}

```

The final part was to make the actual resize event listeners. These would will for a `mousedown` event on the resize handles, then create `mousemove` and `mouseup` event listeners on said resize handles. The `mousemove` listener will change the position attributes of the handle so that it emulates where the mouse is, and the `mouseup` listener will locate the nearest grid row/column and update the `--position` variable of the resized element.

```

// creates event listeners for resize corners on currently selected elements
function createResizeCornerEventListeners() {
    // get all corners of every resizable element
    for (corner of document.querySelectorAll(
        `.${queryprefix_elementresizable} .element-resize-container .resize-corner`)) {

        // make a copy of the corner - this removes any existing event listeners
        // so that there are not multiple resize events being triggered
        newcorner = corner.cloneNode(true)
        corner.parentNode.replaceChild(newcorner, corner)

        corner = newcorner

        // mouse down event listener
        corner.addEventListener("mousedown", (e1) => {

            // get the clicked element
            current = e1.currentTarget

            type = ""
            classlist = current.classList
            postoken = 0

            // work out whether it is a horizontal resize or a 2D resizes by looking at
            // the classes of the elemenet
            if (classlist.contains("resize-corner-top-left") |
                classlist.contains("resize-corner-top-right") |

```

```
        classlist.contains("resize-corner-bottom-left") |  
        classlist.contains("resize-corner-bottom-right"))  
        type="xy"  
  
    if (classList.contains("resize-corner-horizontal-left") |  
        classlist.contains("resize-corner-horizontal-right"))  
        type="x"  
  
    // postoken defines which corner is being dragged, and how the code should  
    // change the attributes based on that  
    if (classList.contains("resize-corner-horizontal-left")) postoken=1  
    if (classList.contains("resize-corner-horizontal-right")) postoken=3  
  
    // get the start position of the mouse  
    startPos = [e1.clientX,e1.clientY]  
  
    // set the mouse move event listener  
    current.addEventListener("mousemove", (e2) => {  
  
        // get the clicked element  
        current = e2.currentTarget  
  
        // get the new position of the mouse  
        newPos = [e2.clientX,e2.clientY]  
  
        // get the full style of the current element  
        current_style = getComputedStyle(current)  
  
        // get the current transformation property by converting  
        // matrix(a,b,c,d,e,f)  
        // into  
        // ["a","b","c","d","e","f"]  
        current_transform = current_style.transform.replace(  
            "matrix","",).replace(")","").replaceAll(" ","").split(",")  
  
        //console.log(e2.clientX,startPos[0],e2.clientX - startPos[0])  
  
        // if it is a horizontal resize  
        if (type=="x") {  
            // get the difference in position since the last render  
            diff = newPos[0] - startPos[0]  
  
            // set the horizontal translation property of the matrix  
            current_transform[4] = String(parseInt(current_transform[4])+diff)  
  
            // set the new matrix as the transform property  
            current.style.transform = "matrix(" + current_transform.join(",") + ")"  
  
            // get the outline element of the resize box  
            outline=current.parentNode.parentNode.querySelector(".element-outline")  
            outline_style = getComputedStyle(outline)  
            outline_width = outline_style.width  
            outline_width = parseInt(outline_width.replace("px",""))
```

```

outline_left = outline_style.left
outline_left = parseInt(outline_left.replace("px",""))

outline_right = outline_style.right
outline_right = parseInt(outline_right.replace("px",""))

// if the element is being resized from the left handle
if (postoken == 1) {
    // set the style properties to reflect the transformation
    outline.style.left = String(outline_left + diff) + "px"
    outline.style.width = String(outline_width - diff) + "px"
}

// if the element is being resized from the right handle
if (postoken == 3) {
    // set the style properties to reflect the transformation
    outline.style.right = String(outline_right + diff) + "px"
    outline.style.width = String(outline_width + diff) + "px"
}

// set the start position so that the next render resize works properly
startpos = newpos

})

// set the mouse up event listener
current.addEventListener("mouseup", (e2) => {

    // set the new properties for the element
    elementResize(e2)

    // recreate the resize boxes and event listeners to remove any visual
    // errors and clear themousemove and mouseup event listeners
    createResizeBoxes()
    createResizeCornerEventListeners()

})
})
}
}

```

To make the code easier to read, the code that actually assigns the new `position` attribute to the element is in another function, `eventResize`, that takes the event object as an argument. To work out which column is the closest, it takes the `grid-template-columns` style attribute from the parent section's `section-grid` element, converts it into a list of offsets from the start of the grid, and then compares it to the cursor's offset from the start of the grid.

```
// event listener function for element resizing
function elementResize(e2) {
    // get the corner that has been dragged
    current = e2.currentTarget

    // set the selected element and section to the parents of the corner

    parent = current.parentElement
    while (!(parent.hasAttribute("data-kraken-element"))) {
        parent = parent.parentElement
    }

    setSelectedElement(parent)

    while (!(parent.hasAttribute("data-kraken-section"))) {
        parent = parent.parentElement
    }

    setSelectedSection(parent)

    // get the current section's grid and current element
    currentsection = document.querySelector(queryprefix_sectionselected
        +" section .section-grid")

    currentelement = document.querySelector(queryprefix_elementselected)

    // get the computed styles of said elements and any required parameters
    currentsection_style = getComputedStyle(currentsection)
    currentelement_style = getComputedStyle(currentelement)

    currentsection_columngap = parseInt(
        currentsection_style.columnGap.replace("px", ""))
}

currentsection_gridtemplatecolumns = currentsection_style.gridTemplateColumns
currentsection_startx = currentsection.getBoundingClientRect().left
currentelement_position = currentelement_style.getPropertyValue("--position")
    .split("/")

columnlist = []
cumulative = 0
snappoints = []

//console.log(currentelement)
//console.log(currentsection)

//console.log(currentsection_columngap)

// get a list of the template columns as integers
// eg [0,0,4,4,4,4,4,4,0,0]
for (var col of currentsection_gridtemplatecolumns.split(" "))
    columnlist.push(parseInt(col.replace("px", "")))
```

```

// get the cumulative values of the column positions
// eg [0,0,4,8,12,16,24,32,32,32] when the columngap is 0
// this gets the offset from the start of the grid, in pixels, of every column
for (var col of columnlist) {
    snappoints.push(cumulative)
    cumulative += col + currentsection_columngap
}

//console.log(snappoints)

type = ""

// get the type of resize based on the classes of the handle
if (classList.contains("resize-corner-top-left") |
    classList.contains("resize-corner-top-right") |
    classList.contains("resize-corner-bottom-left") |
    classList.contains("resize-corner-bottom-right"))
    type="xy"

if (classList.contains("resize-corner-horizontal-left") |
    classList.contains("resize-corner-horizontal-right"))
    type="x"

// if its a horizontal resize
if (type=="x") {

    //console.log("Old position variable: ",currentelement_position)

    // get the final x position
    // (from the reference of the start of the viewport)
    endx = e2.clientX
    //console.log("End x position: ",endx)

    // get the final x position
    // (from the reference of the start of the section grid)
    endxoffset = endx-currentsection_startx
    //console.log("End x offset from parent: ", endxoffset)
    //console.log("Column offsets: ", snappoints)

    // get the closest column to the final value
    closestsnappoint = closestValInArray(endxoffset,snappoints)
    //console.log("Closest column offset to end x offset: ", closestsnappoint)

    // get the index of the closest column, ie the column number - 1
    index = snappoints.indexOf(closestsnappoint)
    //console.log("Index of closest column: ", index)

    // set the column number for the position variable
    columnnumber = String(index+1)
    //console.log("Closest column number: ",columnnumber)
}

```

```

    // assign the new position based on the postoken
    currentelement_position[postoken] = columnnumber
    //console.log("New position variable: ",currentelement_position)

    // assign the new position property to the selected element
    currentelement.style.setProperty("--position",
        currentelement_position.join("/"))

}

}

```

```

function closestValInArray(num, arr) {
  prev = arr[0]
  for (val of arr) {
    if (val > num) return prev
    prev = val
  }
}

```

After some testing with the resize function, I realised that the `closestValInArray` was insufficient for snapping to the grid, as it simply went to the closest value below. I rewrote it so that it would find the closest, instead of the one lower

```

function closestValInArray(num, arr) {
  prev = arr[0]
  diff = Math.abs(num - prev)
  for (var val of arr) {
    var newdiff = Math.abs (num - val)
    if (newdiff < diff) {
      diff = newdiff
      prev = val
    }
  }
  return prev
}

```

The elements could now be resized using the resize handles that are rendered when an element is hovered or selected. The element cannot be resized to larger than the grid due to the `closestValInArray` being only snapping to the extremes as a maximum, and how the `grid-position` property is handled in CSS. The handles have a invisible area around them that also responds to the `mousemove` and `mouseup` events, so that it can still be moved if the user moves their cursor outside of the handle's area (by going vertically or moving too fast). The testing evidence of this can be found in Appendix B.

Dragging and Dropping Elements

The most important component of the editor is the ability to drag and drop elements. The code will work in a similar structure to how the resize functions do.

changes to /static/js/site-edit.js

```
// creates event listeners for all draggable elements
function createDragEventListeners() {
    // get every draggable element
    for (element of draggable()) {

        // create the mouse down event listener
        element.addEventListener("mousedown", (e1) => {

            // end the listener if a resize corner is being clicked, because otherwise
            // the drag and resize functions will be happening simultaneously
            if (e1.target.classList.contains("resize-corner") ||
                e1.target.classList.contains("resize-corner-visual-element"))
                return

            // get the current element, set it as being dragged, and fetch the content
            current = e1.currentTarget
            current.setAttribute("data-kraken-dragging", "")
            content = current.querySelector("div")

            // get the starting cursor position
            startpos = [e1.clientX, e1.clientY]

            // get the current --position value
            storedPosition = current.style.getPropertyValue("--position")

            // set the box shadow and background color for the element
            content.style.boxShadow = "0px 0px 8px 3px rgba(0,0,0,0.25"
            content.style.backgroundColor = "var(--colors-light)"

            // create the mouse move event listener
            current.addEventListener("mousemove", (e2) => {

                // get the selected element
                current = e2.currentTarget

                // store the new cursor position
                newpos = [e2.clientX, e2.clientY]

                // fetch the style of the selected element
                current_style = getComputedStyle(current)

                // get the differences in position
                xdiff = newpos[0] - startpos[0]
                ydiff = newpos[1] - startpos[1]
```

```
// get the bounding boxes of the section and element
sectionbox = selectedSection.getBoundingClientRect()
currentbox = current.getBoundingClientRect()

// get the element's width and height in pixels
currentwidth = current.offsetWidth
currentheight = current.offsetHeight

// get the top and left positions relative to the section boundary
relativetop = currentbox.top - sectionbox.top + ydiff
relativeleft = currentbox.left - sectionbox.left + xdiff

// set the element style so that it follows the cursor
current.style.position = "absolute"
current.style.setProperty("--position", "")
current.style.top = String(relativetop) + "px"
current.style.left = String(relativeleft) + "px"
current.style.width = String(currentwidth) + "px"
current.style.height = String(currentheight) + "px"
current.style.zIndex = "999999"

// set the start position so that the next render works properly
startpos = [e2.clientX, e2.clientY]

})

// set the mouse up event listener
current.addEventListener("mouseup", (e2) => {

    // get the dragged element, remove the dragging tag, and get its content
    current = e2.currentTarget
    current.removeAttribute("data-kraken-dragging")
    content = current.querySelector("div")

    // get the dragged element's final position
    endpos = current.getBoundingClientRect()

    // reset the styling used in the dragging animation
    content.style.boxShadow = ""
    content.style.backgroundColor = ""

    current.style.removeProperty("position")
    current.style.removeProperty("top")
    current.style.removeProperty("left")
    current.style.removeProperty("width")
    current.style.removeProperty("height")
    current.style.removeProperty("z-index")

    // set the element position back to what it was
    // while the new one is being calculated
    current.style.setProperty("--position", storedPosition)
```

```

        // calculate and set the new position property for the element
        elementDrop(e2,endpos)

        // duplicate the element to remove all used event listeners
        newcurrent = current.cloneNode(true)
        current.parentNode.replaceChild(newcurrent,current)
        current = newcurrent

        // recreate the element event listeners
        elementEventListeners()

    })
}
}
}

```

```

// event listener function for element drag-and-drop drop
function elementDrop(e2,endpos) {

    // get the dragged element
    current = e2.currentTarget

    // get the current section, and any required style parameters
    currentsection = document.querySelector(
        "[data-kraken-section-selected] section .section-grid")

    currentsection_style = getComputedStyle(currentsection)
    currentsection_columngap = parseInt(currentsection_style.columnGap
        .replace("px",""))
    currentsection_rowgap = parseInt(currentsection_style.rowGap.replace("px",""))
    currentsection_gridtemplatecolumns = currentsection_style.gridTemplateColumns
    currentsection_gridtemplatelrows = currentsection_style.gridTemplateRows
    currentsection_startx = currentsection.getBoundingClientRect().left
    currentsection_starty = currentsection.getBoundingClientRect().top

    // get the elements style and position values
    current_style = getComputedStyle(current)
    current_position = current_style.getPropertyValue("--position").split("/")

    //console.log("Position: ",current_position)

    columnlist = []
    cumulative = 0
    columnsnappoints = []

    // get a list of the template columns as integers
    for (var col of currentsection_gridtemplatecolumns.split(" "))
        columnlist.push(parseInt(col.replace("px","")))

```

```

// get the cumulative values of the column positions
// this gets the offset from the start of the grid, in pixels, of every column
for (var col of columnlist) {
    columnsnappoints.push(cumulative)
    cumulative += col + currentsection_columngap
}

// repeat for rows

rowlist = []
cumulative = 0
rowsnappoints = []

// get a list of the template rows as integers
for (var row of currentsection_gridtemplatерows.split(" "))
    rowlist.push(parseInt(row.replace("px","")))

// get the cumulative values of the row positions
// this gets the offset from the top of the grid, in pixels, of every row
for (var row of rowlist) {
    rowsnappoints.push(cumulative)
    cumulative += row + currentsection_rowgap
}

//console.log("Col snap points: ",columnsnappoints)
//console.log("Row snap points: ",rowsnappoints)

// get the end x and y positions
// (endpos is the bounding box of the element when it was dropped)
endx = endpos.left
endy = endpos.top

// get the end x and y positions in relation to the section
endxoffset = endx-currentsection_startx
endyoffset = endy-currentsection_starty

//console.log("End X Offset: ",endxoffset)
//console.log("End Y Offset: ",endyoffset)

// get the closest column/row offset point to the end offsets
closestcolumnsnappoint = closestValInArray(endxoffset,columnsnappoints)
closestrowsnappoint = closestValInArray(endyoffset,rowsnappoints)

//console.log("Closest Col snap point ",closestcolumnsnappoint)
//console.log("Closest Row snap point ",closestrowsnappoint)

// get the numbers of the column/row
columnindex = columnsnappoints.indexOf(closestcolumnsnappoint)
rowindex = rowsnappoints.indexOf(closestrowsnappoint)

//console.log("Col index: ",columnindex)
//console.log("Row index: ",rowindex)

```

```

// set the column and row numbers
columnnumber = String(columnindex+1)
rownumber = String(rowIndex+1)

//console.log("Col number: ",columnnumber)
//console.log("Row number: ",rownumber)
//console.log("Position: ",current_position)

// 0 = start row
// 1 = start column
// 2 = end row
// 3 = end column

// Set the new end positions based on the diff between original and new
current_position[2] =
  String(parseInt(current_position[2]) + parseInt(rownumber)
    - parseInt(current_position[0]))

current_position[3] =
  String(parseInt(current_position[3]) + parseInt(columnnumber)
    - parseInt(current_position[1]))

// set the new start positions
current_position[1] = columnnumber
current_position[0] = rownumber

console.log("Position: ",current_position)

// set the --position style variable
current.style.setProperty("--position",current_position.join("/"))

}

```

Grid Preview System

The functions went through many iterations and changes while being developed, including a number of cases where the element would snap to seemingly random grid spaces, before realising that I had mixed up row and column at one point. The development of both the drag-and-drop and resize functions was aided by the grid preview function in DevTools, that allows you to view where the grid rows and columns are. It looks like this:



My research into other solutions included finding similar systems, wherein users could see the grid spaces for where the element would land as they were dragging/resizing it. Squarespace, for example, used an image per row for showing the preview. As this wouldn't work with the way my sections are created, I decided to add a `section-grid-preview` element, which would contain elements for every rendered square and mimic the CSS of the `section-grid` element.

changes to `/static/js/site-edit.js`

```
function createGridPreviews() {
    // for every section
    for (var section of document.querySelectorAll(
        "[data-kraken-section]:not([data-preview]) section")) {

        // remove any existing grid previews in the section
        for (var existing of section.querySelectorAll(".section-grid-preview")) {
            existing.parentNode.removeChild(existing)
        }

        // create the grid container element and add required classes
        container = document.createElement("div")
        container.classList.add("section-grid-preview")

        // get the style of the section grid and any required parameters
        gridstyle = getComputedStyle(section.querySelector(".section-grid"))
        gridstyle_rows = gridstyle.gridTemplateRows.split(" ")
        gridstyle_columns = gridstyle.gridTemplateColumns.split(" ")
        gridstyle_rowgap = parseInt(gridstyle.rowGap.replace("px", ""))
        gridstyle_columngap = parseInt(gridstyle.columnGap.replace("px", ""))

        columnoffset = parseInt(gridstyle.getPropertyValue("--block-padding-left"))
        rowoffset = parseInt(gridstyle.getPropertyValue("--block-padding-top"))

        //console.log(gridstyle.gridTemplateColumns)

        // for every column in the grid
        for (var columnwidth of gridstyle_columns) {

            // set the rowoffset back to the starting value
            rowoffset = parseInt(gridstyle.getPropertyValue("--block-padding-top"))

            // for every row in the column
            for (var rowheight of gridstyle_rows) {
                //console.log(columnoffset, rowoffset)

                // add a child box with the correct positions
                container.appendChild(
                    createGridPreviewBox(columnwidth, rowheight, rowoffset, columnoffset))

                //console.log(parseInt(rowheight.replace("px", "")))
            }
        }
    }
}
```

```

        // increase the row offset by the width of the row and the row gap
        rowoffset = rowoffset + parseInt(rowheight.replace("px",""))
            + gridstyle_rowgap
    }

    // increase the column offset by the width of the column and the column gap
    columnoffset = columnoffset + parseInt(columnwidth.replace("px",""))
        + gridstyle_columngap

}

// add the grid preview
section.appendChild(container)
}
}

```

The creation of a grid preview box is in its own function. It takes the width, height, position from the top of the section, and position from the left of the section, as arguments, and uses them when assigning the CSS. The box itself is an `absolute` element with a parent `relative` element, meaning that `top` and `left` can be used to position it anywhere inside the parent.

```

// creates a single box for the grid previews
function createGridPreviewBox(width,height,top,left) {
    // create the box element and add any required classes
    box = document.createElement("div")
    box.classList.add("section-grid-preview-box")

    // set the style to match the arguments
    box.style.width = width
    box.style.height = height
    box.style.top = top+"px"
    box.style.left = left+"px"

    // hide the box if one of the dimensions is 0
    if (width == "0px" || height == "0px") box.style.opacity = "0"

    return box
}

```

By default, the grid preview is kept hidden in CSS, so I added code to the `createResizeCornerEventListeners` and `createDragEventListeners` functions that shows and hides it when an element is being resized or dragged.

```

// get the parent section by iterating through every parent element
parent = current.parentElement
while (!(parent.hasAttribute("data-kraken-section")))
    parent = parent.parentElement

```

```
// render the grid preview element
parent.querySelectorAll(".section-grid-preview").forEach((e)=>{
  e.setAttribute("data-kraken-visible","");
})
```

```
// get the parent section by iterating through every parent element
parent = current.parentElement
while (!(parent.hasAttribute("data-kraken-section")))
  parent = parent.parentElement

// hide the grid preview element
parent.querySelectorAll(".section-grid-preview").forEach((e)=>{
  e.removeAttribute("data-kraken-visible")
})
```

Now, when elements are being dragged or resized, this background is displayed:



And when the operation is finished, the grid is hidden.

Stage 7 - Loading from and Saving to the Server

Loading from the Server

One of the most important components of the website is the ability to load content from and save content to the server-side storage system, so that users can easily edit their websites instantly. To do this, I looked into different ways of sending information via JavaScript, and found that jQuery's AJAX system would be the best suited.

AJAX (or Asynchronous JavaScript And XML) allows client-side applications to communicate with remote servers in an asynchronous way (without interfering with the behaviour of the webpage). It essentially means that I can make dynamic data requests from and to the server. On a rewrite, I would probably use AJAX to handle the importing of sections to make the code consistent, but as the current code works fine, and I am on a strict time schedule, I won't worry about doing that now.

I started by creating the `run` command, which is where all of the code necessary to begin the webpage will be contained, such as event listener creation and dynamic content generation. It is asynchronous, to allow for the loading of user content. For any fetching of user content, I loaded it via the `fetch` command and handled it with `.then`. Any required functions are `await`ed to ensure they are finished before the program continues.

changes to /static/js/site-edit.js

```
// called on the page being loaded
// loads content from the server, starts all required event listeners,
// creates any required resize boxes, and starts the autosave timer
async function run() {
    // get the username and sitename from the site url
    user = window.location.pathname.split("/")[1]
    site = window.location.pathname.split("/")[2]

    // get the siteinfo dictionary from the server
    siteinfo = await fetch(
        `../../../../static/data/userData/${user}/sites/${site}/siteDat.json`)
        .then(response => { return response.json(); })

    // get the current site page, defaults to "Home" if nothing is set
    currentpage = siteinfo["pages"][currentsitename]

    // load the current page
    await loadcurrentPage(user,site,currentpage)

    // create any required event listeners
    sectionEventListeners()
    elementEventListeners()

    // create any required resize boxes for elements and grid previews for sections
    createResizeBoxes()
```

```
createGridPreviews()

// import the template sections
addSectionModal_loadsections()

// constant checks for stuff
document.querySelector(".site-builder").addEventListener("click",() => {
    sectionEditActions_checkVisibility()
    createResizeBoxes()
    createGridPreviews()
})
}
```

The asynchronous function `loadcurrentpage` loads the required HTML into the editor via the `sethtml` and `loadhtmlfile` commands, based on the arguments `user`, `site`, and `currentpage`.

```
// loads the current page into the editor from a set criteria
async function loadcurrentpage(user,site,currentpage) {
    return sethtml(await loadhtmlfile(
        `../../static/data/userData/${user}/sites/${site}/files/${currentpage}`))
}
```

The asynchronous function `loadhtmlfile` takes a path and fetches it. It will return the given content if successful.

```
// loads a file from a path
async function loadhtmlfile(path) {
    return await fetch(path)
    .then( response => {
        if (!response.ok) { throw new Error(`HTTP error: ${response.status}`) }
        return response.text()
    })
    .then( text => {
        return text
    })
}
```

The `sethtml` function sets the editor content to a given HTML string.

```
// sets the content of the editor
function sethtml(html) {
    builder.innerHTML=html
}
```

This now means that, when the webpage is loaded, the content of the default home file, `1.html`, in the server directory `/static/data/userData/<username>/sites/<sitename>/files/1.html` is loaded into the editor for the user.

Saving to the Server

Saving to the server is a bit more complicated as it requires use of the AJAX request methods. This is the solution I came up with.

changes to `__init__.py`

The `app.route` function `site_edit_save` has been added to the Python code. It responds to `post` methods on the url `/<username>/<sitename>/save/<pagename>`, which is what will be used to save content. From there, it fetches the data and saves it to the designated file in storage. No error catching to inform the server is required as that can all be handled by AJAX.

```
@self.app.route("/<name>/<site>/save/<page>",methods=[ "post"])
@login_required
def site_edit_save(name:str="",site:str="",page:str="") -> None:
    # get the content of the request - the HTML to save
    tosave:str = request.form.get("content")

    # get the path of the file to save from the arguments
    sitePath:str = self.os.path.abspath(
        f"static/data/userData/{name}/sites/{site}/files/{page}")

    # save to the file
    with open(sitePath,"w") as f: f.write(tosave)
    return "success"
```

changes to `/static/js/site-edit.js`

The asynchronous function `saveContent` will be called whenever the save button is pressed in the editor, and whenever the autosave process is triggered. It uses the jQuery function `ajax` to send a `POST` method request to the server, on the URL `/<username>/<sitename>/save/<pagename>`, with the content `siteroot().outerHTML` - i.e. the content of the editor. The parameters `error` and `success` define the functions to be called when the request is finished, based on the outcome. As of now, they simply write to the console. Testing of this using DevTools's Network Conditions functionality can be found in Appendix B.

```
// sends a request to the server to save the content of the editor
// to the server file storage
async function saveContent() {
    req = $.ajax({
        type:"post",
```

```

    url: `.../${window.location.pathname.split("/")[1]}/
      ${window.location.pathname.split("/")[2]}/save/
      ${siteinfo["pages"][currentsitename]}`,

    data:{content:siteroot().outerHTML},
    datatype:"json",
    error: function(xhr,ajaxOptions,thrownError) { console.log(thrownError) },
    success: function(response) { console.log("Saved") },
  )
}

```

The combination of these two functions mean that a user can now save their page when it is called, but there is no way of it being triggered. As such, I added a save icon to the local navigation bar. It also includes three hidden icons that are used in the saving animation.

changes to /templates/site-edit.html

```

<li class="localnav-item four">
  <a class="link unformatted" id="localnav_save">
    <div class="localnav-item-collapsible-icon">
      <i class="faicon fa-solid fa-save"></i>
    </div>

    <div class="localnav-item-collapsible-text text bold primary">
      Save Website
    </div>

    <div class="localnav-item-collapsible-icon saveloader">
      <i class="faicon fa-solid fa-loader"></i>
    </div>

    <div class="localnav-item-collapsible-icon savetick">
      <i class="faicon fa-solid fa-check"></i>
    </div>

    <div class="localnav-item-collapsible-icon savecross">
      <i class="faicon fa-solid fa-xmark"></i>
    </div>

  </a>
</li>

```

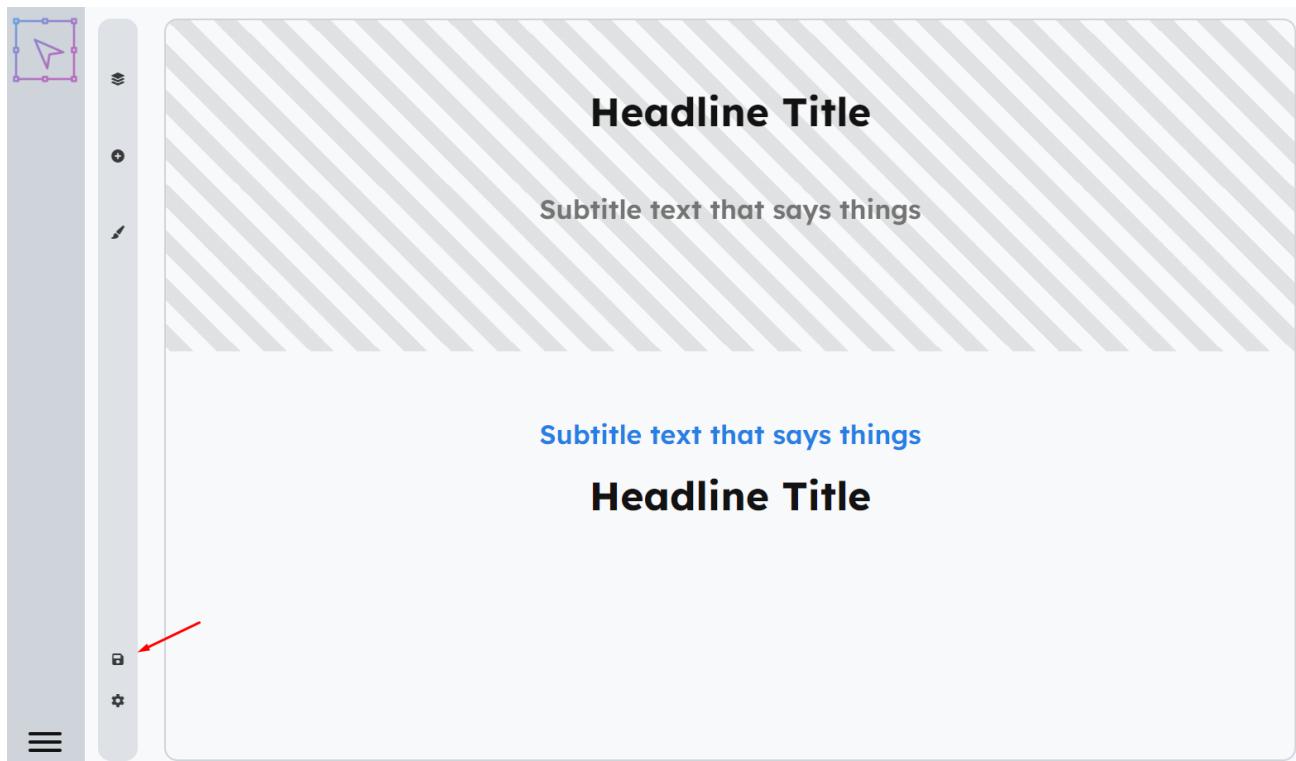
The CSS for when the icons are being shown is defined in `site-edit.css`. In short:

- When there is no `data-kraken-savestate` attribute:
 - `fa-save` is shown
 - All others are hidden
- When `data-kraken-savestate="saving"`:

- `fa-loader` is shown
- All others are hidden
- When `data-kraken-savestate="success"` :
 - `fa-check` is shown in green
 - All others are hidden
- When `data-kraken-savestate="error"` :
 - `fa-xmark` is shown in red
 - All others are hidden

All of the icons have opacity transitions with appropriate delays so that the animation looks nice. The `fa-loader` spins slowly to represent that something is happening.

By default, the icon looks like this:



changes to `/static/js/site-edit.js`

The functions `showSaveLoader` and `hideSaveLoader` are used to trigger the animation by updating the `data-kraken-savestate` data tag on the `.localnav-item.four` element.

```
// starts the save loader animation
function showSaveLoader() {
  document.querySelector(".localnav.one .localnav-item.four")
    .setAttribute("data-kraken-savestate", "saving")
}
```

The `hideSaveLoader` function is asynchronous as it makes use of the `sleep` function, which delays the upcoming code by a given amount of milliseconds, in this case 1500. What this means in context is that the element will show either the tick or cross for 1.5 seconds, then revert back to the default icon.

```
// finishes the save loading animation and shows a success/fail icon
// then resets to normal after 1500ms
async function hideSaveLoader(success) {
    document.querySelector(".localnav.one .localnav-item.four")
        .setAttribute("data-kraken-savestate", success?"success":"error")

    await sleep(1500)

    document.querySelector(".localnav.one .localnav-item.four")
        .removeAttribute("data-kraken-savestate")
}
```

```
// source https://www.grepper.com/answers/17868/javascript+sleep

const sleep = (milliseconds) => {
    return new Promise(resolve => setTimeout(resolve, milliseconds))
}

// endsource
```

The asynchronous function `waitForNotSaving` also makes use of the `sleep` function to ensure that the program waits until the current save has finished before executing another one. It checks the `.localnav-item.four` element for the `data-kraken-savestate` tag and delays by 100 milliseconds before checking again if it has the tag.

```
// await function used to ensure that multiple saves are not occurring
// at the same time
async function waitForNotSaving() {
    while (document.querySelector(".localnav.one .localnav-item.four")
        .hasAttribute("data-kraken-savestate")) {
        await sleep(100)
    }

    return true
}
```

I then combined this all together with a `setTimeout` function for the autosave feature, and an event listener on the save icon. The `setTimeout` function creates a delay, in milliseconds, before a given function is called, meaning that I can call it at the end of the `saveContent` function to create a loop. In the AJAX code in `saveContent`, I also modified the existing `error` and `success` functions to call `hideSaveLoader` with the appropriate argument for the success of the request

```
async function saveContent() {
    // wait for the current save to finish, if running
    await waitForNotSaving()

    console.log("Saving...")

    // start the save animation
    showSaveLoader()

    req = $.ajax({
        ...
        error: function(xhr, ajaxOptions, thrownError) { hideSaveLoader(false) },
        success: function(response) { hideSaveLoader(true) },
    })

    // create the new timeout
    saveTimeout()
}
```

The `saveTimeout` function is used to create the timeout for the next call of the function. The value, 60000, represents the time, in milliseconds, before it is called. In this case, it equates to one minute.

```
// starts the timeout for the saveContent function
function saveTimeout() {
    setTimeout(saveContent, 60000) // run the savecontent function every minute
}
```

```
async function run() {
    ...
    // previous code

    // start the autosave process
    saveTimeout()

}
```

After running preliminary tests on a lower timeout delay, I found that it could very quickly get to the point where it was constantly running the saving animation. As such, I added the `clearTimeout` function to make sure that only one timeout was running for the `saveContent` function at a time.

```
// starts the timeout for the saveContent function
function saveTimeout() {
  clearTimeout(saveContent) // clear any previous timeouts
  setTimeout(saveContent, 60000) // run the savecontent function every minute
}
```

```
async function saveContent() {
  // remove all existing timeouts to reduce conflicts
  clearTimeout(saveContent)

  ... // previous code

}
```

In Appendix B, I used DevTools' Network Conditions throttling settings to test how it worked on lower connections, including when the site was offline.

Evaluation

Due to time constraints, I have had to finish the first release of the project in its current state. Although not fully functional, the project fulfills a large portion of the success criteria outlined in the Analysis section.

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.

The login/signup system is fully functional with integration to the server's SQL database, that uses two tables: `User` and `Site`. The user can view their passwords before submitting, and there is appropriate validation on both the client-side and server-side, which lines up with the criteria defined above. The data is validated client-side because it means that the user can receive instant feedback and reduce server load, and it is validated server-side as well to ensure that the client-side validation hasn't been tampered with. Testing was performed during development on both instances of the validation to ensure that the features all worked properly. The backend implements the `flask-login` system to integrate users being able to login. This means that the backend can access information about the current logged-in user for the session that's being requested, which is useful for executing SQL queries about the specific user. The system fully meets this criteria.

- Homepage
 - 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

The homepage renders any of the user's current sites, if there are any, by using an SQL query. The backend will return two different templates based on whether the user has any sites. The system fully meets this criteria.

- **Site Creation**

- 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

On the first page of the website creation system, the user is prompted for a site name, an optional description, and privacy flag. Similarly to the login system, the validation system for the site name is executed both client-side and server-side to negate any risks of malicious attacks. The system fully meets this criteria.

- **Site syntax**

- 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 & grey colours, and primary and secondary fonts.

The if not logged in, as of now, the `/<username>/<sitename>` page currently redirects to the login page. If the current user is logged in but no the owner, it receives a message stating that they are not the owner, regardless of the privacy settings. This feature has not yet been implemented, and as such, does not meet the criteria.

The site config file is written to when the site is created (as of yet there is no way to change it), and contains the settings, such as the website name, owner, and description; colours, including `light`, `dark`, `primary`, `secondary`, `accent` & their variants, and the `grey-100` through `grey-900` variables; and fonts, including `header` and `body`.

For example:

```

[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

```

- CMS

- File storage system to store user site files
- CMS system for users to be able to upload custom content

This is currently not implemented into the project, partially due to the amount of time required to implement it with use of validation and malware checking. Hence, the project does not meet this criteria.

- Editing a site

- 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.

The `/<username>/<sitename>` is handled partially by the back-end of the system, in that it can return the site if the logged-in user is the owner of the site, but otherwise will simply return a message for the user stating that they can't view the website. The project partially meets this criteria.

- 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.

The site page does not render a preview of the current user's site, or give customizability options for the website - except for the link to edit the site. The project does not meet this criteria.

- 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 editing page reflects some of the design set out in the success criteria: there is a local navigation bar on the left containing the options given in the success criteria; there is a central canvas that contains the page that is currently being edited; there is a popup modal that appears when adding a section, but not when adding an element, as that functionality has not yet been implemented; and there is no styling section on the right hand side. The project partially meets this criteria.

- 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.

The client-side JavaScript successfully fetches the required HTML from the server - as of yet the project does not require the importing of any CSS. The JavaScript uses the data tags, such as `data-kraken-section` to parse the HTML and create appropriate event listeners and control elements. The project meets this criteria.

- 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.

An element, when hovered or selected (identified with the `data-kraken-element-selected` tag), shows a box around it depending on whether it is resizable and/or draggable (identified with the `data-kraken-resizable` and `data-kraken-draggable` tags respectively), and a set of "corners" on the box which, when dragged, will resize the element. The elements are contained in a child element of the element which contains the outline and any corners required. The elements can have either two corners on either side, or four corners on the vertices, depending on its type (such as `data-kraken-text-element`). The code, despite having the multiple corner style functionalities, only resizes when horizontal corners are present. Despite this, the project meets this criteria.

The style menu on the right has not yet been implemented, and, as such, the project does not meet this criteria.

- 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 dragging and dropping system is defined with the `data-kraken-draggable` tag. Any elements with the tag will be assigned `mousedown` event listeners for when they are dragged. It is important to note that these events cancel if a resize corner is detected as the element that has been clicked, as otherwise the element would be simultaneously being dragged and resized. As the element is being dragged, the code listens for `mousemove` events, and updates the element's `top` and `left` CSS properties to ensure that the element properly tracks to the cursor when being moved. The element is also rendered with a solid background and drop shadow to indicate that it has been "picked up".

All sections have a hidden `section-grid-preview` element inside, that contains the content needed to render the outline of the parent section's grid system, so that when an element is being dragged, the user knows where the element will snap to when dropped. This element is added in by the JavaScript, and is shown via the `data-kraken-visible` tag. Due to the nature of how it has been created, it will render the correct positions of each of the grid areas even if the content area (i.e. the browser window the user is viewing it on) has been resized.

The project meets this criteria.

- 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.

The project does not meet this criteria, as the plan for developing the editor was to ensure that the content for desktop views of the web page were finished before moving on to mobile, and the desktop implementation has not been completed yet.

- When a widget is right-clicked, it will show useful commands such as copy, paste, delete and duplicate.

This functionality has not been added, and, as such, the project does not meet this criteria.

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.

The ability to import and export zip files of the site is not a required feature, meaning they are not essential to the finished project and do not have to be implemented. Due to time constraints, they were not developed. While they would add another level of portability to the system that is not required, the amount of time needed to develop it, in comparison to the usefulness it adds, is not justified in comparison to the other, more important features.

- An easy-to-use settings interface where the user can customise things such as public profile appearance and account settings.

The settings pages, despite being mostly added to the project, do not have any back-end functionality. They have been added as it was fast and easy to create the HTML for them, and it means that when the features are developed in a later iteration of the project, myself or another developer will find it easier to implement. The pages currently have indicators that there is no functionality to the displayed features.

- A list of predefined templates for sites when creating a site, that can be organised and filtered via relevant tags.

There is currently no list of templates that the user can select from when creating a site. This is due to the amount of time it would take to create and categorise each template. When the editor is completed, it will be easier to create them and will mean that they will have native compatibility with it, so their creation has been delayed until then.

- 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.

During the site creation process, the user can select options such as colours to build a colour palette, and header-paragraph font combinations. The colour options include a light/dark theme selection, the colour temperature of the monochromatic colours (tinting their white balance), the light and dark bounds (the maximum lightness and darkness of the monochromatic colours), and selections of the primary, secondary, and accent colors (including generation of light and dark variants). This means that the project meets this criteria, as, despite it only being desirable, it is an important component of the usability of the editor.

- 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.

This has not been implemented into the project, as it is a feature that is quite far down the development timeline, and not required for the project to be fully functional. It would require the database to be recreated to allow for a linking table between the two original tables, and more complex back-end functionality, which would take a lot of time to implement.

- 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.

Despite accessibility support and audience interaction features being important components of modern websites, they have not yet been implemented into the editor as more important features have been prioritised due to time constraints.

- 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.

The import/export functionality has not yet been added due to the unfinished editor, and there is not yet being much use for it. The import feature especially would take a lot of effort to implement due to the complexity required for malware checking. As such, it is a feature that has been delayed until later in the application development.

Overall, a large portion of the desirable features have not been implemented due to the prioritisation of more important functionality, in combination with time constraints.

Usability

Having discussed the usability of the system with the stakeholders

Performance & Maintenance

Security

User Satisfaction

Accessibility

Overall Comments