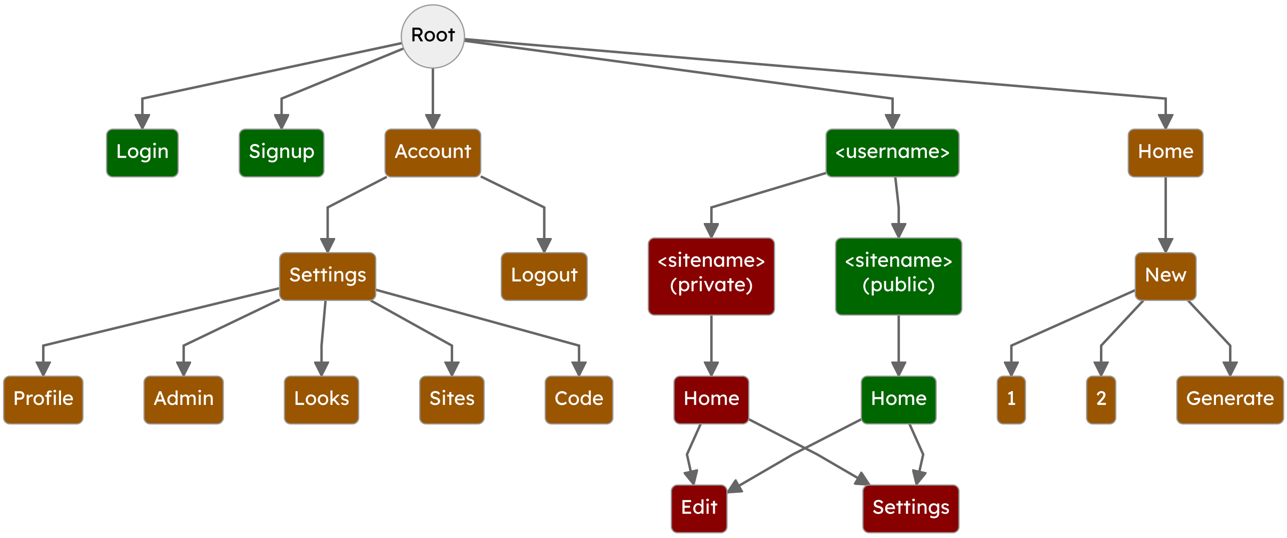
Design

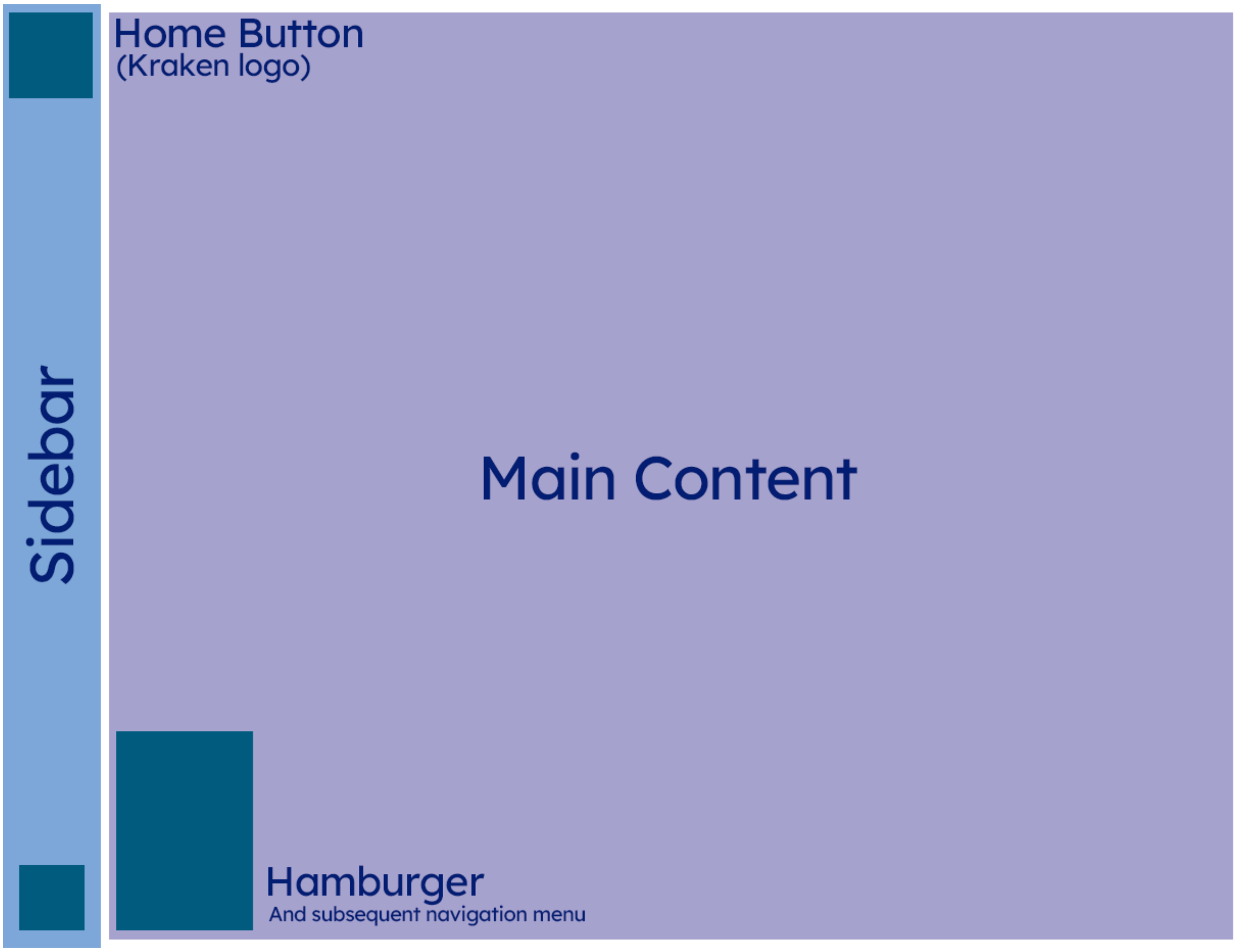
URL Navigation



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.

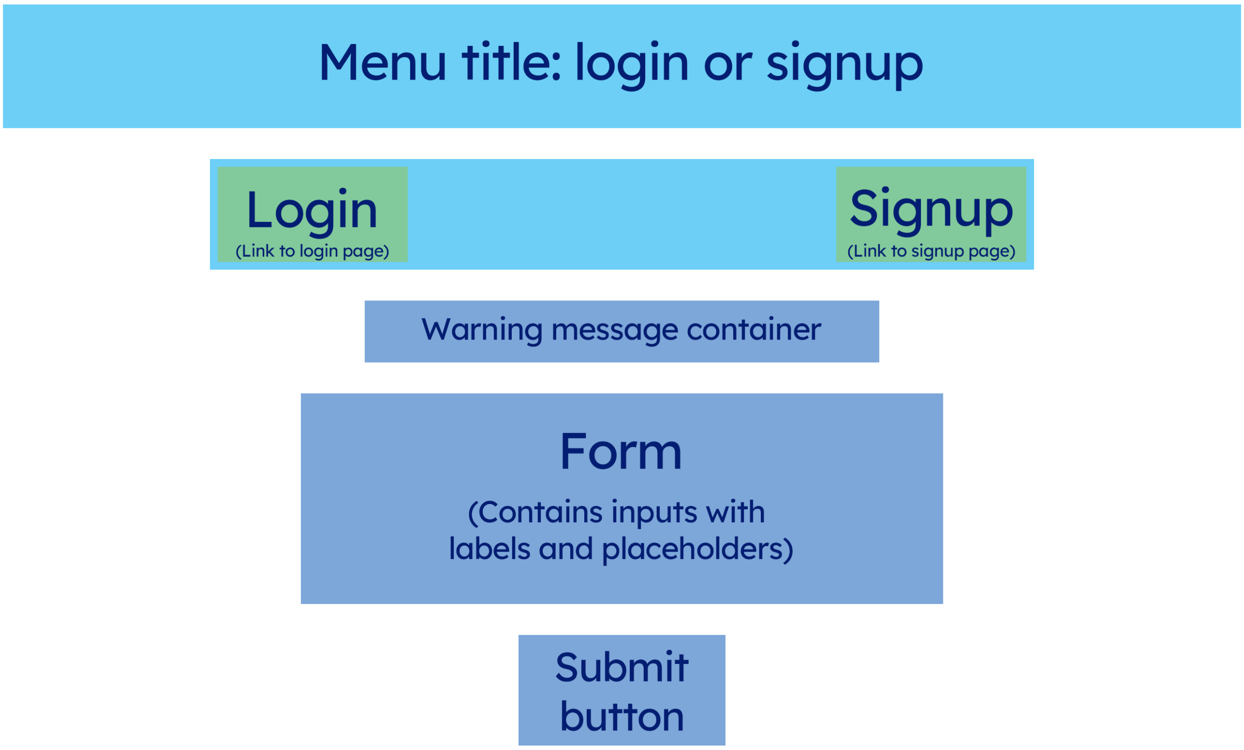
Hamburger

> 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.

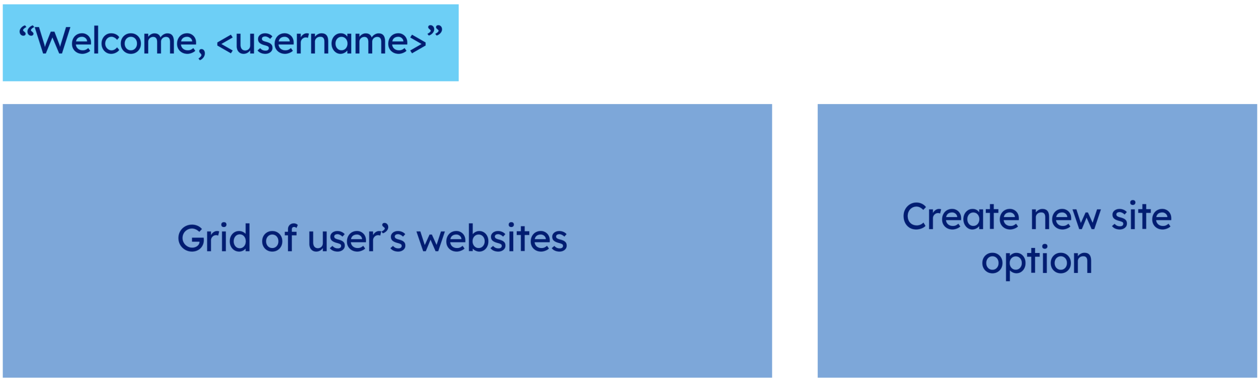
Login and Signup Pages


Menu title: login or signup
Login
(Link to login page)
Signup
(Link to signup page)
Warning message container
Form
(Contains inputs with labels and placeholders)
Submit button

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

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

Homepage



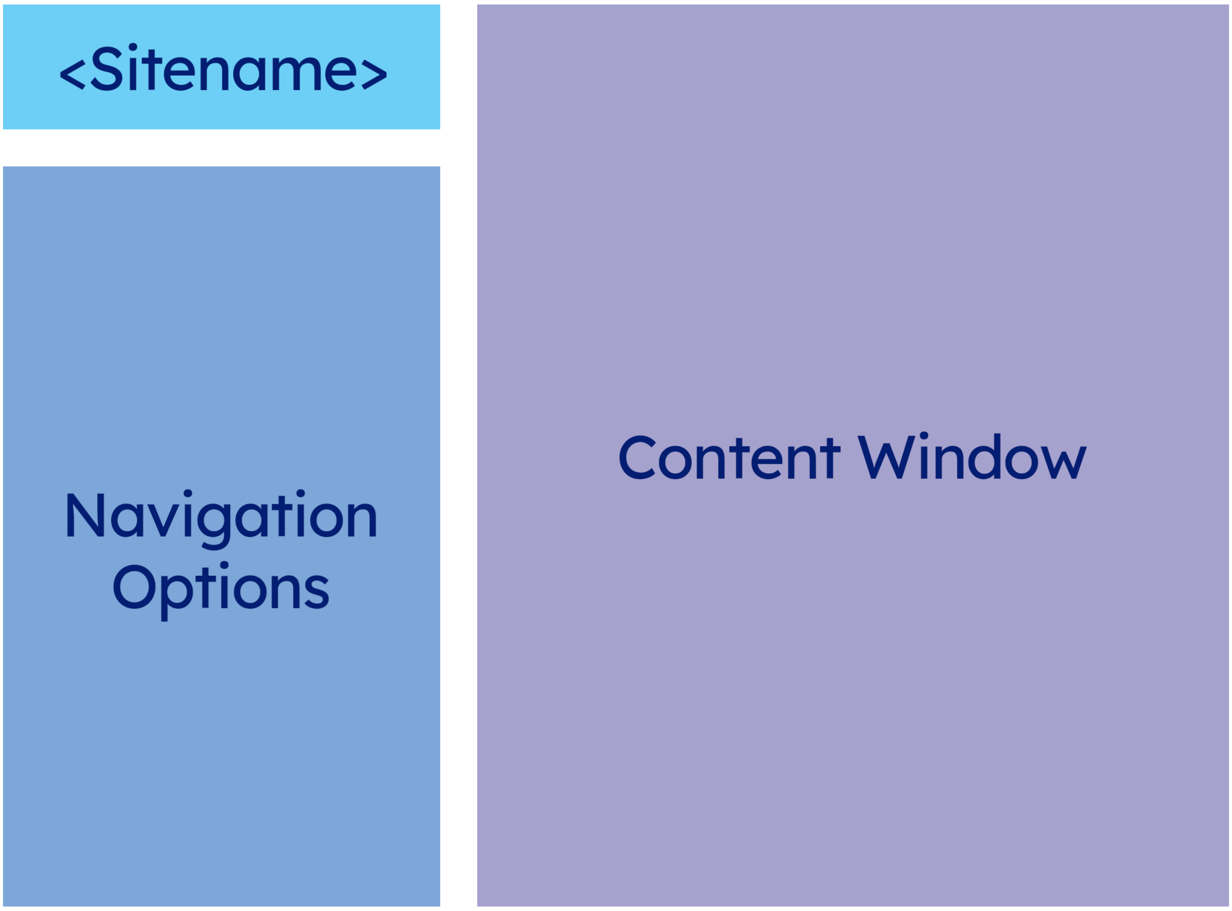
This shows the layout of the homepage once the user has logged in, if the user already has created a website. If the user has not made any websites yet, it will look similar to this but without the grid of their websites. Built inside the main template, it contains the following:

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

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

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

Site Home



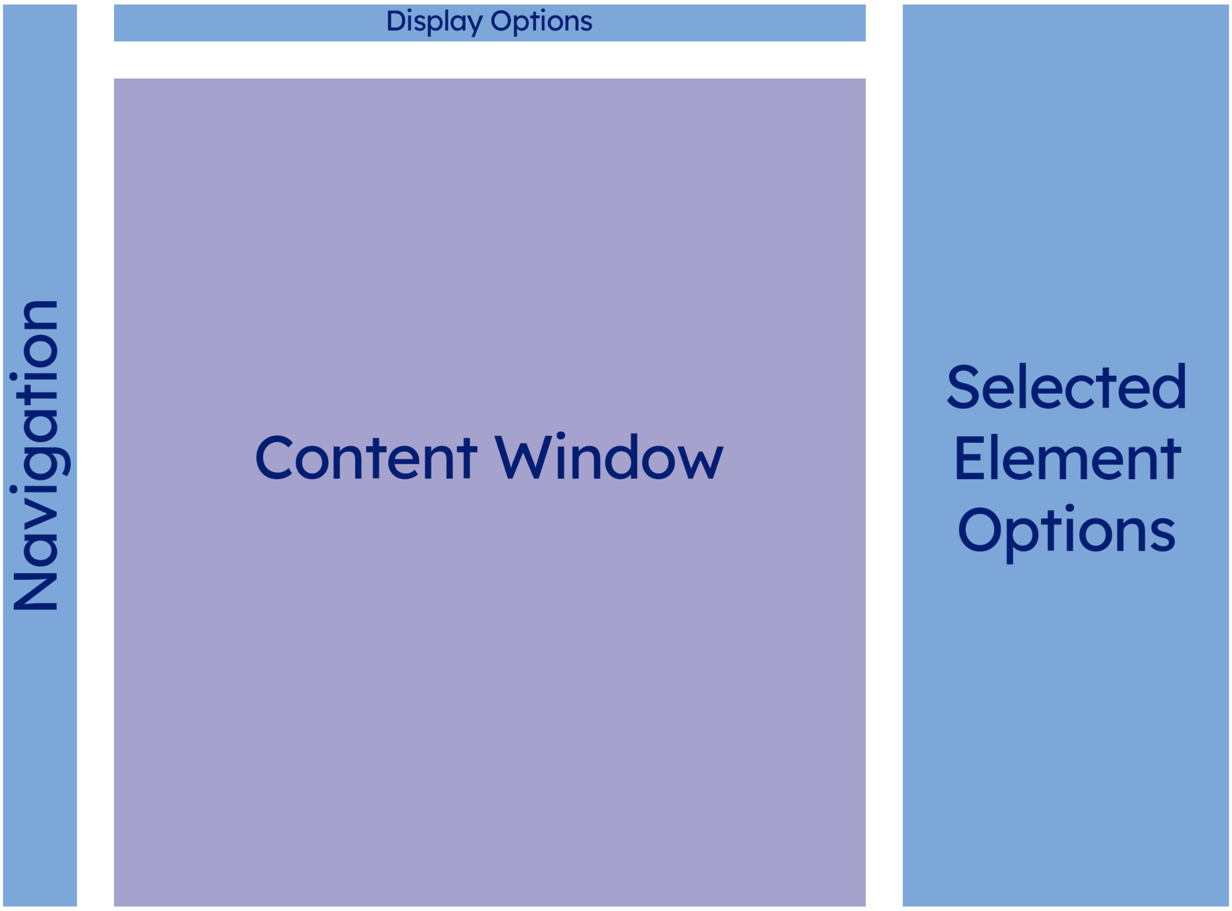
This shows the layout of the site page for the owner (when they visit `/<username>/<sitename>`). Built inside the main template, it contains the following:

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

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

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

Site Edit



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

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

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

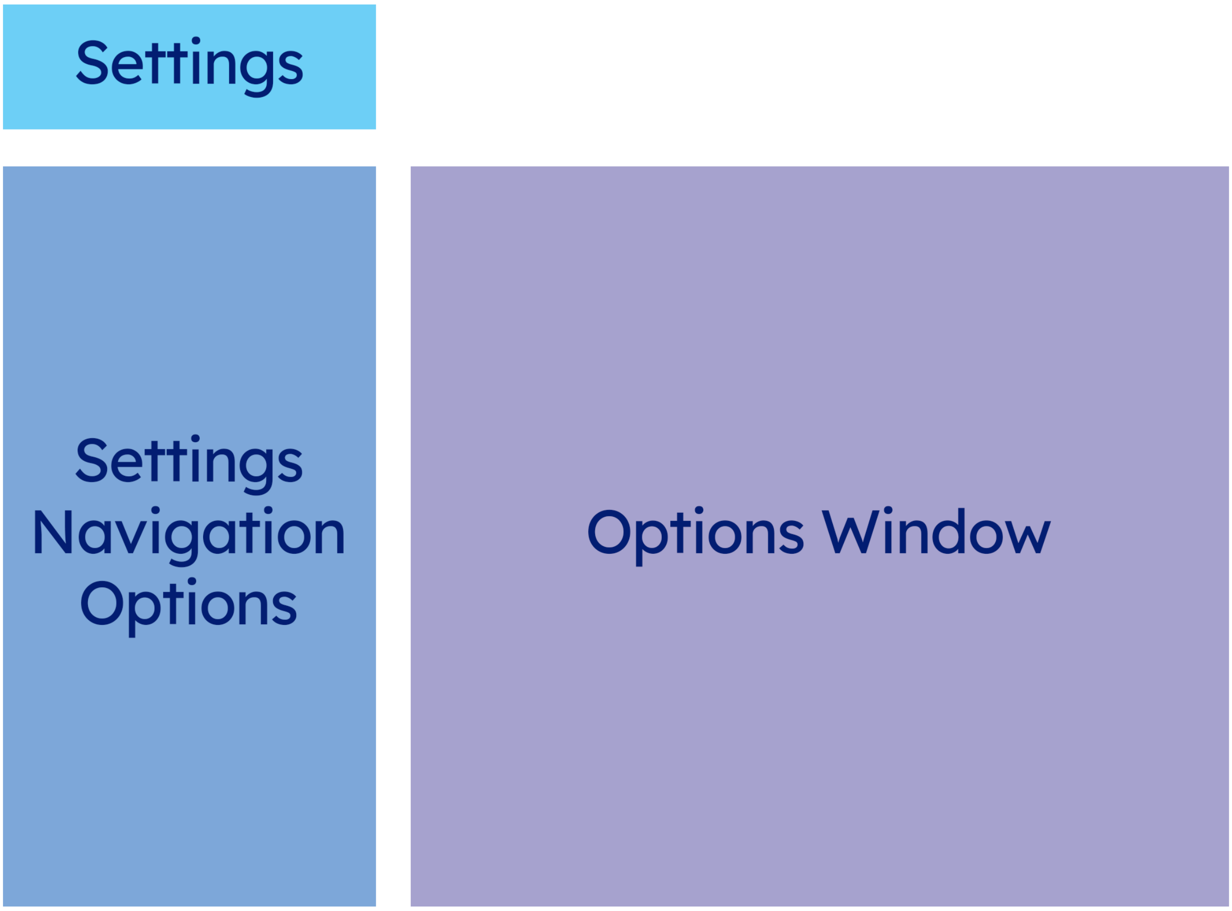
- The display options bar, docked to the top, will contain some settings for how the editor is shown and are put here so the user can easily access them.

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

- The selected element options, docked to the top, will contain quick-access site settings until an element is selected in the content window. When an element is selected, it will display style settings.

- The content window will display the site so the user can edit it. There is more information on the mechanics of this section in other parts of the report.

User Settings Pages



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

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

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

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

Usability

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

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

Accessibility

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

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

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

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

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

ARIA

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

Stakeholder input

Website structure and backend

Flask

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

Jinja

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

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

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

base.html

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

    {% block content %}
    {% endblock %}

  </body>
</html>
```

itemlist.html

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

{% block content%}

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

{% endblock %}
```

main.py

```
from flask import Flask, render_template

app = Flask(__name__)

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

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

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

HTML, CSS and JavaScript

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

Data storage

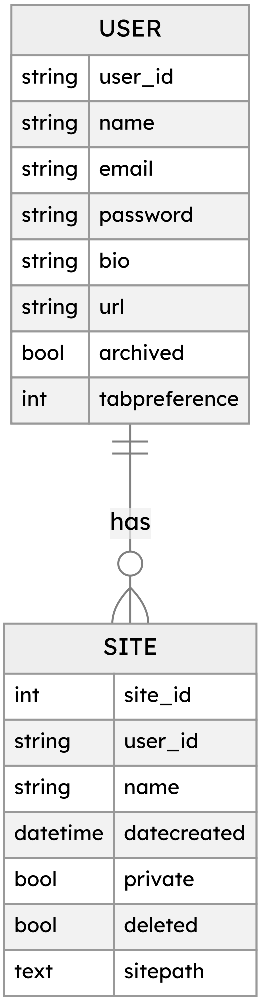
There will be two different methods of storing information for the website:

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

SQL database storage

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

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

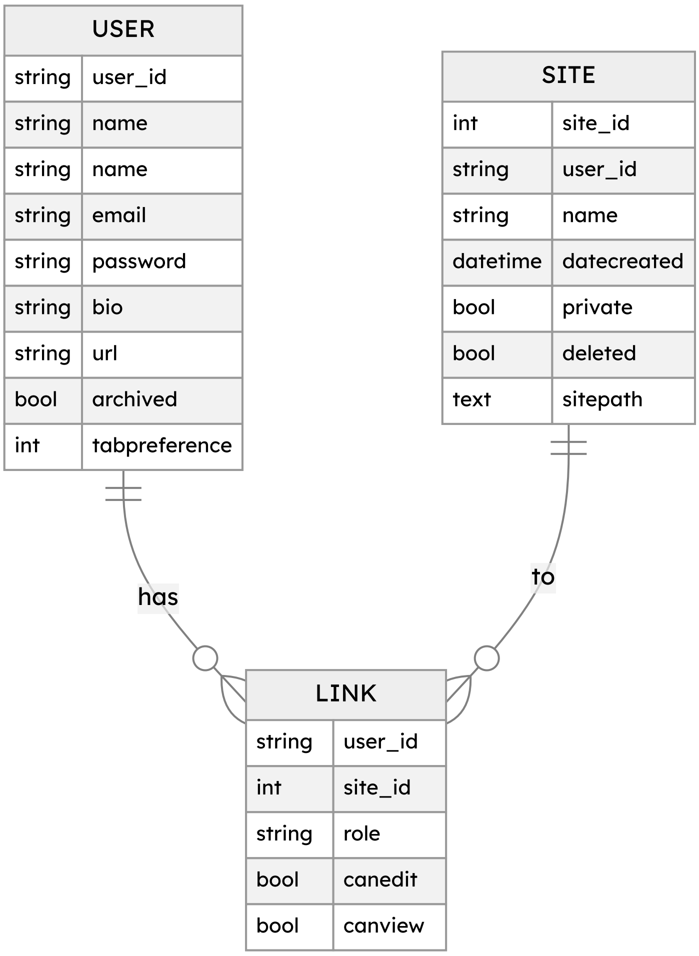


USER	
string	user_id
string	name
string	email
string	password
string	bio
string	url
bool	archived
int	tabpreference

SITE	
int	site_id
string	user_id
string	name
datetime	datecreated
bool	private
bool	deleted
text	sitepath

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

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



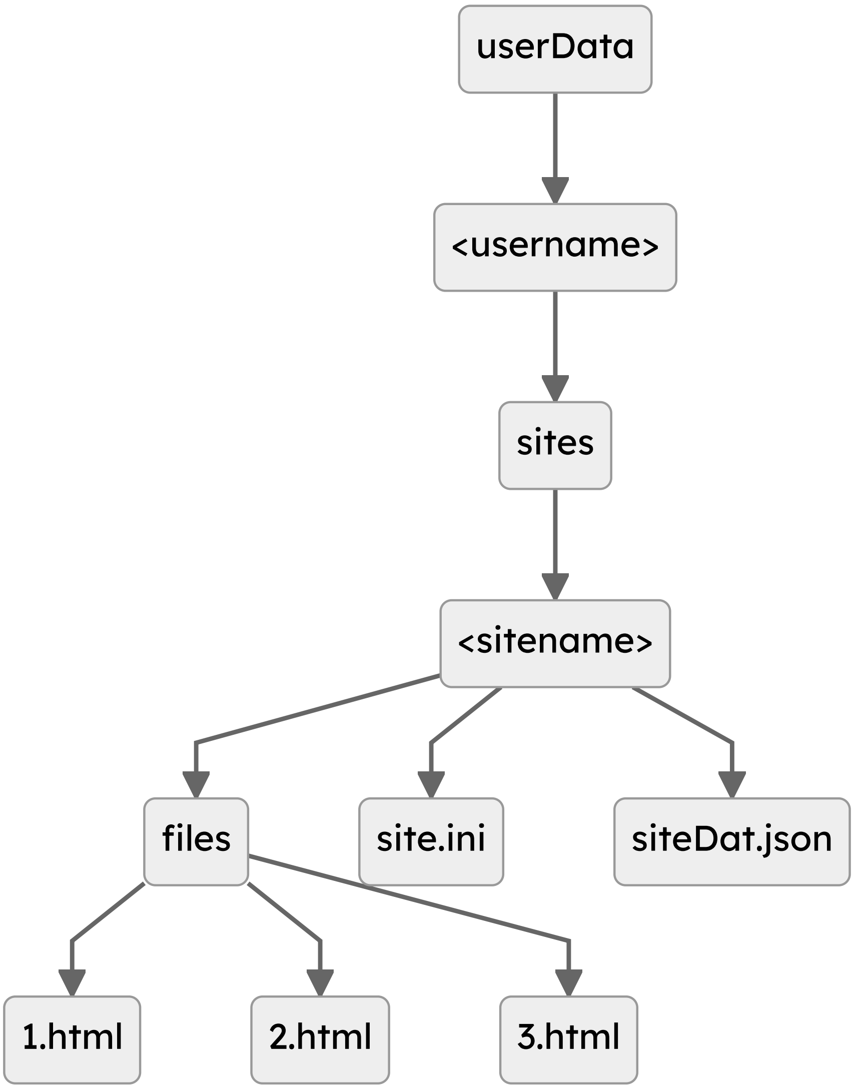
USER	
string	user_id
string	name
string	name
string	email
string	password
string	bio
string	url
bool	archived
int	tabpreference

SITE	
int	site_id
string	user_id
string	name
datetime	datecreated
bool	private
bool	deleted
text	sitepath

LINK	
string	user_id
int	site_id
string	role
bool	canedit
bool	canview

Server-side file storage

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



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

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

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

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

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

Algorithms

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

Drag-and-drop editor algorithms

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

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

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

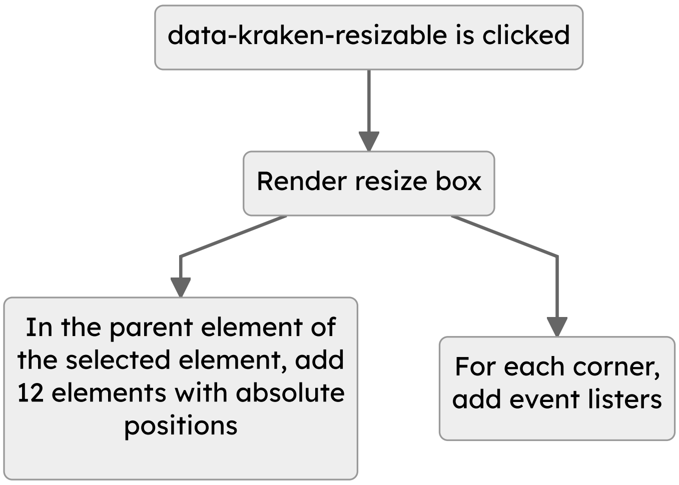
- `data-kraken-resizable`
- `data-kraken-draggable`
- `data-kraken-editable-text`
- `data-kraken-editable-style`
- `data-kraken-locked`

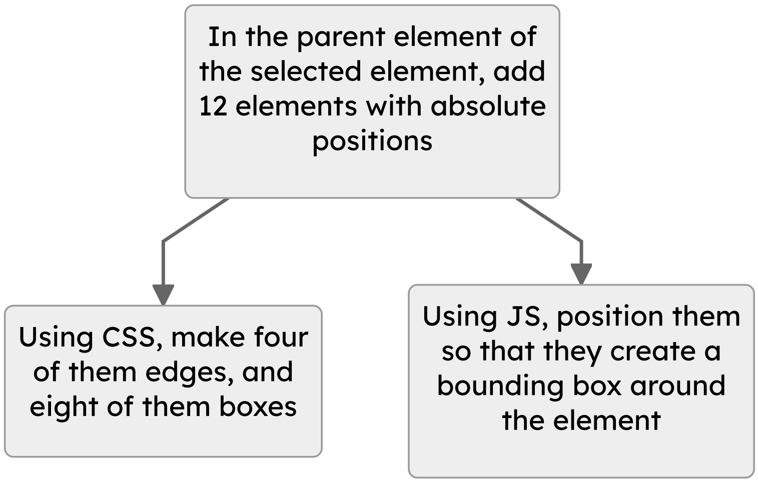
These will define which functionalities can be used for each element. For all of the below diagrams, if the element has the tag `data-kraken-locked`, it will only show a button next to the element/section to unlock it.

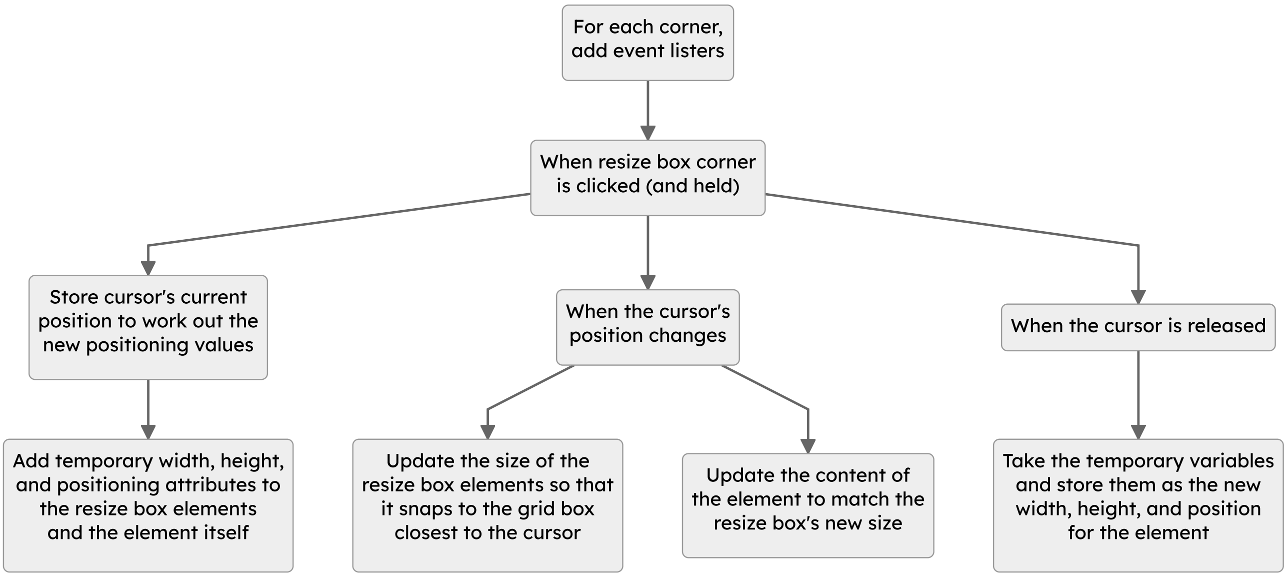
When an element is selected, depending on its function, it will be tagged with one of these attributes so that the JavaScript can easily edit it:

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

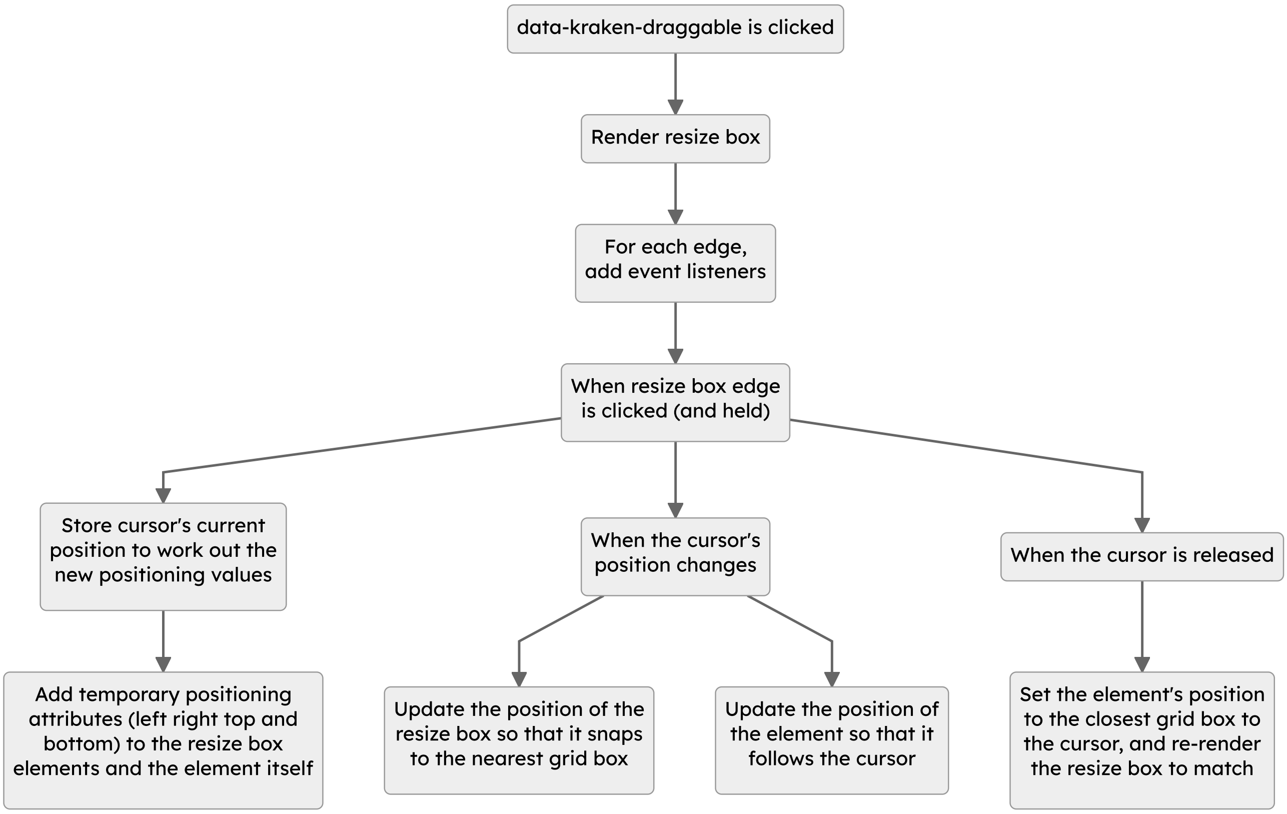
Resize box


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

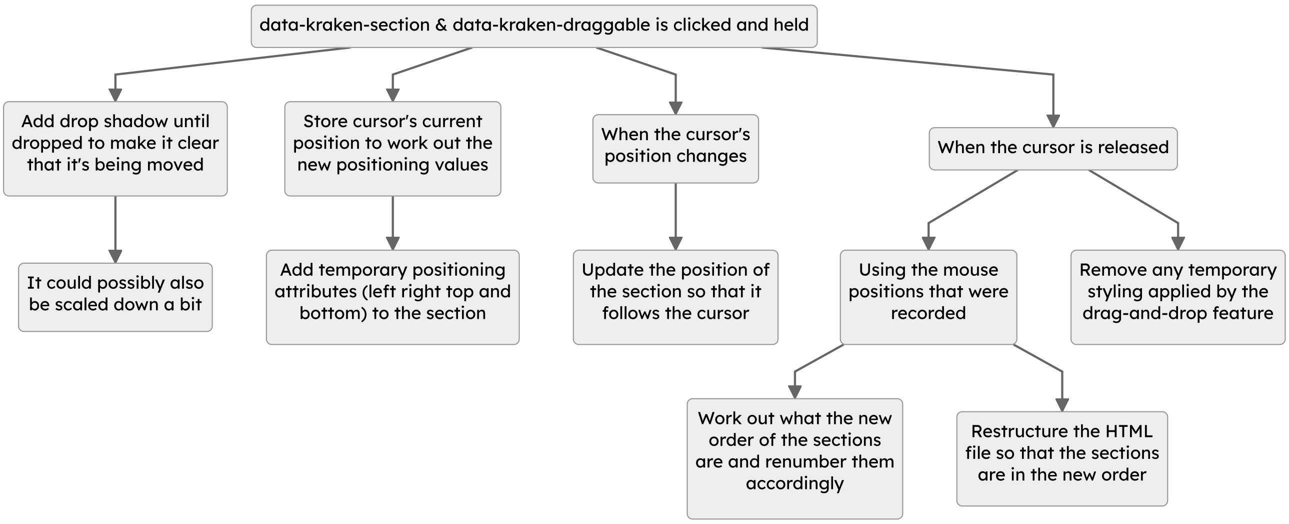

In the parent element of the selected element, add 12 elements with absolute positions
Using CSS, make four of them edges, and eight of them boxes
Using JS, position them so that they create a bounding box around the element


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

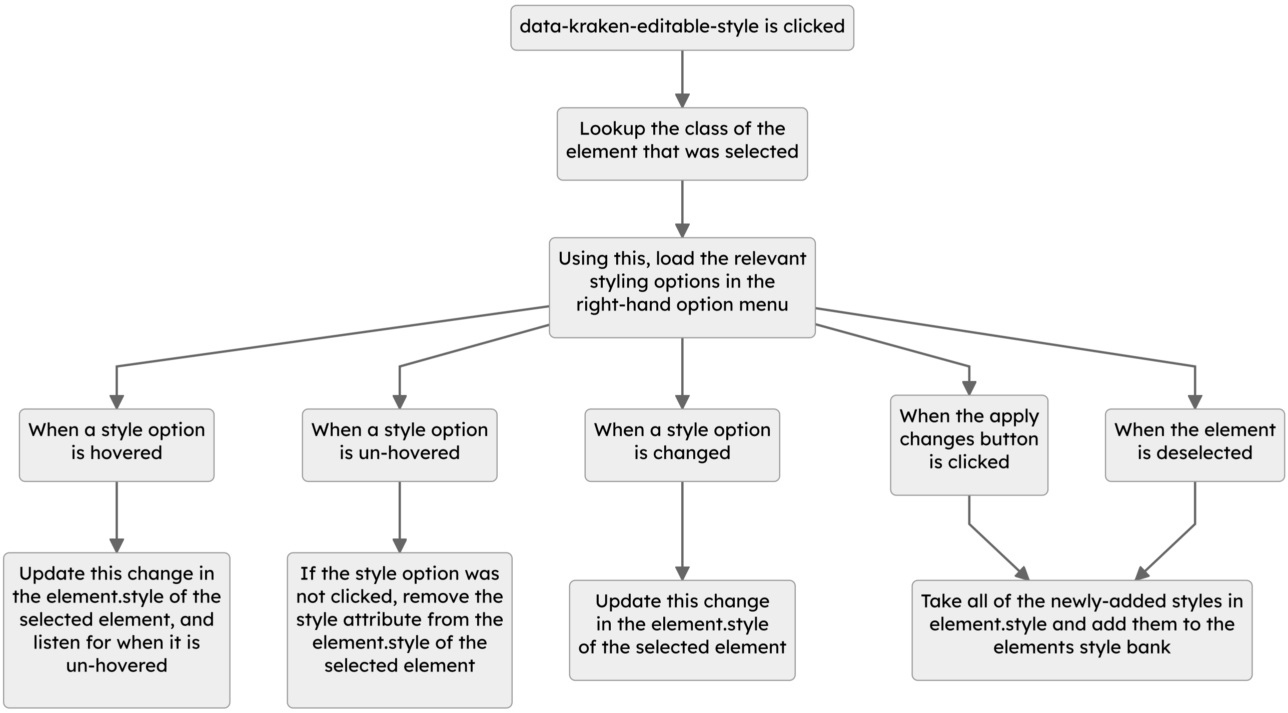
Dragging and dropping elements


data-kraken-draggable is clicked
Render resize box
For each edge, add event listeners
When resize box edge is clicked (and held)
Store cursor's current position to work out the new positioning values
Add temporary positioning attributes (left right top and bottom) to the resize box elements and the element itself
When the cursor's position changes
Update the position of the resize box so that it snaps to the nearest grid box
Update the position of the element so that it follows the cursor
When the cursor is released
Set the element's position to the closest grid box to the cursor, and re-render the resize box to match

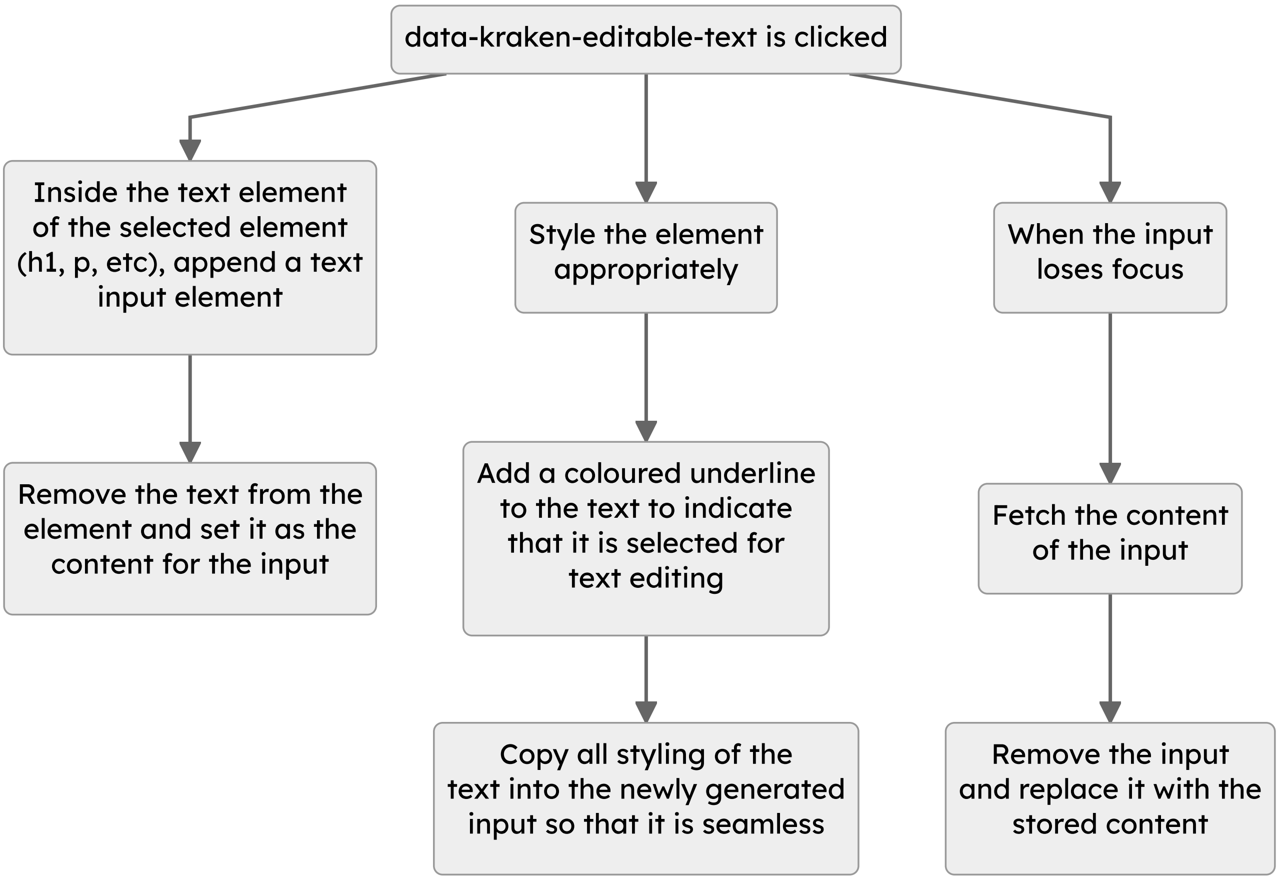
Dragging and dropping sections


data-kraken-section & data-kraken-draggable is clicked and held
Add drop shadow until dropped to make it clear that it's being moved
It could possibly also be scaled down a bit
Store cursor's current position to work out the new positioning values
Add temporary positioning attributes (left right top and bottom) to the section
When the cursor's position changes
Update the position of the section so that it follows the cursor
When the cursor is released
Using the mouse positions that were recorded
Work out what the new order of the sections are and renumber them accordingly
Restructure the HTML file so that the sections are in the new order
Remove any temporary styling applied by the drag-and-drop feature

Displaying element and section options


data-kraken-editable-style is clicked
Lookup the class of the element that was selected
Using this, load the relevant styling options in the right-hand option menu
When a style option is hovered
When a style option is un-hovered
When a style option is changed
When the apply changes button is clicked
When the element is deselected
Update this change in the element.style of the selected element, and listen for when it is un-hovered
If the style option was not clicked, remove the style attribute from the element.style of the selected element
Update this change in the element.style of the selected element
Take all of the newly-added styles in element.style and add them to the elements style bank

Displaying text editors


data-kraken-editable-text is clicked
Inside the text element of the selected element (h1, p, etc), append a text input element
Style the element appropriately
When the input loses focus
Remove the text from the element and set it as the content for the input
Add a coloured underline to the text to indicate that it is selected for text editing
Fetch the content of the input
Copy all styling of the text into the newly generated input so that it is seamless
Remove the input and replace it with the stored content

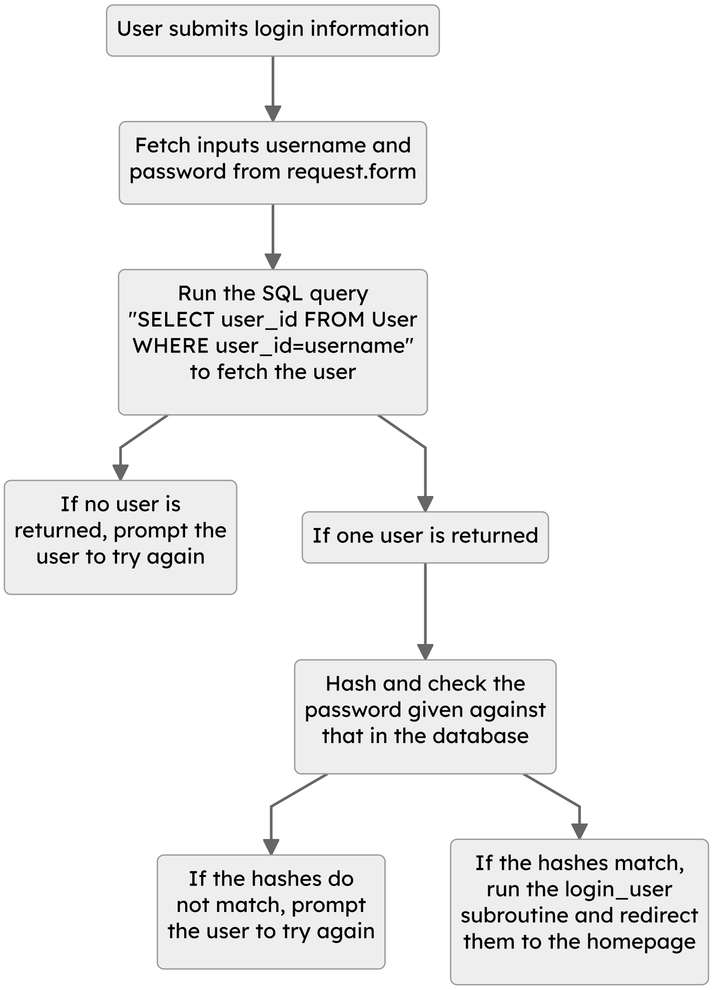
Multi-user system algorithms

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.

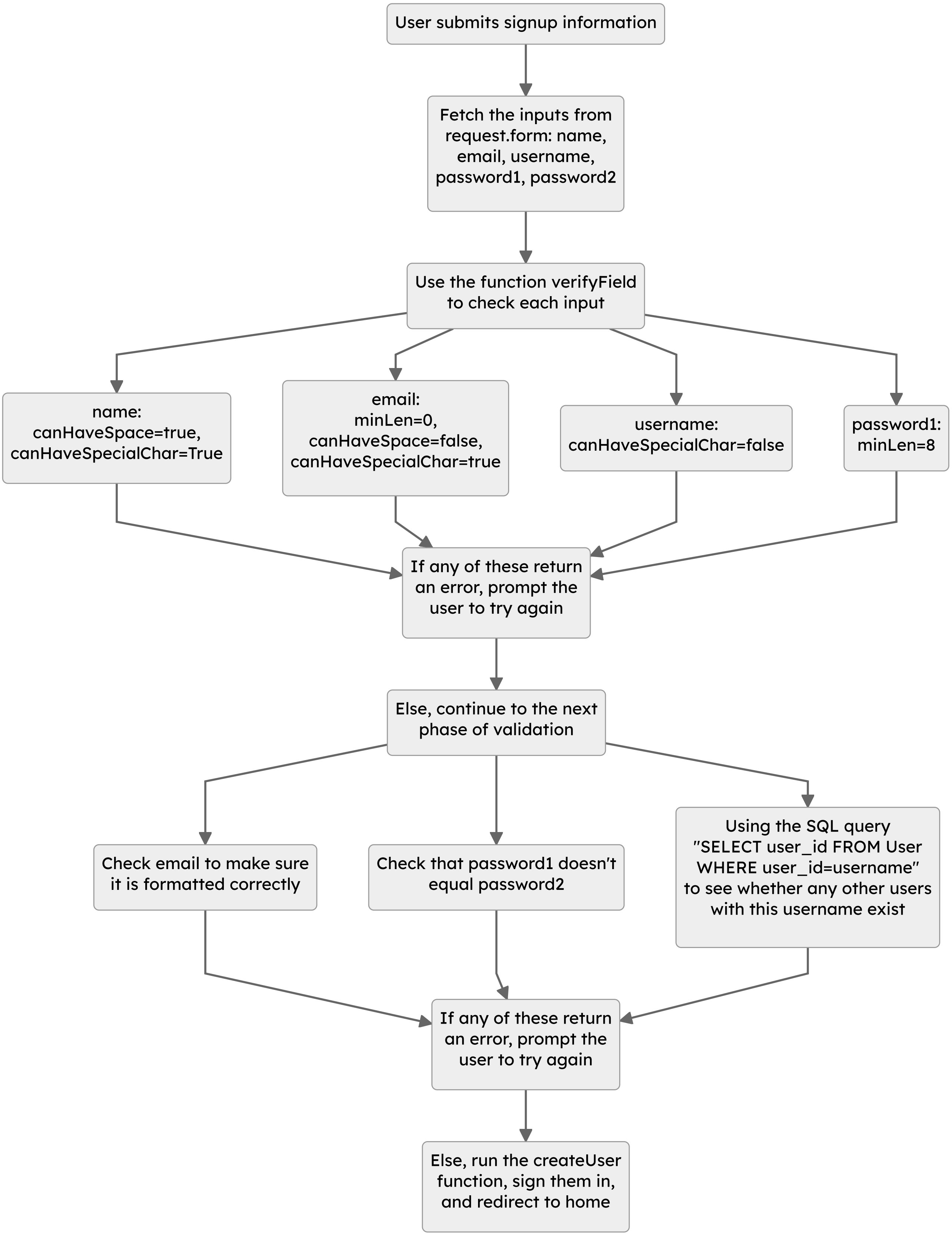
The main things that the multi-user system should be able to do are:

- 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

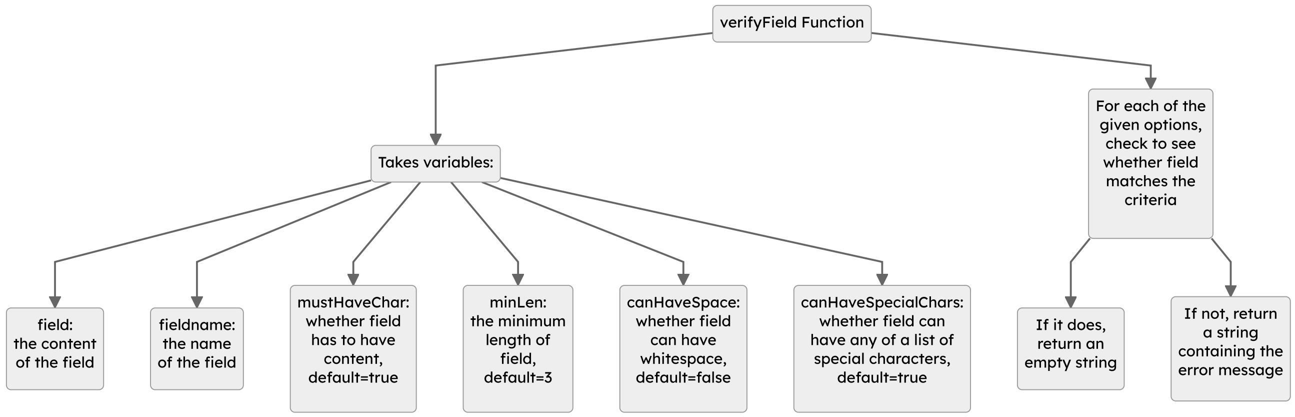
Login page


User submits login information
Fetch inputs username and password from request.form
Run the SQL query "SELECT user_id FROM User WHERE user_id=username" to fetch the user
If no user is returned, prompt the user to try again
If one user is returned
Hash and check the password given against that in the database
If the hashes do not match, prompt the user to try again
If the hashes match, run the login_user subroutine and redirect them to the homepage

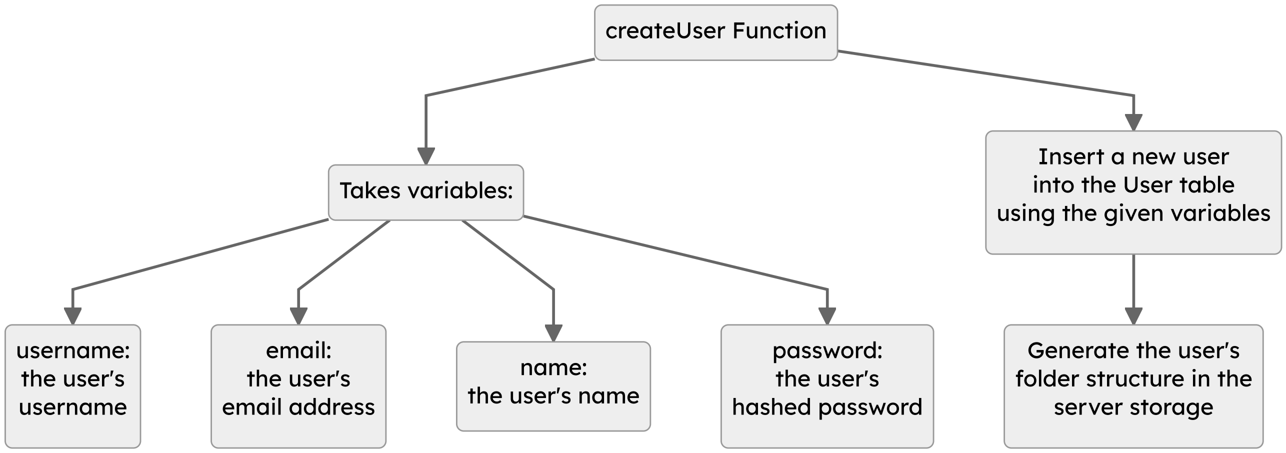
Signup page


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

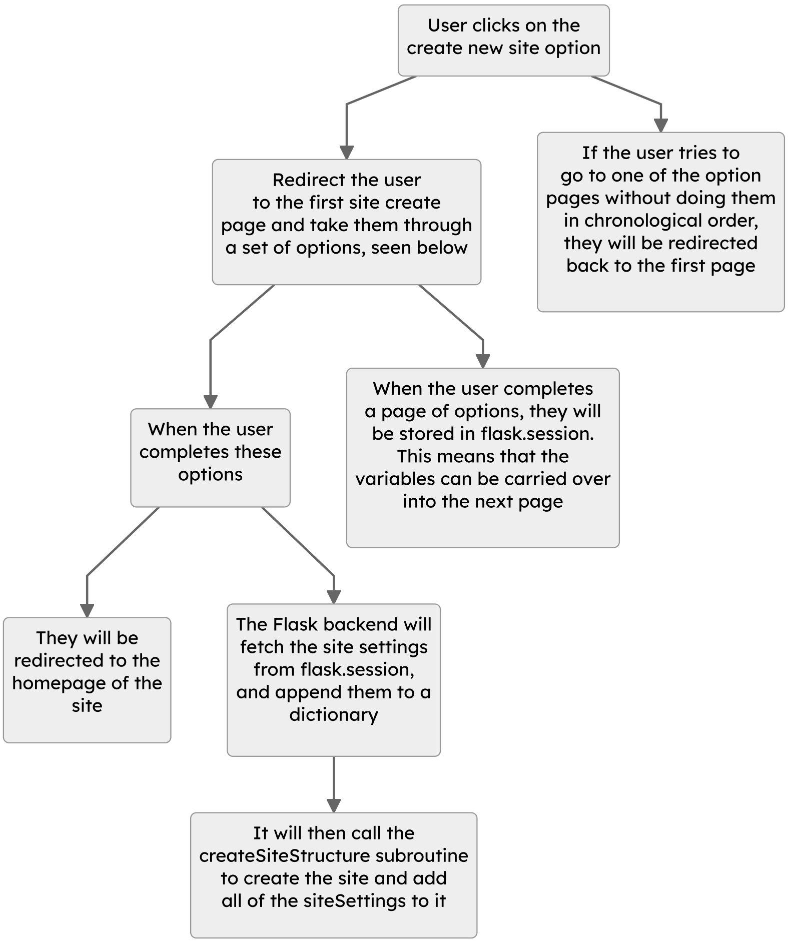
Field verification


verifyField Function
Takes variables:
field: the content of the field
fieldname: the name of the field
mustHaveChar: whether field has to have content, default=true
minLen: the minimum length of field, default=3
canHaveSpace: whether field can have whitespace, default=false
canHaveSpecialChars: whether field can have any of a list of special characters, default=true
For each of the given options, check to see whether field matches the criteria
If it does, return an empty string
If not, return a string containing the error message

Creating a new user


createUser Function
Takes variables:
username: the user's username
email: the user's email address
name: the user's name
password: the user's hashed password
Insert a new user into the User table using the given variables
Generate the user's folder structure in the server storage

Creating a new site



User clicks on the create new site option

Redirect the user to the first site create page and take them through a set of options, seen below

If the user tries to go to one of the option pages without doing them in chronological order, they will be redirected back to the first page

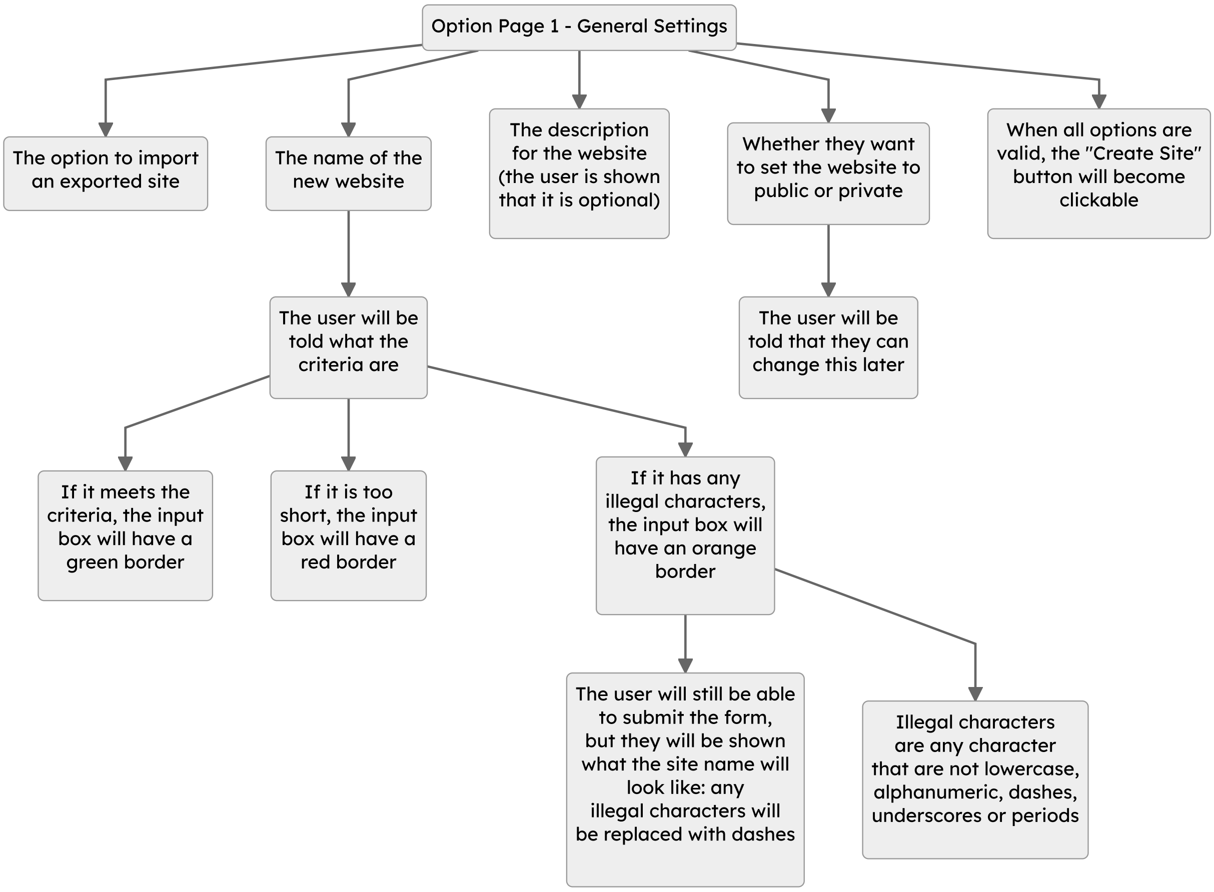
When the user completes these options

When the user completes a page of options, they will be stored in flask.session. This means that the variables can be carried over into the next page

They will be redirected to the homepage of the site

The Flask backend will fetch the site settings from flask.session, and append them to a dictionary

It will then call the createSiteStructure subroutine to create the site and add all of the siteSettings to it



Option Page 1 - General Settings

The option to import an exported site

The name of the new website

The description for the website (the user is shown that it is optional)

Whether they want to set the website to public or private

When all options are valid, the "Create Site" button will become clickable

The user will be told what the criteria are

The user will be told that they can change this later

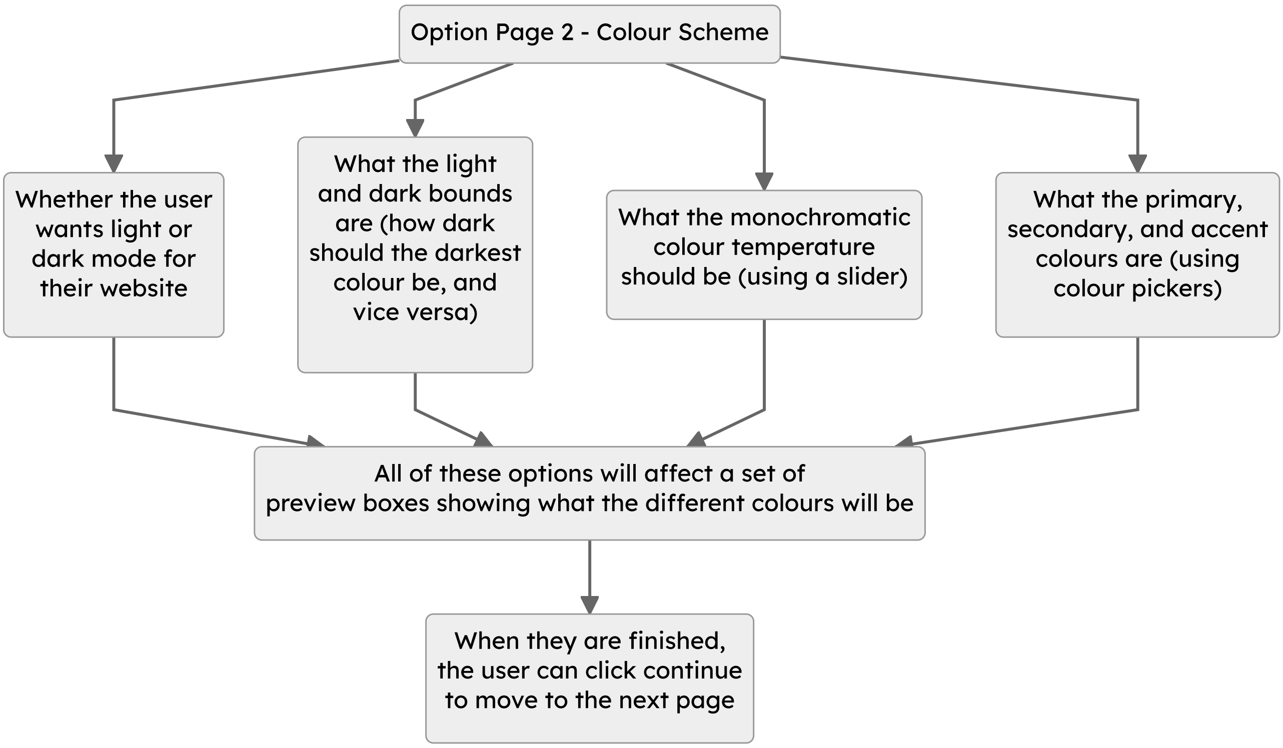
If it meets the criteria, the input box will have a green border

If it is too short, the input box will have a red border

If it has any illegal characters, the input box will have an orange border

The user will still be able to submit the form, but they will be shown what the site name will look like: any illegal characters will be replaced with dashes

Illegal characters are any character that are not lowercase, alphanumeric, dashes, underscores or periods


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

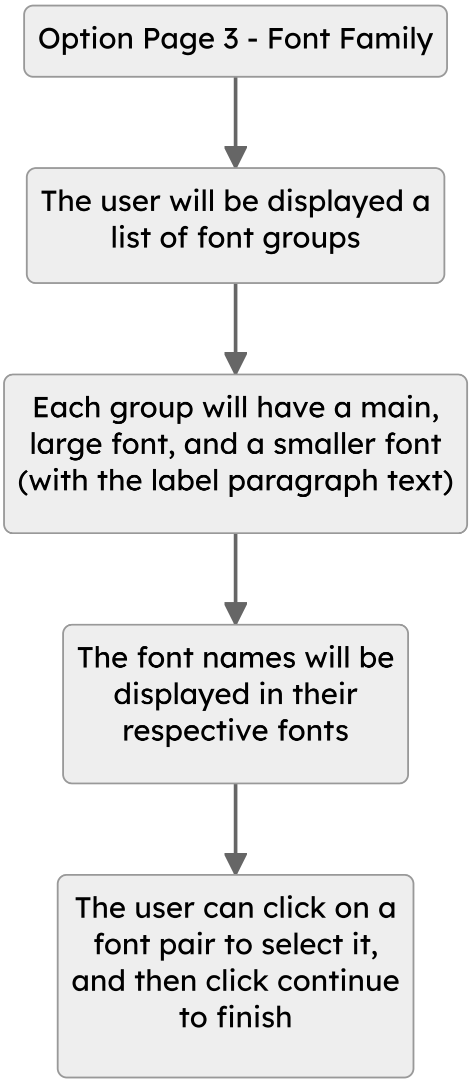

Option Page 3 - Font Family
The user will be displayed a list of font groups
Each group will have a main, large font, and a smaller font (with the label paragraph text)
The font names will be displayed in their respective fonts
The user can click on a font pair to select it, and then click continue to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

Diagram showing how the subroutines link

Subroutines

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

Multi-user system - login system

auth_login_post

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

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

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

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

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

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

auth_signup_post

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

verifyField

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

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

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

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

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

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

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

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

  return ""
}
```

createUser

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

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

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

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

  // create all the required folders
  generateFolderStructure(folderStructure)

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

Multi-user system - creating a new site

First page submit button subroutine

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

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

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

Website name formatting subroutines

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

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

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

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

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

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

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

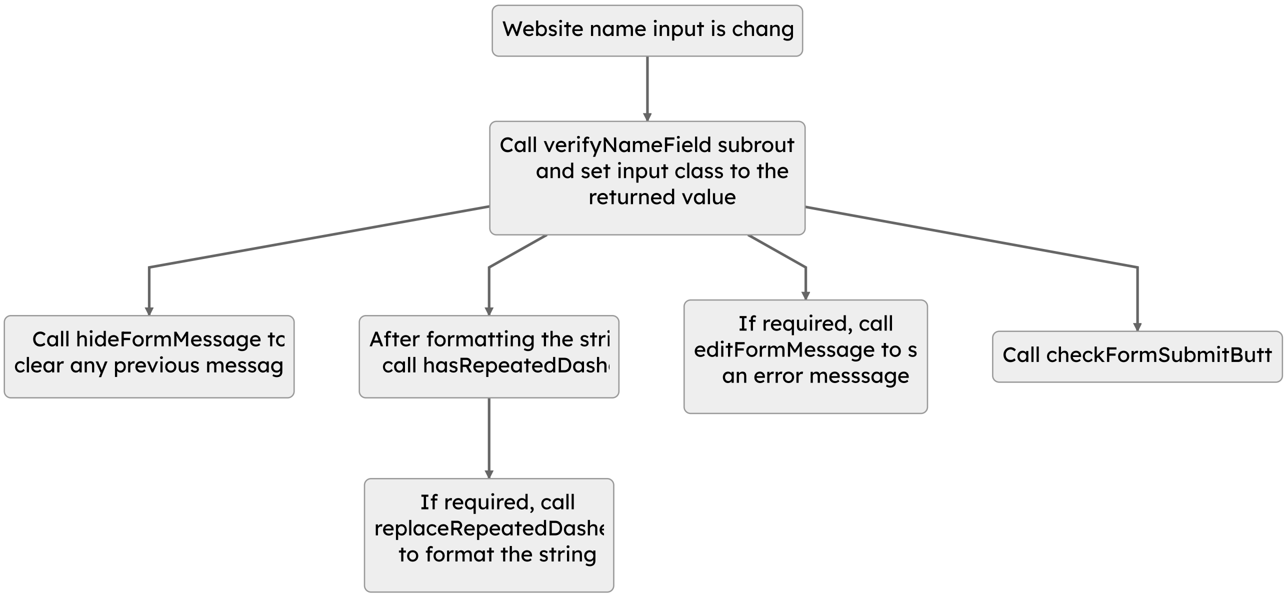
```
function hasRepeatedDashes(val) {
  // checks whether a string has repeated dashes in it
  for (i in range(val.length))
    if (val[i] == "-" && val[i+1] == "-")
      return true
  return false
}
```

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

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

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

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

  messageSpan.innerHTML=val
}
```

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

  val = forminput_websiteName.value;

  hideFormMessage()

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

  // must include at least one of the given characters

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

  if check return "danger"

  sitenames = request.db.query(
    f"SELECT sitename from SITE where userid={session.user.id}")
  if (val in sitenames) {
    messageSpan.innerHTML= "A site with this name already exists!"
    return "danger"
  }

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

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

  hideFormMessage()
  return "success"
}
```

Diagram showing how these subroutines link


Website name input is chang
Call verifyNameField subrout and set input class to the returned value
Call hideFormMessage tc clear any previous messag
After formatting the stri call hasRepeatedDash
If required, call editFormMessage to s an error messsage
Call checkFormSubmitButt
If required, call replaceRepeatedDashe to format the string

Website colour palette selection subroutines

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

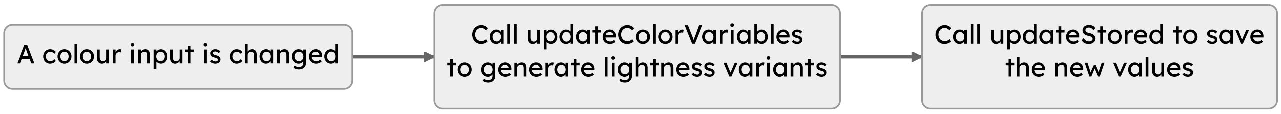
The `updateStored` subroutine will take the content of the `userSelectedColors` data and append it to a HTML element (`colorOutputSpan` in the JS code) inside a `<form>`, so that it can be sent to the server. This is the only way I've been able to send JavaScript-generated information to the server,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

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

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

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

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

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

  }

  updateStored()
}
```

Diagram showing how these subroutines link


A colour input is changed
Call updateColorVariables to generate lightness variants
Call updateStored to save the new values

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.

```python
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.

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

Explanation and justification of this process

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

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

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

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

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

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

Inputs and Outputs

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

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

Key variables

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

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

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

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

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

Validation

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

Login and signup forms

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

Website Name when creating a new site

The website name must meet these specific requirements:

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

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

Importing an exported site

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

Uploading data to the CMS

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

Data Sanitization

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

Testing method

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

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

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

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