SAIB Frontend Documentation

Empowering your Digital Presence

2025-01-21

Ivanne Dave Bayer

Table of Contents

Abstract

This document serves as a comprehensive guide to the company's front-end development standards, designed to onboard new front-end developers effectively. By drawing on insights and examples from past projects, it offers practical guidance to help developers excel in their roles.

The primary goal of this document is to deepen readers' understanding of front-end development fundamentals. Through detailed discussions of implementation practices, it provides a strong foundation for tackling tasks with confidence and supporting professional growth.

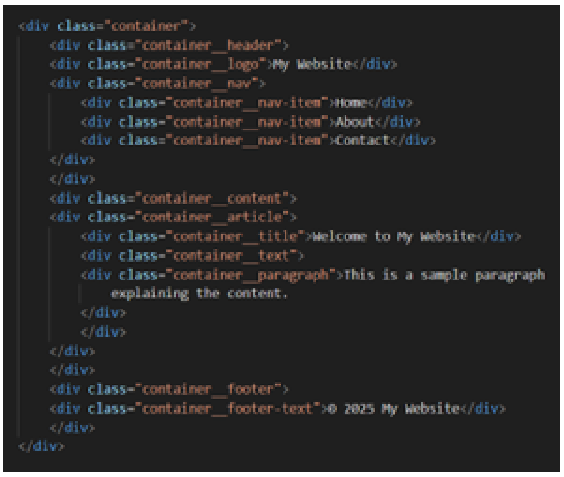
Before engaging in hands-on activities, it is essential to familiarize yourself with the core guidelines for writing high-quality, maintainable code. While new frontend employees of SAIB are expected to understand front-end technologies such as **HTML, CSS, JavaScript, Typescript, React, and Node.js.** this documentation focuses on the principles of organizing these elements into clean, well-structured, and scalable codebases.

By adhering to these standards, developers can ensure their work is both efficient and aligned with the company's commitment to excellence.

I) HTML Guidelines

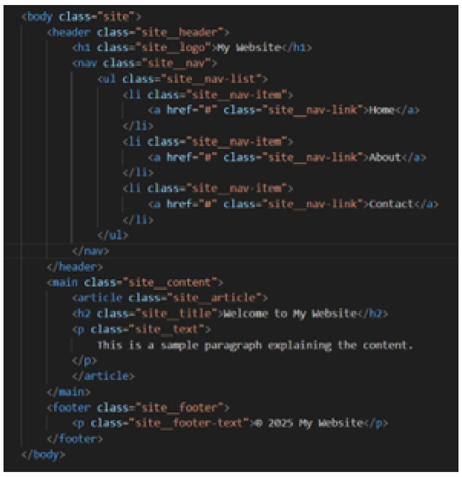
I.1) Overview

One of the most common challenges for new front-end developers is overlooking the importance of clean semantics when writing quality code. This issue often manifests as excessive nesting of `<div>` elements, resulting in obfuscated code that lacks clear meaning and purpose. Such practices make it difficult for other developers to understand the intent behind each implementation, which hinders productive maintainability during the process.



```html
<div class="container">
    <div class="container__header">
    <div class="container__logo">My Website</div>
    <div class="container__nav">
        <div class="container__nav-item">Home</div>
        <div class="container__nav-item">About</div>
        <div class="container__nav-item">Contact</div>
    </div>
    </div>
    <div class="container__content">
    <div class="container__article">
        <div class="container__title">Welcome to My Website</div>
        <div class="container__text">
        <div class="container__paragraph">This is a sample paragraph
            explaining the content.
        </div>
        </div>
    </div>
    </div>
    <div class="container__footer">
    <div class="container__footer-text">© 2025 My Website</div>
    </div>
</div>
```

Figure 1: Not recommended HTML tags structure



```html
<body class="site">
    <header class="site__header">
        <h1 class="site__logo">My Website</h1>
        <nav class="site__nav">
            <ul class="site__nav-list">
                <li class="site__nav-item">
                    <a href="#" class="site__nav-link">Home</a>
                </li>
                <li class="site__nav-item">
                    <a href="#" class="site__nav-link">About</a>
                </li>
                <li class="site__nav-item">
                    <a href="#" class="site__nav-link">Contact</a>
                </li>
            </ul>
        </nav>
    </header>
    <main class="site__content">
        <article class="site__article">
        <h2 class="site__title">Welcome to My Website</h2>
        <p class="site__text">
            This is a sample paragraph explaining the content.
        </p>
        </article>
    </main>
    <footer class="site__footer">
        <p class="site__footer-text">© 2025 My Website</p>
    </footer>
</body>
```

Figure 2: Recommended HTML tags structure

The images above show the recommended HTML styling in which front-end developers are advised to implement. I will also be shortly discussing what tags to use, their description, and when to use each.

I.2) HTML Tags

Below is a list of the commonly used HTML tags, along with short descriptions of the semantic behind them. For more information on other HTML tags and when to use them, you can refer to this website.

HTML Tags	Description
`<html>`	The root element of an HTML document that wraps all other elements.
`<head>`	Contains metadata, links to stylesheets, scripts, and other non-visual elements for the document.
`<body>`	Encloses the visible and interactive content of the webpage.
`<div>`	A generic container for grouping and styling elements; avoid overuse in favor of semantic tags.
`<main>`	Denotes the main content area of the webpage, where the primary information is displayed.
`<header>`	Contains introductory content like page titles, logos, or navigation links.
`<footer>`	Holds footer information like copyright, related links, or contact details.
`<nav>`	Groups navigational links for site navigation.
`<section>`	Represents a thematic grouping of related content, usually with a heading.
`<article>`	A self-contained piece of content like a blog post, comment, or news article.
`<aside>`	Defines content related to the main content, like sidebars or pull quotes.
`<h1>`–`<h6>`	Headings, ranked by importance from `<h1>` (most important) to `<h6>` (least important).
`<p>`	Defines a paragraph of text.
`<code>`	Displays a snippet of code in a monospaced font.
`<span>`	An inline container for applying styles or grouping content without semantic meaning.
`<ul>`	Creates an unordered list (bulleted).
`<ol>`	Creates an ordered list (numbered).
`<li>`	Represents an item within a list (`<ul>` or `<ol>`).
`<table>`	Defines a table.
`<thead>`, `<tbody>`, `<tfoot>`	Group table rows into header, body, and footer sections.
`<tr>`, `<td>`, `<th>`	Represent table rows, data cells, and header cells.
`<img>`	Embeds an image. Requires the src attribute for the image path and alt for alternative text.
`<form>`	Creates a form for user input.
`<input>`	Accepts various types of user input, such as text, passwords, or check-boxes.

HTML Tags	**Description**
`<textarea>`	Defines a multi-line text input area.
`<button>`	Represents a clickable button.
`<a>`	Creates hyperlinks for navigation.

II) Open Graph

II.1) Overview

The Open Graph protocol allows any web page to transform into a rich object within a social graph, enhancing how content is represented when shared across platforms like Facebook, Twitter, or Discord.

In this section of the documentation, we will use the Crashr website as a reference to explore how the Open Graph protocol is implemented. This will include examining metadata integration, common practices, and tips for optimizing content previews for social sharing.

II.2) Basic Metadata

To implement the Open Graph protocol, <`meta`> tags must be placed within the <`head`> section of a web page. These tags define how your content is represented within the social graph. The essential Open Graph meta tags include:

- **og:title:** Specifies the title of your object as it should appear in the graph. For example: “The Rock”.
- **og:type:** Indicates the type of your object, such as “video.movie”. Depending on the type specified, additional properties may be required.
- **og:image:** Provides the URL of an image that represents your object within the graph.
- **og:url:** Defines the canonical URL of your object, serving as its permanent ID within the graph. For example: https://www.crashr.io/.

In addition to these basic tags, the Open Graph protocol supports many other metadata options for customizing your content’s appearance in the social graph. For a full list of available metadata properties and their usage, refer to the official Open Graph documentation.

II.3) Checking Open Graph Implementation

To verify the Open Graph metadata of a website, developers can use tools like OpenGraph.xyz. Simply navigate to the website and input the desired URL to analyze its Open Graph implementation. This tool provides a quick and visual way to ensure your metadata is correctly set up and displays as intended when shared on social platforms. Below is an image which shows how to check for open graph implementation.

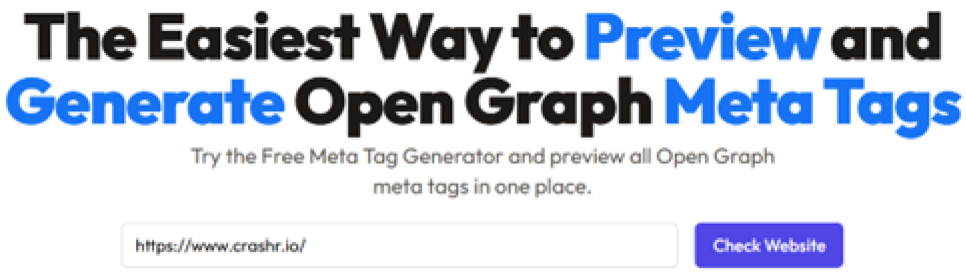
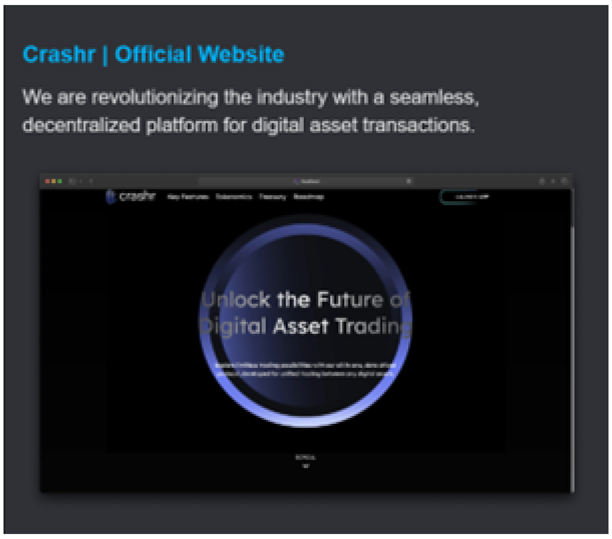

The Easiest Way to Preview and Generate Open Graph Meta Tags
Try the Free Meta Tag Generator and preview all Open Graph meta tags in one place.
https://www.crashr.io/
Check Website

Figure 3: Search interface of open graph website

After checking a website's Open Graph implementation using a tool like [OpenGraph.xyz](), the tool displays the relevant metadata details used for social sharing. For this documentation, we will reference the Crashr website's metadata as an example. A copy of its metadata tags for various social platforms will also be provided for better understanding. Below are examples of the metadata tags associated with specific platforms:

- **HTML Meta Tags**

```
<title>Crashr | Official Website</title>
<meta name="description" content="We are revolutionizing the industry with
a seamless, decentralized platform for digital asset transactions.">
<meta property="og:url" content="https://www.crashr.io/">
<meta property="og:type" content="website">
<meta property="og:title" content="Crashr | Official Website">
<meta property="og:description" content="We are revolutionizing the industry
with a seamless, decentralized platform for digital asset transactions.">
<meta  property="og:image"  content="https://www.crashr.io/crashr-website.
webp">
```


Crashr | Official Website

We are revolutionizing the industry with a seamless, decentralized platform for digital asset transactions.

Figure 4: Open graph result of the crashr website

- **Twitter Meta Tags**

```
<meta name="twitter:card" content="summary_large_image">
<meta property="twitter:domain" content="crashr.io">
<meta property="twitter:url" content="https://www.crashr.io/">
<meta name="twitter:title" content="Crashr | Official Website">
<meta name="twitter:description" content="We are revolutionizing the
industry with a seamless, decentralized platform for digital asset
transactions.">
<meta name="twitter:image" content="https://www.crashr.io/crashr-website.
webp">
```

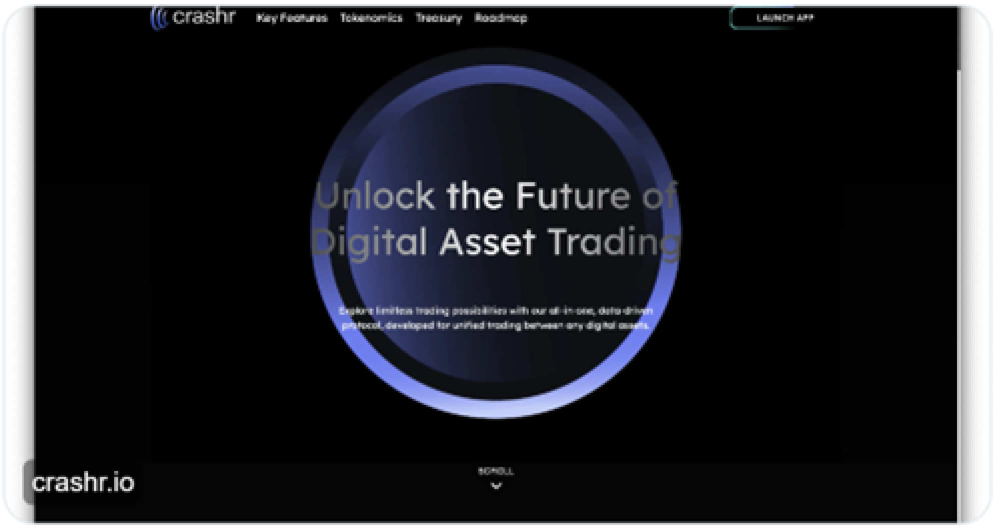

crashr
Key Features
Tokenomics
Treasury
Roadmap
LAUNCH APP
Unlock the Future of Digital Asset Trading
Explore limitless trading possibilities with our all-in-one, data-driven protocol, developed for unified trading between any digital assets.
SCROLL
crashr.io

Figure 5: Open graph result of the crashr website in twitter

III) CSS Guidelines

III.1) Overview

In this section, I will focus on key CSS concepts and essential elements that are crucial for understanding the fundamentals of front-end development. This section is intended as a refresher, so readers are expected to have a basic understanding of CSS fundamentals.

III.2) CSS Naming Convention

It is essential to adopt a consistent and clear naming convention when writing CSS to ensure code maintainability and readability. Naming conventions may vary depending on company standards, but it is essential to follow a specific convention for consistency. One popular and effective naming methodology is **BEM (Block, Element, Modifier)**.

BEM helps create reusable and modular components by structuring CSS classes into three main parts:

- **Block:** Represents a standalone entity that is meaningful on its own. For example, a navigation bar (nav) or a button (button).
  Ex.
  ```
  .button {}
  .menu {}
  ```
- **Element:** Represents a part of a block that has no standalone meaning and is dependent on the block. Elements are separated from the block name by a double underscore (__).
  Ex.
  ```
  .button__icon {}
  .menu__item {}
  ```
- **Modifier:** Represents a different state or version of a block or element. Modifiers are separated from the block or element name by a double hyphen (–).
  Ex.
  ```
  .button--primary {}
  .menu__item--active {}
  ```

For more information regarding this naming convention, readers may refer to this article on BEM.

III.3) CSS Box Model

In a nutshell, the box model is essentially a box that wraps around an element. It consists of content, padding, borders and margins. Below is an image that illustrates this:

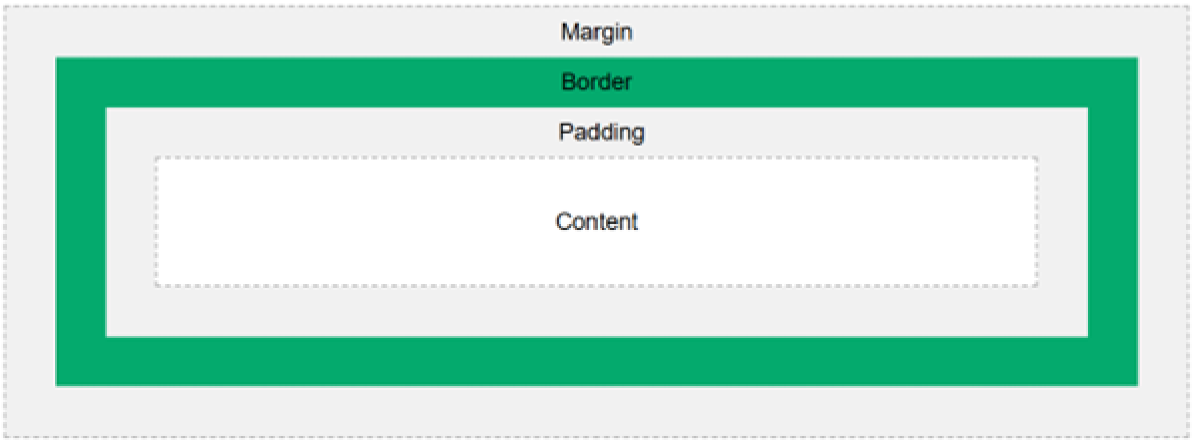


Figure 6: Illustration on the structure which defines the box model

Each section is defined by the following:

- **Content:** The content of the box, where text and images appear
- **Padding:** Clears an area around the content.
- **Border:** A border that goes around the padding and content
- **Margin:** Clears an area outside the border.

The image below illustrates how box models are interpreted based on the AEGIS website, which serves as a reference. Each numbered section represents an element used to construct the webpage:

- **Background Section:** Number 1 represents the Aegis homepage background, which contains the navigation and content area. Number 8 represents another box model that contains an SVG as its background image. While the box may appear curved due to the SVG design, it is inherently a standard rectangular box model, as the SVG itself introduces the curves visually.
- **Navigation Bar:** Number 2 represents the navigation bar, which contains three child elements (3, 4, and 5). These elements together form the navigation section.
- **Content Area:** Number 6 is a distinct box model that contains the Aegis shield animation. Number 7 is part of a distinct box model that is centered on the webpage, which can be represented as an article.

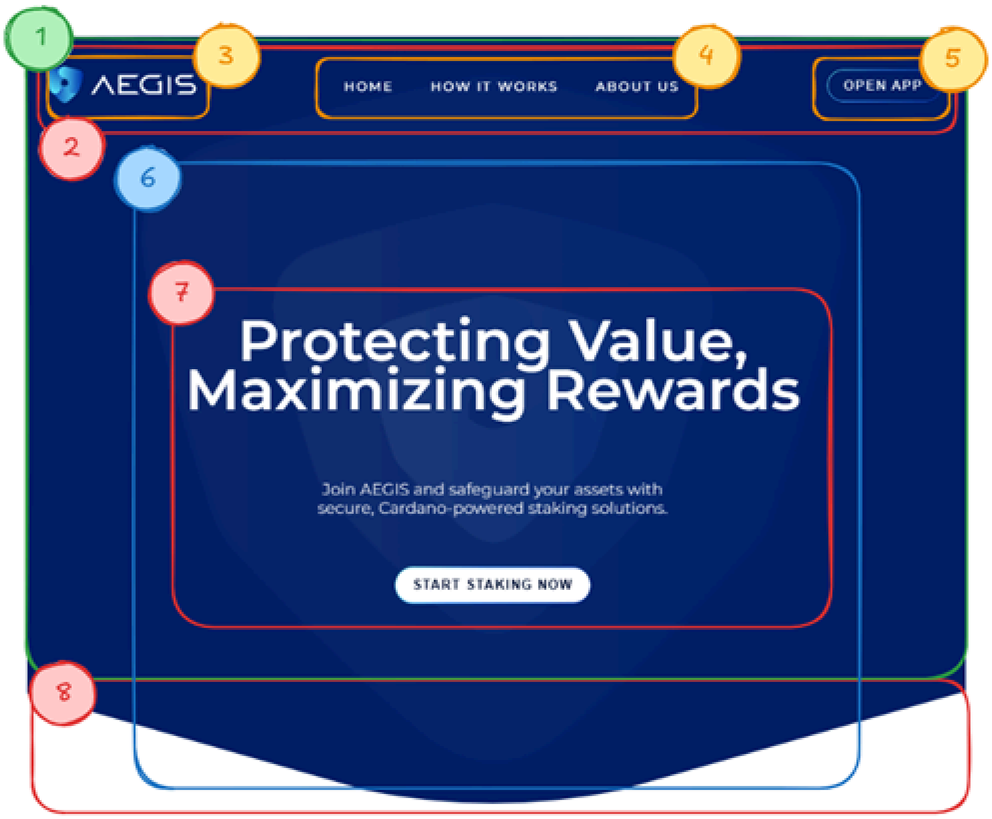

1
2
3
4
5
6
7
8
AEGIS
HOME
HOW IT WORKS
ABOUT US
OPEN APP
Protecting Value, Maximizing Rewards
Join AEGIS and safeguard your assets with secure, Cardano-powered staking solutions.
START STAKING NOW

Figure 7: CSS box model of the aegis first section from the homepage

III.4) CSS Reset

The goal of a CSS reset is to minimize browser inconsistencies in default styling, such as line heights, margins, font sizes for headings, and other built-in properties that vary across browsers. These differences can make it difficult for developers to accurately position elements, as they may not know which default styles are being applied.

For example, the default margin for an `h1` tag is typically set to `margin-y: 0.67em`. This can interfere with consistent layout and positioning. By using a reset stylesheet, you can remove or standardize these default styles. More information regarding default attributes for every element can be found in this link.

```css
* {
  font-family: "Montserrat", sans-serif;
  line-height: 1.5;
  font-weight: 400;
  margin: 0;
  padding: 0;
  box-sizing: inherit;
}

html,
body {
  box-sizing: border-box;
  overflow-x: hidden;
  padding-right: 0;
}
```

III.5) CSS Display

In CSS, the display property determines how elements are rendered on a page. Each HTML element has a default display value, which influences how it behaves in relation to other elements. For example, a `<div>` is a block-level element by default, meaning it occupies the full width of its container and begins on a new line, while a `<p>` tag is inline by default, meaning it only takes up as much width as its content and does not cause line breaks.

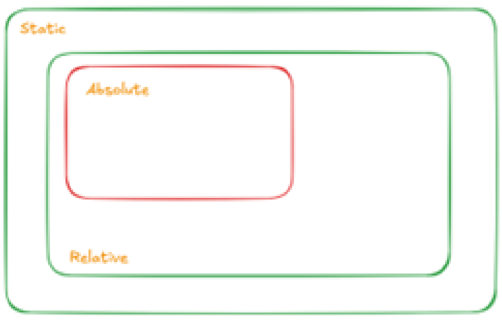
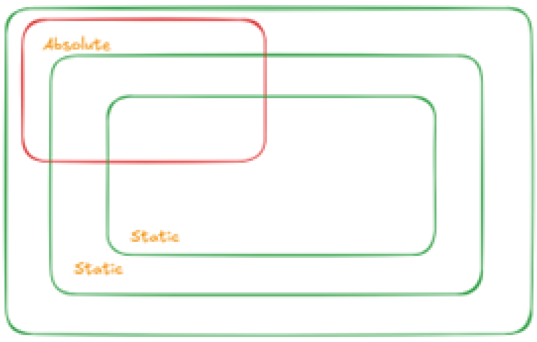
Display Properties	Description
None	The element is completely removed from the document layout. It takes up no space and is not visible. Child elements are also not displayed.
Block	Block-level elements take up the full width of their container (unless otherwise specified) and cause line breaks before and after the element. Common block-level elements include `<div>`, `<section>`, and `<article>`.
Inline	Inline elements only take up as much width as their content and do not cause line breaks. They sit alongside other inline elements. Examples include `<span>`, `<a>`, and `<strong>`.

Display Properties	Description
Inline-Block	Combines characteristics of both block and inline. The element behaves like an inline element (it does not cause line breaks) but can also have width, height, padding, and margins applied like a block element.
Flex	Enables a flexible container that distributes space between and aligns items along both horizontal (row) and vertical (column) axes. Flexbox is ideal for responsive layouts where the alignment of elements needs to adapt based on the container's size.
Grid	Creates a grid layout system with rows and columns. It allows for precise control over the placement of elements within a grid container, making it suitable for complex layouts.

III.6) CSS Positioning

This section focuses on the five positioning properties essential for controlling the placement of elements on a webpage. Each property type is discussed in detail, with its functionality and use cases explained. For more information, refer to this article.

Positioning Properties	Description
Static	HTML elements are positioned as static by default. They follow the normal flow of the page and cannot be positioned using the top, bottom, left, or right properties.
Relative	Elements with relative positioning remain in the normal document flow but are positioned relative to their original position. They can be adjusted using the top, bottom, left, and right properties.
Absolute	Elements with absolute positioning are positioned relative to their nearest positioned ancestor (an element that is not static). They are removed from the normal flow, allowing them to overlap with other elements. If no such ancestor exists, the element is positioned relative to the document (root element). This can be referenced based on the image which compares both:

Positioning Properties	Description
	 Figure 8: Absolute element with relative ancestor  Figure 9: Absolute element without positioned ancestor
Fixed	Is positioned relative to the viewport, meaning it stays in the same place even when the page is scrolled. The top, right, bottom, and left properties can be used to position the element.
Sticky	Is positioned based on the user's scroll position.

III.7) CSS Units

They are used to define the lengths and dimensions of elements. They can be categorized into absolute and relative units.

- **Absolute Units:** They have a fixed length and do not depend on other elements or the viewport. They are commonly used for maintaining consistent sizes, such as for images or print layouts.

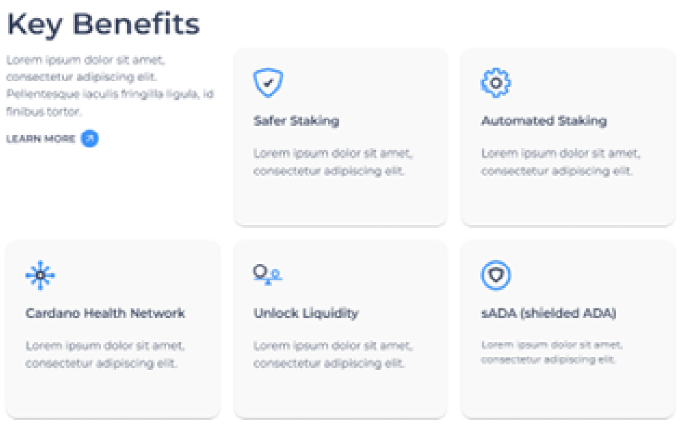
Absolute Units	Description
px	Pixels, a fixed unit commonly used for on-screen element dimensions.
cm	Centimeters, mainly used in print layouts.
mm	Millimeters, used primarily for print layouts.
in	Inches, equivalent to 2.54 cm, often used in print.
pt	Points, commonly used in print (1 pt = $\frac{1}{72}$ of an inch).
pc	Picas, used in print (1 pc = 12 pt).

- **Relative Units:** They define lengths relative to another property or element. They are ideal for responsive design as they adapt to the viewport and font size.

Relative Units	Description
%,	Relative to the parent element.
em	Relative to the font-size of the element (2em means 2 times the size of the current font).
rem	Relative to font-size of the root element.
vw	Relative to 1% of the width of the viewport.
vh	Relative to 1% of the height of the viewport.
vmin	Relative to 1% of viewport's smaller dimension.
vmax	Relative to 1% of viewport's larger dimension.
ch	Relative to the width of the character "0" of the current font.
ex	Relative to the height of the font's lowercase "x".

III.8) CSS Responsiveness

Webpages are viewed in various sizes depending on the user's device. Developers should ensure that the content adapts seamlessly to all screen sizes, providing a consistent and accessible experience across devices. The image below illustrates how a webpage should appear on different devices:



Key Benefits

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

LEARN MORE

Safer Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Automated Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Cardano Health Network

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

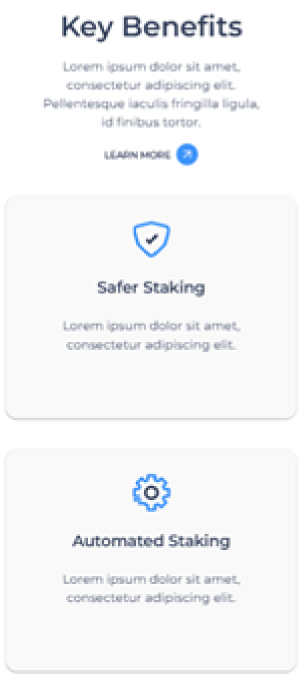
Unlock Liquidity

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

sADA (shielded ADA)

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Figure 10: Desktop view of key benefits section



Key Benefits

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

LEARN MORE

Safer Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Automated Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Figure 11: Mobile view of key benefits section

III.9) Media Queries

These are CSS rules that help achieve responsiveness on a website. They use the `@media` rule to apply a block of CSS properties only when a specified condition is met. For more in-depth details on how they work, readers can refer to this article.

In the context of responsive design, media queries are used to define breakpoints, which determine how elements should adapt based on the screen size. To implement this, developers often follow the **mobile-first principle** (tailwindcss also uses this principle), designing for mobile devices first and then adding styles for larger screens.

Below is an example of how media queries are applied, (the code used here is referenced from the Aegis homepage shown above):

Desktop

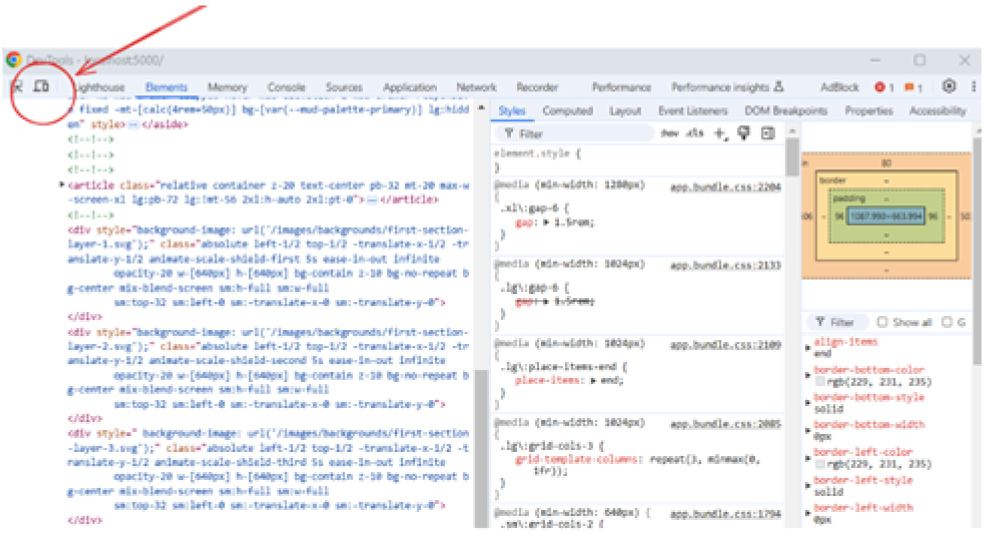
```css
@media screen and (min-width: 1280px)
{
  .keyBenefits__container {
    display: flex;
    flex-direction: row;
    flex-wrap: wrap;
    justify-content: space-between;
  }
}
```

Mobile

```css
@media screen and (min-width: 480px)
{
  .keyBenefits__container {
    display: flex;
    flex-direction: column;
  }
}
```

Developer tools in chrome are also available which contains a setting for checking responsiveness on specific devices. Here are the following steps to achieve this:

1. Right click on the web page and select **inspect**.
2. In the Developer Tools window, click on the **Toggle** device toolbar button (the small phone/ tablet icon) located at the top-left of the Developer Tools pane.



```
 fixed -mt-[calc(4rem+50px)] bg-[var(--mud-palette-primary)] lg:hidd
en" style>…</aside>
<!--!-->
<!--!-->
<!--!-->
<article class="relative container z-20 text-center pb-32 mt-20 max-w
-screen-xl lg:pb-72 lg:!mt-56 2xl:h-auto 2xl:pt-0">…</article>
<!--!-->
<div style="background-image: url('/images/backgrounds/first-section-
layer-1.svg');" class="absolute left-1/2 top-1/2 -translate-x-1/2 -tr
anslate-y-1/2 animate-scale-shield-first 5s ease-in-out infinite
        opacity-20 w-[640px] h-[640px] bg-contain z-10 bg-no-repeat b
g-center mix-blend-screen sm:h-full sm:w-full
        sm:top-32 sm:left-0 sm:-translate-x-0 sm:-translate-y-0">
</div>
<div style="background-image: url('/images/backgrounds/first-section-
layer-2.svg');" class="absolute left-1/2 top-1/2 -translate-x-1/2 -tr
anslate-y-1/2 animate-scale-shield-second 5s ease-in-out infinite
        opacity-20 w-[640px] h-[640px] bg-contain z-10 bg-no-repeat b
g-center mix-blend-screen sm:h-full sm:w-full
        sm:top-32 sm:left-0 sm:-translate-x-0 sm:-translate-y-0">
</div>
<div style=" background-image: url('/images/backgrounds/first-section
-layer-3.svg');" class="absolute left-1/2 top-1/2 -translate-x-1/2 -t
ranslate-y-1/2 animate-scale-shield-third 5s ease-in-out infinite
        opacity-20 w-[640px] h-[640px] bg-contain z-10 bg-no-repeat b
g-center mix-blend-screen sm:h-full sm:w-full
        sm:top-32 sm:left-0 sm:-translate-x-0 sm:-translate-y-0">
</div>
```

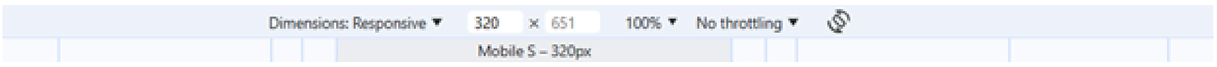
```
element.style {
}
@media (min-width: 1280px)          app.bundle.css:2204
{
  .xl\:gap-6 {
    gap: ▶ 1.5rem;
  }
}
@media (min-width: 1024px)          app.bundle.css:2133
{
  .lg\:gap-6 {
    ~~gap: ▶ 1.5rem;~~
  }
}
@media (min-width: 1024px)          app.bundle.css:2109
{
  .lg\:place-items-end {
    place-items: ▶ end;
  }
}
@media (min-width: 1024px)          app.bundle.css:2085
{
  .lg\:grid-cols-3 {
    grid-template-columns: repeat(3, minmax(0,
        1fr));
  }
}
@media (min-width: 640px) {         app.bundle.css:1794
  .sm\:grid-cols-2 {
```

```
▶ align-items
  end
▶ border-bottom-color
  rgb(229, 231, 235)
▶ border-bottom-style
  solid
▶ border-bottom-width
  0px
▶ border-left-color
  rgb(229, 231, 235)
▶ border-left-style
  solid
▶ border-left-width
  0px
```

Figure 12: Dev tools interface

3. In the device toolbar, you'll see a dropdown that lets you select a device type (e.g., iPhone, iPad, Nexus, etc.). In relation to this, you can also enter custom device dimensions by selecting Responsive in the dropdown and adjusting the width and height manually.

.



Dimensions: Responsive ▾ 320 × 651 100% ▾ No throttling ▾

Mobile S – 320px

Figure 13: Dev tools responsive dropdown

III.10) CSS Pseudo Elements

Pseudo-elements are used to style specific parts of an element or insert additional content without modifying the HTML. The two most common pseudo-elements are `::before` and `::after`.

- **`::before`**: It is used to insert content before the actual content of an element. You can add styles to this generated content.
- **`::after`**: It is used to insert content after the actual content of an element. Similarly, you can style the inserted content.

Both `::before` and `::after` require the **content** property to function, even if it's just an empty string (`content: "";`).

Below is a button from the Crashr website developed by SAIB. To achieve a gradient border, using pseudo-elements can be an alternative solution to address this issue effectively. Here is a link of the codepen on how this feature was implemented.

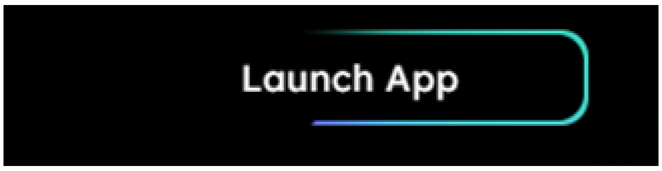


Figure 14: Crashr button with rotating border gradient

IV) Stacking Context

A stacking context is a conceptual layer in the browser where elements are rendered in a specific order. It determines how elements visually overlap on the webpage. HTML elements occupy this space in priority order based on element attributes.

In simpler terms:

1. Stacking contexts can be contained in other stacking contexts and together create a hierarchy of stacking contexts.
2. Each stacking context is completely independent of its siblings: only descendant elements are considered when stacking is processed.
3. Each stacking context is self-contained: after the element's contents are stacked, the whole element is considered in the stacking order of the parent stacking context.

Below is an image which shows how every positioned element creates its own stacking context, because of their positioning and z-index values. For more information, developers can refer to this article.

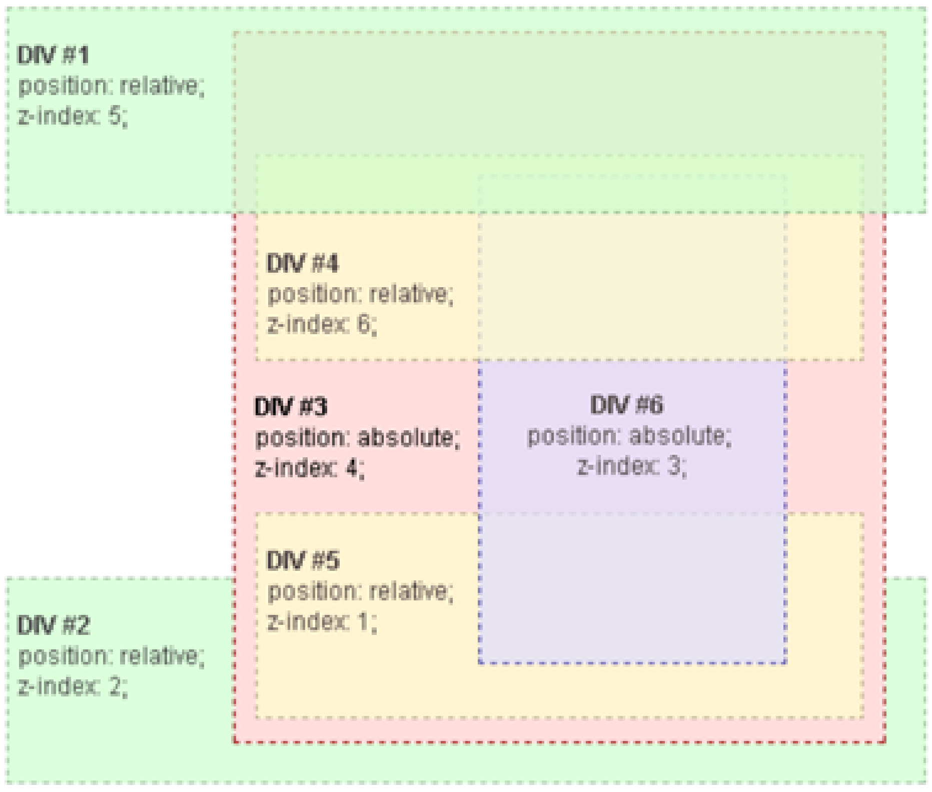


Figure 15: Layout representing stacking context

V) Javascript

V.1) Overview

Javascript is a versatile, prototype-based programming language that supports multiple paradigms, including object-oriented, imperative, declarative, and functional programming. It operates as a single-threaded, dynamic language, making it an essential tool for modern web development. A solid understanding of JavaScript is crucial for developers to grasp how it works under the hood, enabling them to write efficient and effective code. In this section, I will explore key concepts that are vital for understanding it as a language. For an in-depth overview of the language, developers can refer to this documentation.

V.2) Prerequisites

Before proceeding with this activity, it's essential to have a solid understanding of the fundamental properties of HTML and CSS.

V.3) DOM Manipulation

Before diving into DOM manipulation, it's crucial to understand what the Document Object Model (DOM) actually represents. Every document loaded in the browser is structured as a DOM tree, which is a hierarchical representation of the HTML document. The browser creates this structure, allowing programming languages—such as JavaScript—to access, modify, and interact with HTML elements dynamically. The DOM is also used by the browser itself to apply CSS styles, attributes, and other information to the correct elements when rendering a page. Once a page is loaded, developers can manipulate the DOM with JavaScript to create interactive experiences. In reference to this, an example is provided below:

```html
<html lang="en-US">
  <head>
    <title>SAIB</title>
  </head>
  <body>
    <h1>Cardano</h1>
  </body>
</html>
```

The DOM tree structure for the above HTML document looks like this:

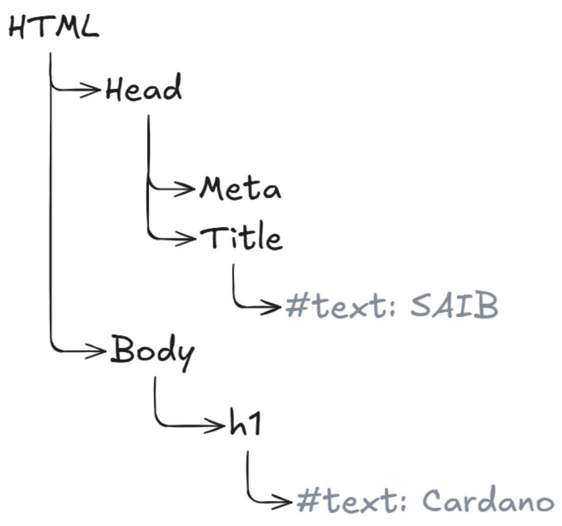


Figure 16: DOM interpretation of the HTML code

Each entry in this tree is called a node, and some nodes represent HTML elements such as: `<html>`, `<head>`, `<meta>`, `<title>`, `<body>`. Additionally, the `#text` nodes represent the actual content inside the elements, such as text within `<title>` or `<h1>`.

In the context of DOM manipulation, we interact with these nodes programmatically to modify the structure, style, or content of a webpage. Below, we will demonstrate how to change the styling of an element in the DOM and perform an operation such as deleting a button using JavaScript. For this part we will use the SAIB navigation bar as an example. Initially, the navigation bar contains a button, and we will manipulate the DOM to change its style and later remove it entirely. Below is a visual representation of the SAIB navigation bar before any DOM manipulation.

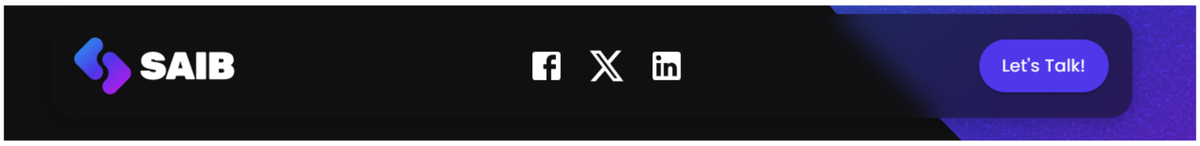
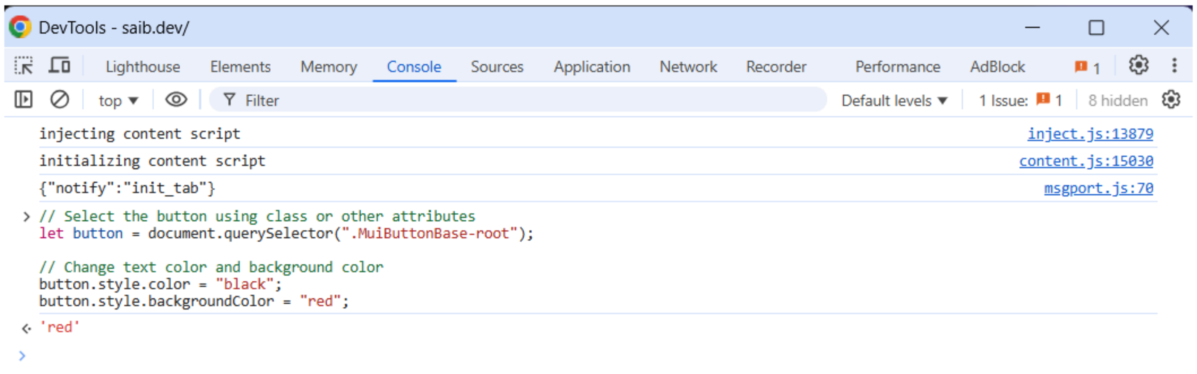


Figure 17: Navigation bar of SAIB website

To begin, we access the Developer Tools in the browser by pressing F12 or right-clicking and selecting "Inspect", which gives us the ability to interact with the DOM through the Console tab. This allows us to execute JavaScript that manipulates the page dynamically:



```
injecting content script                                    inject.js:13879
initializing content script                                 content.js:15030
{"notify":"init_tab"}                                       msgport.js:70
> // Select the button using class or other attributes
  let button = document.querySelector(".MuiButtonBase-root");

  // Change text color and background color
  button.style.color = "black";
  button.style.backgroundColor = "red";
< 'red'
>
```

Figure 18: SAIB developer tools

The code above achieves the following:

- `let button = document.querySelector(".MuiButtonBase-root")`: This method selects the first element in the DOM.
- `button.style.color = 'black'`: Modifies the text color of the button to black.
- `button.style.backgroundColor = 'red'`: Changes the button's background color to red.

This JavaScript code is executed directly in the browser's Console section, affecting the appearance of the button on the page. After running the code, the appearance of the button will update as shown in the following image:

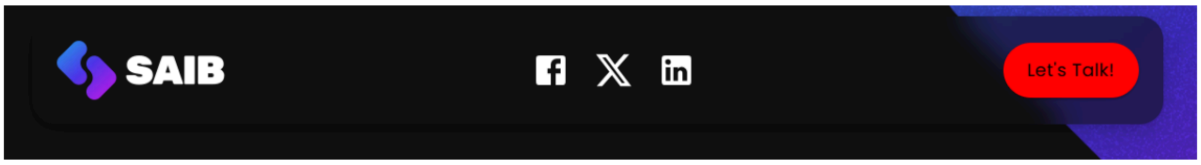


Figure 19: Navigation bar of SAIB website after DOM manipulation

Additionally, if we wish to remove the button entirely, we can use the following command: `button.remove()`. This command targets the selected button element and removes it from the DOM, making it disappear from the rendered page.

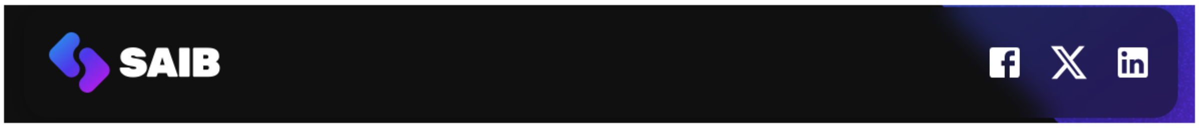


Figure 20: Navigation bar of SAIB website after deleting button

V.4) Synchronous Programming

This refers to a sequential execution model where the program processes one task at a time, step by step. Each task must be completed before moving on to the next, meaning the program waits for the current operation to finish before executing the next line of code. This approach is straightforward and predictable but can be inefficient if a task takes a long time to complete, as it blocks the program's progress.

```
console.log("Task 1: Start");
console.log("Task 2: Processing...");
console.log("Task 3: Done");
```

Based on the given example, each statement is executed one after the other, and no line proceeds until the current one completes. While this model is simple, it can lead to performance bottlenecks if a time-intensive task is executed synchronously which leads us to our next topic.

V.5) Asynchronous Programming

As discussed in the previous section, one of the major drawbacks of synchronous programming is how it handles time-intensive tasks. Such tasks can block the execution of the entire program. Consider the example below:

```
console.log("Task 1: Start Math Calculation");

// Simulating a time-intensive task (calculating the sum of large numbers)
function calculateSum() {
  let sum = 0;
  for (let i = 0; i < 1000000000; i++) {
    sum += i;
  }
  return sum;
}

calculateSum();
console.log("Task 2: Calculation Complete");
console.log("Task 3: User Can Proceed");
```

In this example, the `calculateSum` function simulates a time-intensive calculation by summing a large range of numbers. The entire program is blocked while the calculation is running, which means the user can't interact with the site during this time. To avoid blocking the UI, we can process the calculation asynchronously, allowing the application to remain responsive while performing the task:

```javascript
console.log("Task 1: Start Math Calculation");

// Simulating an asynchronous calculation using setTimeout
setTimeout(() => {
  let sum = 0;
  for (let i = 0; i < 1000000000; i++) {
    sum += i;
  }
  console.log("Task 2: Calculation Complete");
}, 0);

console.log("Task 3: User Can Proceed");
```

In this case, setTimeout allows the calculation to happen in the background. The application doesn't get blocked. Task 3 is logged immediately after Task 1, while the calculation continues in the background. Once the calculation is done, Task 2 is logged. This demonstrates how asynchronous programming can enhance the user experience by preventing the application from becoming unresponsive during time-intensive tasks like large mathematical calculations.

V.6) Promises

They are used to handle asynchronous operations in JavaScript. A promise represents a task that will eventually complete (either successfully or with an error). It allows you to work with asynchronous operations like data fetching, timers, or calculations without blocking the rest of the program. Here's a basic example using a promise to simulate an asynchronous task:

```
function delay(seconds) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Task completed after ${seconds} seconds`);
    }, seconds * 1000);
  });
}

delay(3)
  .then((message) => {
    console.log(message);
  })
  .catch((error) => {
    console.error(error);
  });

console.log("Task started...");
```

In this example, the delay function returns a promise that resolves after a 3-second delay. This simulates an asynchronous task where the program doesn't block and allows other code to run while waiting for the delay to complete.

- **delay(seconds):** This function returns a promise that simulates a delay by using `setTimeout`. After the specified number of seconds, it resolves with a message.
- **delay(3):** This simulates a 3-second delay before the promise resolves.
- **.then():** The `.then()` method is used to handle the result (message) once the promise is resolved.
- **.catch():** This is used to catch any potential errors in the program.

Eventually, this program executes the output:

```
Task started...
Task completed after 3 seconds
```

V.7) Async and Await

The `async` keyword is used to declare a function that returns a promise. This indicates that the function is asynchronous, and it enables the use of the `await` keyword inside that function to wait for a promise to resolve before proceeding with the rest of the program. This is demonstrated based on the given example below:

```
async function fetchData(){
  // Function for fetching data
}
```

The `await` keyword is used to wait for a promise to resolve or reject within an `async` function. When `await` is used, it pauses the execution of the function until the promise is resolved or rejected which can be demonstrated in the following code:

```
async function fetchData(){
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/posts",
    );
}
```

The `await` keyword waits for the `fetch()` promise to resolve. The `fetch()` function is used to retrieve data from the given URL. The result (a response object) is stored in the response variable once the promise is fulfilled.

Users can also handle the data returned from the promise by checking if the response is `OK` `(status code 200-299)`. If the response is not successful, the error can be handled appropriately.

```
async function fetchData() {
  const response = await fetch(
    "https://jsonplaceholder.typicode.com/posts",
  );

  if (!response.ok) {
    console.log(`HTTP error ${response.status}`);
  } else {
    const data = await response.json();
    console.log(data);
  }
}
```

After fetching the data, we check if the response is OK using response.ok. If it’s not, we log an error message with the status code. Otherwise, we parse the response as JSON using `await response.json()` and log the data.

To handle potential errors in asynchronous code, such as network issues or failed API requests, you can wrap your code in a `try...catch` block. This ensures that if an error occurs at any point in the asynchronous code, the error is caught and handled gracefully.

```
async function fetchData() {
  try {
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/posts",
    );

    if (!response.ok) {
      throw new Error(`HTTP error ${response.status}`);
    }

    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(`Could not get data: ${error.message}`);
  }
}
```

The code inside the `try` block will attempt to fetch the data and process the response. If an error occurs, whether it's a network issue or a failed response, it will throw an error with a custom message. The `catch` block will handle the error, logging a message that explains what went wrong.

With `async` and `await`, working with asynchronous code becomes much more readable and manageable, especially when dealing with multiple asynchronous operations like fetching data from APIs.

V.8) Event Loop

Before diving into event loop, it's essential to understand how JavaScript operates as a single-threaded language and the significance of the call stack. JavaScript executes one operation at a time in a sequential manner. It processes code line by line, and each function call is added to the call stack, which follows a Last In, First Out (LIFO) execution model. To visualize this concept, consider the following example:

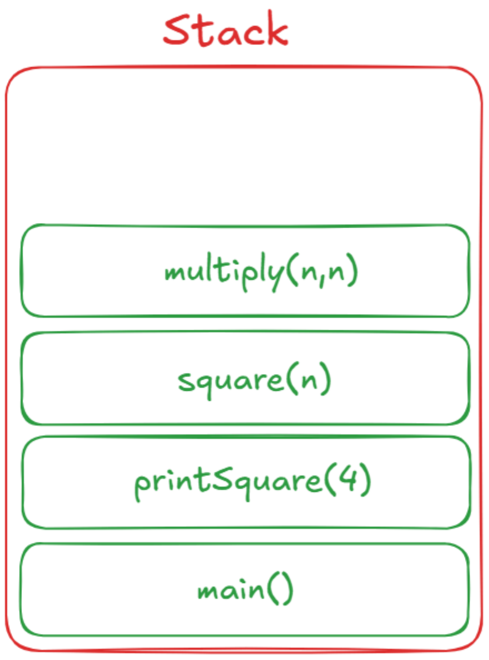
Call Stack

```
function multiply(a, b) {
  return a * b;
}

function square(n) {
  return multiply(n, n);
}

function printSquare(n) {
  var squared = square(n);
  console.log("square is printed");
}

printSquare(4);
```



1. The `main` function starts executing and is placed in the call stack.
2. The `printSquare(4)` function is invoked and pushed onto the stack.
3. Inside `printSquare`, the `square(4)` function is called and added to the stack.

4. Inside `square`, the `multiply(4, 4)` function is called and pushed onto the stack.
5. `multiply(4, 4)` executes, returning 16, and is removed from the stack.
6. `square(4)` receives the result (16) and is then removed from the stack.
7. `printSquare(4)` logs “square is printed” and is finally removed from the stack.
8. At this point, the call stack is empty, indicating that the program has completed execution.

Now that we understand the role of the call stack and how JavaScript processes function calls synchronously, the question arises:

💡 *How does JavaScript handle asynchronous operations like API calls, timers, and event listeners without blocking the execution of other code?*

This is where event loops come into play. JavaScript has a runtime model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. For this part, I will be using a simple example to demonstrate how this works under the hood:

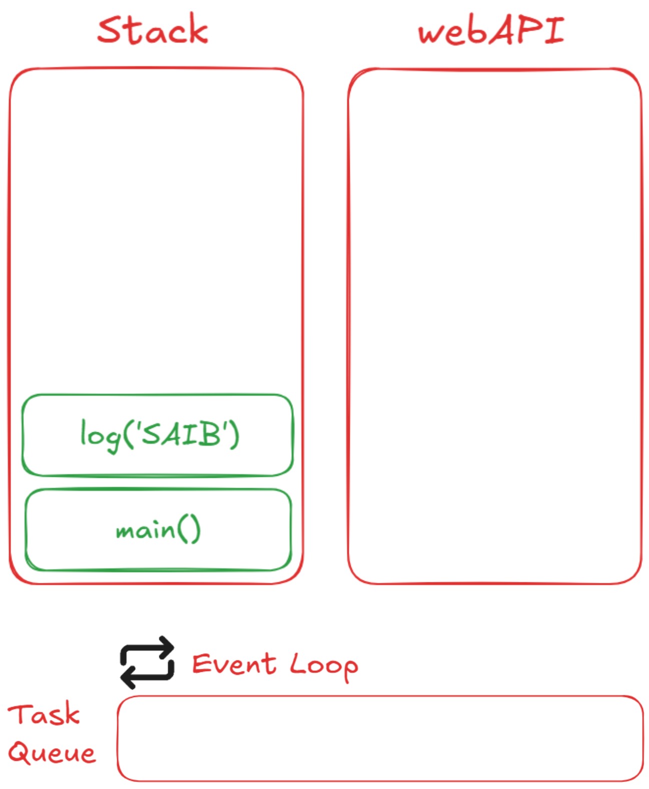
Event Loop

```
console.log("SAIB");

setTimeout(function cb() {
  console.log("Cardano");
}, 5000);

console.log("ADA");
```

```
//Output
SAIB
```



1. The `main` function starts executing and is pushed in the call stack.
2. `console.log("SAIB")` is pushed to the call stack and immediately prints `SAIB`, then it is removed from the stack.

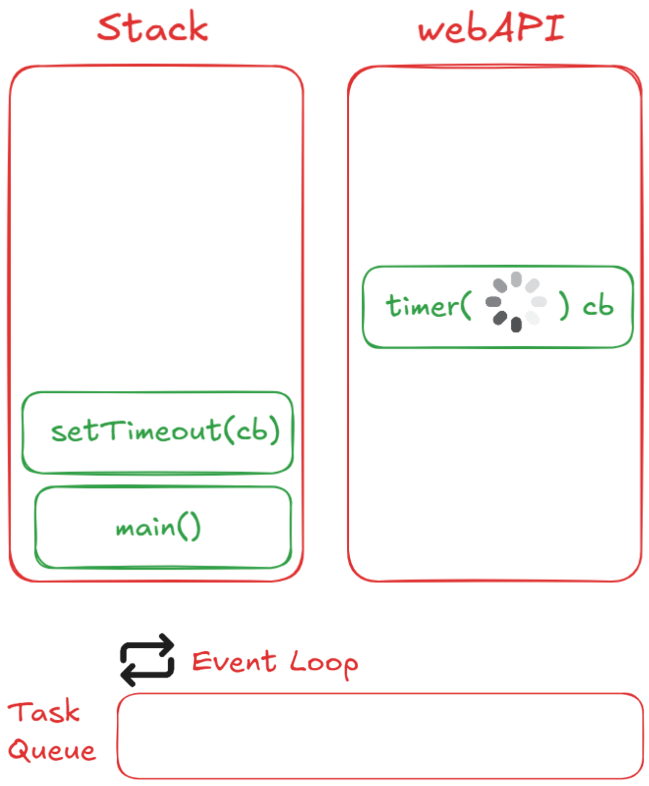
Event Loops

```
console.log("SAIB");

setTimeout(function cb() {
  console.log("Cardano");
}, 5000);

console.log("ADA");
```

```
//Output
SAIB
```



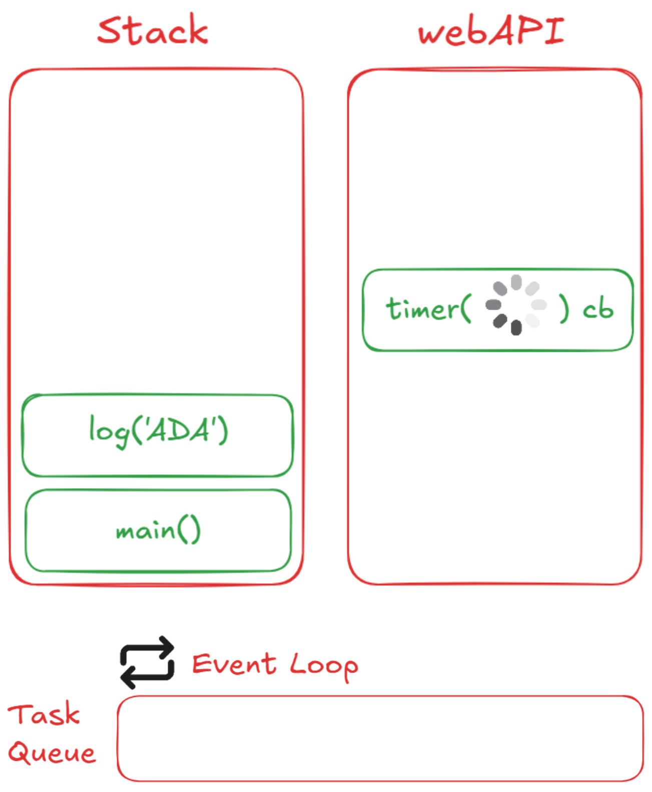
1. The `setTimeout(cb, 5000)` function is added to the call stack.
2. Since `setTimeout` is a Web API function, it is handed over to the Web API environment, which starts the 5-second timer.
3. The `setTimeout` function completes execution and is removed from the call stack.

Event Loops

```
console.log("SAIB");

setTimeout(function cb() {
  console.log("Cardano");
}, 5000);

console.log("ADA");
```



```
//Output
SAIB
ADA
```

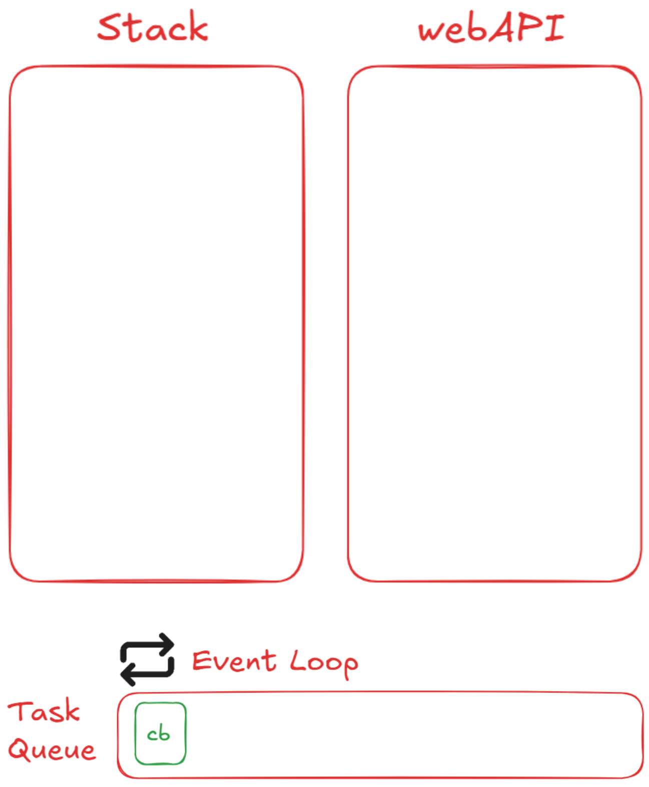
1. `console.log("ADA")` is added to the call stack, immediately prints `ADA`, and is then removed.
2. After 5 seconds, the Web API moves the callback function (cb) to the Task Queue
3. The Task Queue follows First In, First Out (FIFO) where the first completed asynchronous task gets executed first.

Event Loops

```
console.log("SAIB");

setTimeout(function cb() {
  console.log("Cardano");
}, 5000);

console.log("ADA");
```



```
//Output
SAIB
ADA
```

1. The event loop ensures non-blocking behavior, it only moves tasks from the queue when the call stack is empty.
2. The event loop continuously checks if the call stack is empty.
3. Once it confirms the call stack is empty, it moves the callback function from the task queue to the call stack.

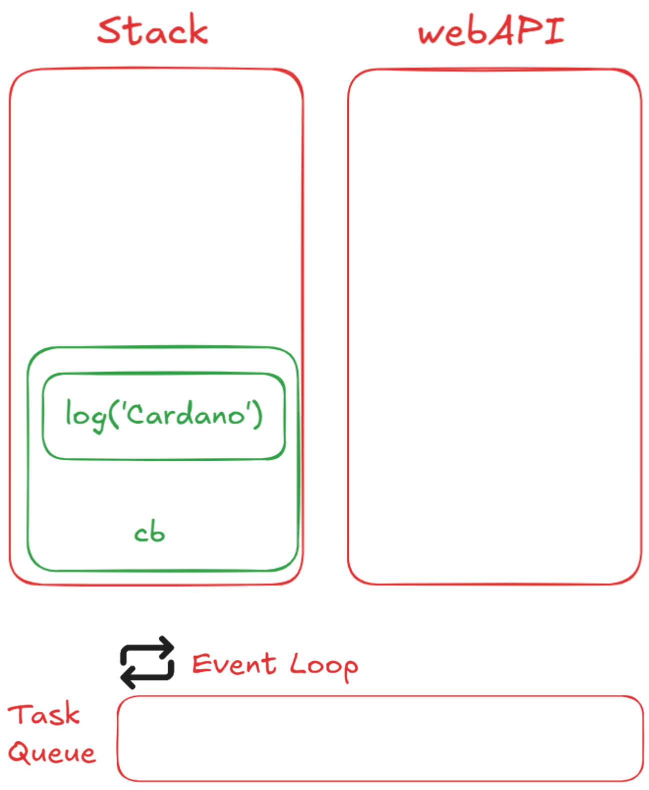
Event Loops

```
console.log("SAIB");

setTimeout(function cb() {
  console.log("Cardano");
}, 5000);

console.log("ADA");
```

```
//Output
SAIB
ADA
Cardano
```



1. The function cb() executes and reads the `console.log("ADA")`
2. The text `Cardano` becomes printed, and then is removed from the call stack.

V.9) Mastering Fundamentals

As SAIB developers, mastering the fundamentals of HTML, CSS, and JavaScript is crucial before diving into frontend frameworks like React or Blazor. These core technologies form the foundation of web development and enable you to troubleshoot, optimize, and understand how frameworks work under the hood. This section highlights a custom carousel created by Jonh Alexis using raw HTML, CSS, and JavaScript, demonstrating how these technologies work together to create interactive features. This carousel implementation can also be checked based on the codepen provided. Understanding such implementations will help deepen your knowledge of web fundamentals and provide a strong base for learning frameworks effectively.

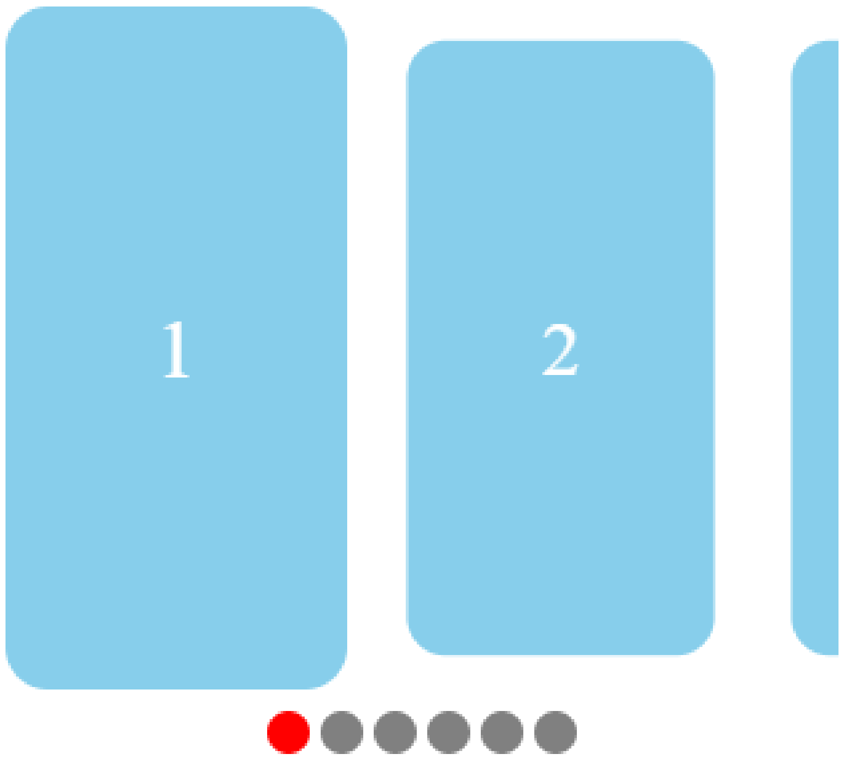


Figure 37: Custom Carousel with raw HTML, CSS, and Javascript

VI) Creating the AEGIS Website Without Flex and Grid

VI.1) Overview

This section is designed as a fun activity for developers to practice layout manipulation without using flexbox or grid, focusing instead on positioning techniques. The Aegis homepage serves as a reference for this exercise. Additionally, I will explain the various implementations I used to create the layouts, using Excalidraw for visual representation. If you're interested in exploring this repository, here's a link to the Aegis website implemented with standard HTML, CSS, and JavaScript.

VI.2) Prerequisites

Before proceeding with this activity, it's essential to have a solid understanding of the fundamental properties of CSS. This includes basic display attributes such as **block, inline,** and particularly **inline-block**. Additionally, a basic understanding of CSS positioning, including **relative, absolute,** and **static**, is necessary for successful implementation.

VI.3) Floats

Flexbox was introduced in 2009, followed by CSS **Grid** in 2011. This raises the question: how did developers create website layouts before these technologies, considering CSS was first developed in 1994 to define front-end UI specifications? One of the earliest methods for positioning elements was the use of **floats**, which allowed elements to be aligned to the left or right of their container.

The float property can have one of the following values:

Float Properties	Description
Left	The element floats to the left of its container.
Right	The element floats to the right of its container
None	The element does not float (will be displayed just where it occurs in the text). This is default
Inherit	The element inherits the float value of its parent.

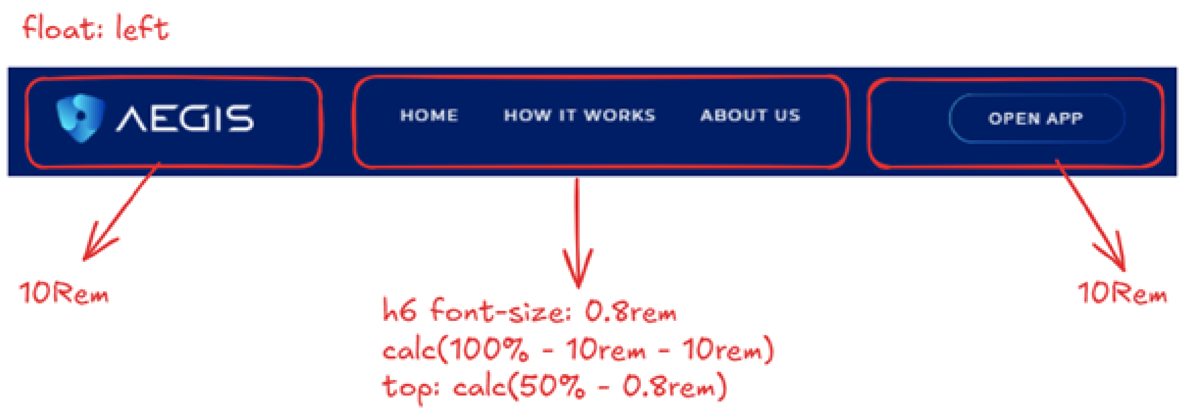


Figure 38: Aegis navigation bar implementation using float

In this layout, the float property is used to position the elements within the navigation bar. Below explains how the layout was properly implemented.

- **Logo:** Floated to the left, it takes up a fixed width (10rem) and spans the full height of the navigation bar.
- **Menu Items:** Floated left as well, with a width calculated to fill the remaining space in the navigation bar. It takes up the space left after the logo and button container.
- **Menu Links:** These are styled as inline blocks to position them horizontally. They're vertically centered within the menu using a top value calculation based on the container height.

Here is the codepen link to check how this is being implemented using float.

VI.4) Relevance of Calc Property

For responsive design, the `calc()` function in CSS provides enhanced control by allowing dynamic calculations for property values. This feature enables developers to create flexible layouts that adapt seamlessly to different screen sizes. Below is an illustration that demonstrates a practical use case of the `calc()` function in responsive design with a codepen provided here.

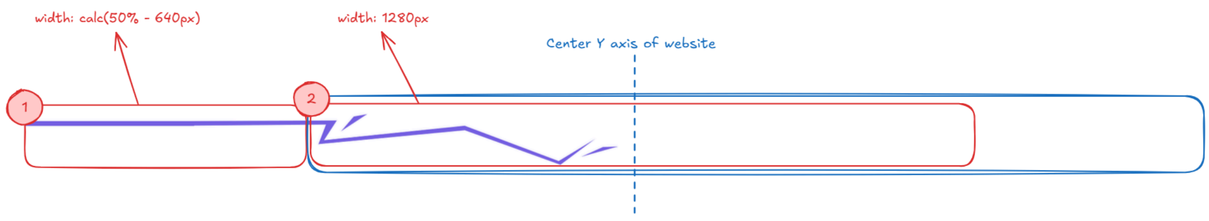


Figure 39: SAIB Responsive Lightning

- **SAIB Lightning:** The number 2 represents the SAIB lightning, which can be extracted either as a background image or an SVG. The lightning is assumed to be wrapped in a container with `width: 1280px`, making it the visual center of the website.
- **Lightning Responsive Extender:** Number 1 extends from the leftmost edge of the website's viewport to the starting point of the lightning. This is achieved using `calc(50% - 640px)`, where `50%` represents the horizontal center of the website (center of the viewport), and `640px` is half the width of the 1280px container. Subtracting `640px` from `50%` shifts the starting point of the extender to align with the left edge of the container, ensuring it ends precisely at the start of the SAIB lightning.

VI.5) Custom Flex Implementation

This section shows a creative use of inline-blocks which basically simulates the behavior of a flex element. Though it is not used in this website, its still worth noting how the creative use of display and position properties help create creative solutions. A codepen is provided here which effectively demonstrates this.

VI.6) First Section

For the first section of the Aegis website, implementation is explained in detail through the linked video. The video serves as a guide on creating the first section without using flexbox or grid, focusing instead on basic CSS fundamentals. While the video covers this section, the remaining parts of the website will be explained in the form of a documentation, with additional details and an explainer for each section.

VI.7) Second Section

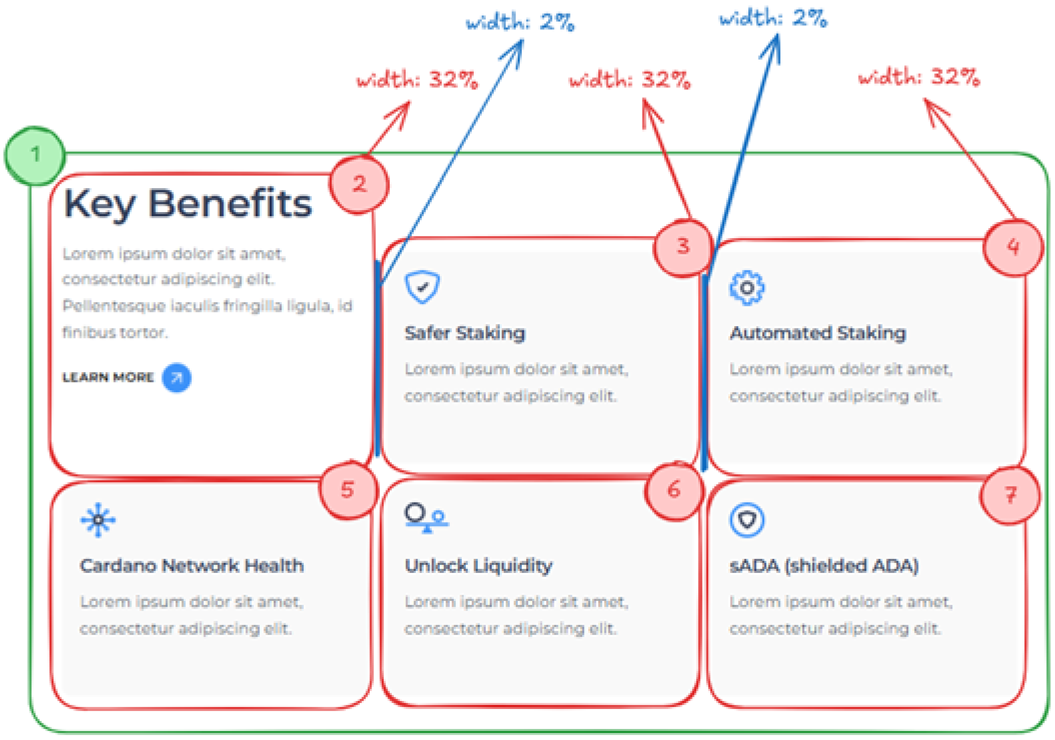
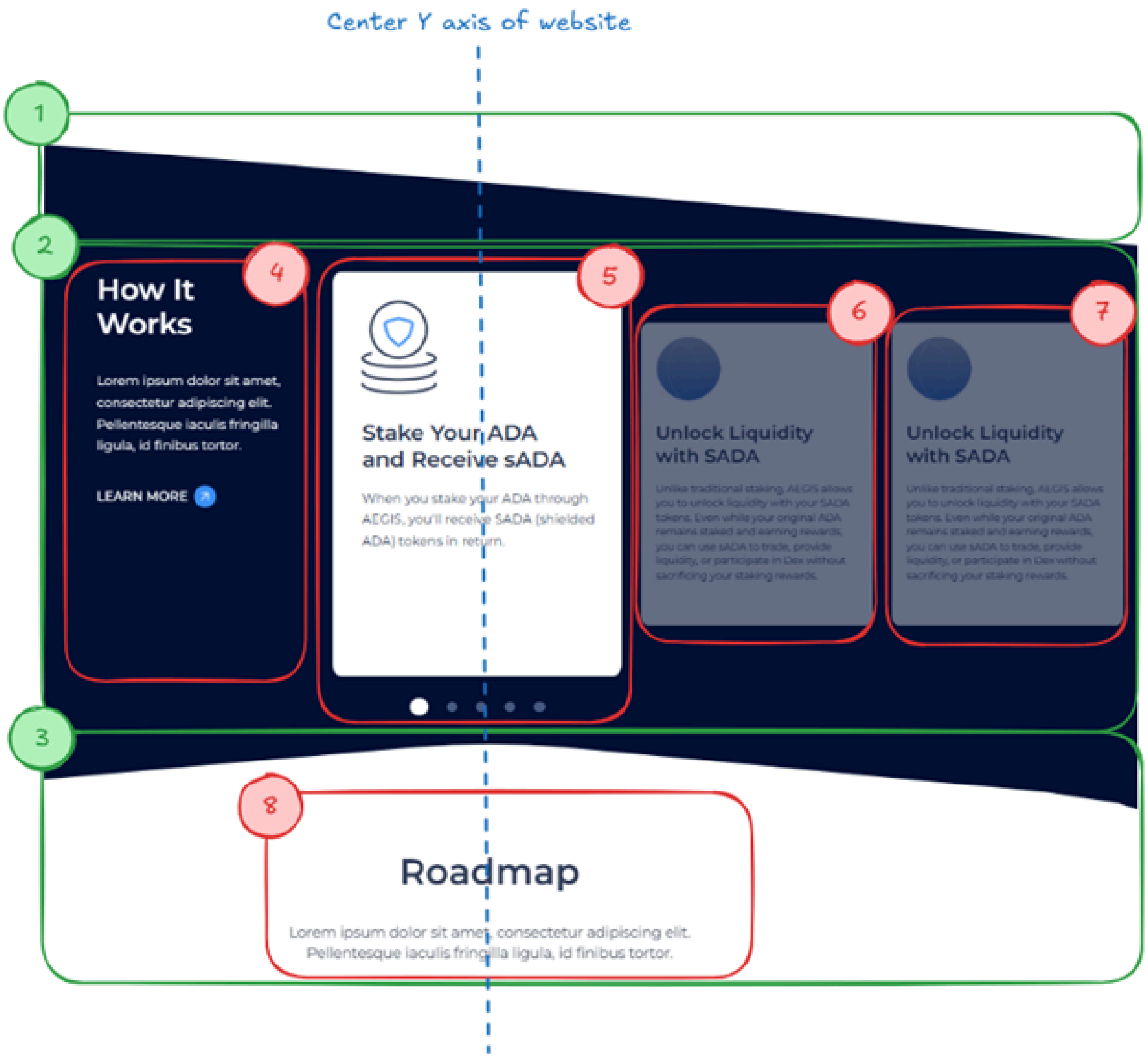

width: 2%
width: 2%
width: 32%
width: 32%
width: 32%
1
2
3
4
5
6
7
Key Benefits
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.
LEARN MORE
Safer Staking
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Automated Staking
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Cardano Network Health
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Unlock Liquidity
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
sADA (shielded ADA)
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Figure 40: Aegis second section without flex and grid

- **Background Section:** Number 1 serves as the container holding all the elements numbered 2 through 7.
- **Key Benefits Text:** Number 2 is the main article text for the key benefits. It has the same width as the other cards and is placed in the same row. However, it contains a different height because it is taller.
- **Key Benefits Cards:** Numbers 3 through 7 are the cards displaying the details of the key benefits. Each card has a width of 32% relative to its parent container (number 1), totaling 96% per row. For the second and third cards in the row, an additional 2% width is added between the cards to create space which totals to 100%.

VI.8) Third Section



How It Works

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

Stake Your ADA and Receive sADA

When you stake your ADA through AEGIS, you'll receive SADA (shielded ADA) tokens in return.

Unlock Liquidity with SADA

Unlike traditional staking, AEGIS allows you to unlock liquidity with your SADA tokens. Even while your original ADA remains staked and earning rewards, you can use sADA to trade, provide liquidity, or participate in Dex without sacrificing your staking rewards.

Unlock Liquidity with SADA

Unlike traditional staking, AEGIS allows you to unlock liquidity with your SADA tokens. Even while your original ADA remains staked and earning rewards, you can use sADA to trade, provide liquidity, or participate in Dex without sacrificing your staking rewards.

Roadmap

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

Figure 41: Aegis third section without flex and grid

- **Background Section:** Numbers 1 and 3 represent box models that contain an SVG as their background image, despite the appearance of curves. Number 2 serves as the background container that holds cards numbered 4 through 7. Since the cards are using inline-block, the `white-space: nowrap` property is applied to prevent the cards from wrapping and ensure they exceed the width of the right container.
- **Key Benefits Text:** Number 8 is the roadmap text which is centered within number 3. I used `left: 50% and translateX(-50%)` to achieve this effect.
- **Key Benefits Cards:** For this section, since the card exceeds the parent containers width to achieve the carousel display effect, I used rem to define the width of the cards instead of percentage. This is also relevant based on the blue line which shows the center y axis of the website which shows that the image displayed has its cards exceeding outside the

max-width 1280px container for its right side. For number 4, I used `vertical-align: top` to push the text on the topmost section, and for number 5-7, I used vertical align: middle so that the subsequent cards will be centered, and this works since the cards are all inline block elements.

VI.9) Fourth Section

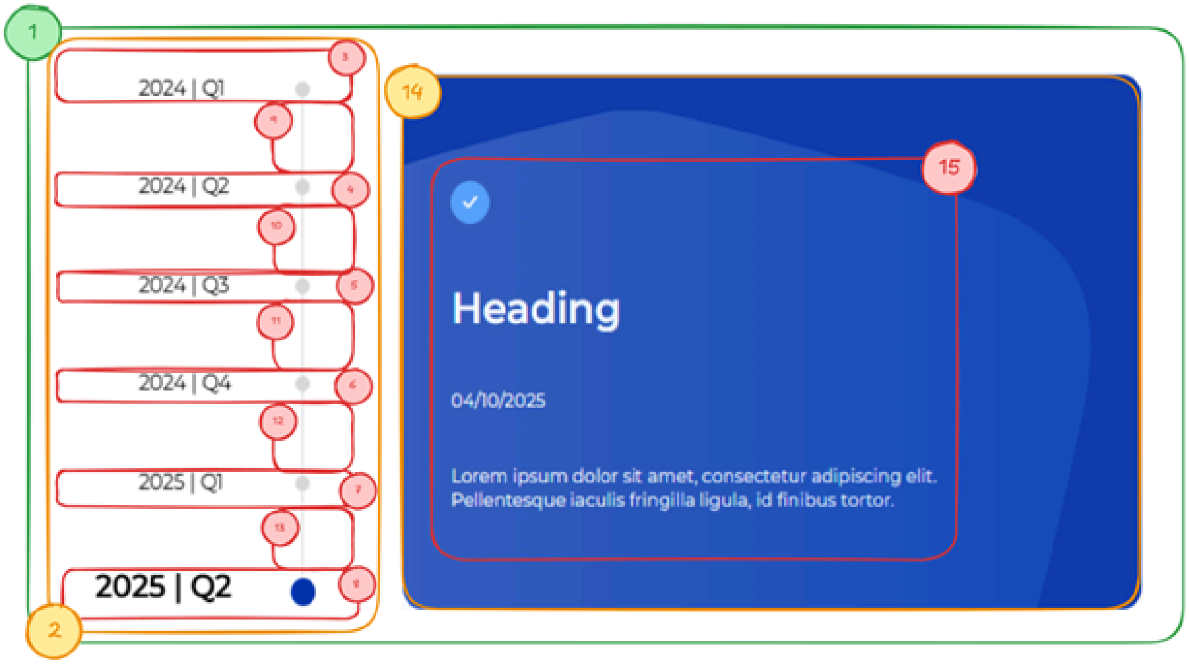

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
2024 | Q1
2024 | Q2
2024 | Q3
2024 | Q4
2025 | Q1
2025 | Q2
Heading
04/10/2025
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

Figure 42: Aegis fourth section without flex and grid

For this section, although the implementation here differs significantly from the one presented in the repository, I believe this approach is better as it adheres to the principles of box models. I will explain how this can be implemented with proper box modeling.

- **Background Section:** Number 1 holds both the number 2 timeline and number 14 heading containers.
- **Timeline Section:** Represented as number 2, this section holds the different quarter numbers in the timeline and contains two types of box models. The first set, numbered from 3 to 8, includes the **year | quarter** text along with a period on the right side of the text. Below each number, there is a box model (numbered 9 to 13) that represents the line extending to the nearest adjacent period.
- **Heading Section:** Represented as number 14, this section features a curved triangle SVG as its background image. The heading text, represented as number 15, is the article text placed on the left side within the heading section container.

VI.10) Fifth Section

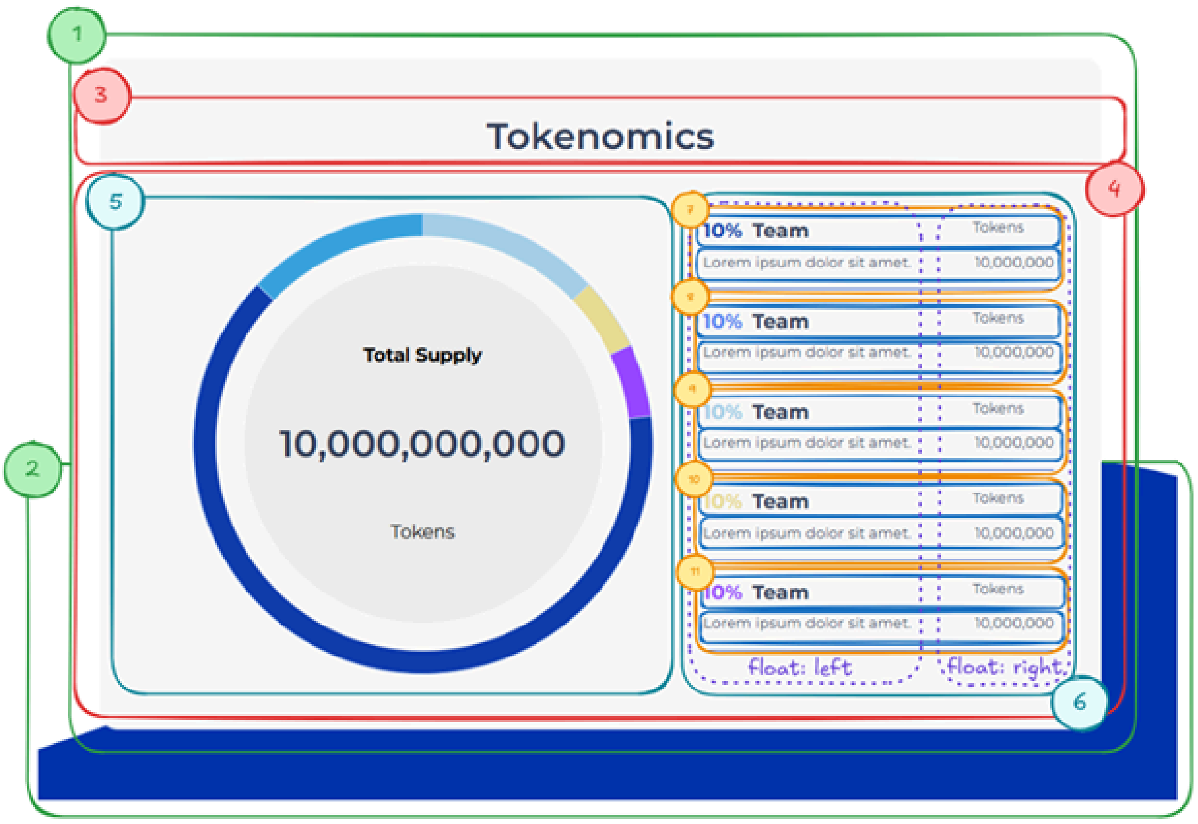


Figure 43: Aegis fifth section without flex and grid

- **Background Section:** Number 1 contains the tokenomics section along with the elements that define it. Number 2 is an element with a background SVG, which gives it a curved appearance, but it is inherently a box model.
- **Tokenomics Title:** This section represents the tokenomics title and is centered vertically with `text-align: center` property.
- **Tokenomics Pie Chart:** Represented as container number 5, this section holds an SVG pie chart with text at the center of the pie. The width of this container is 55% relative to its parent container, which is number 4.
- **Tokenomics Details Section:** This section, represented by number 6, contains details that define the tokenomics statistics, numbered from 7 to 11. The width of the details section is 45%, making the total width of the pie chart and details container parent 100%. The text elements are arranged in two rows and two columns inside their respective orange containers (numbered 7 to 11). To separate the elements, the left text is positioned using `float: left`, while the right text is positioned using `float: right`, which corresponds to the purple square.

VI.11) Fifth Section

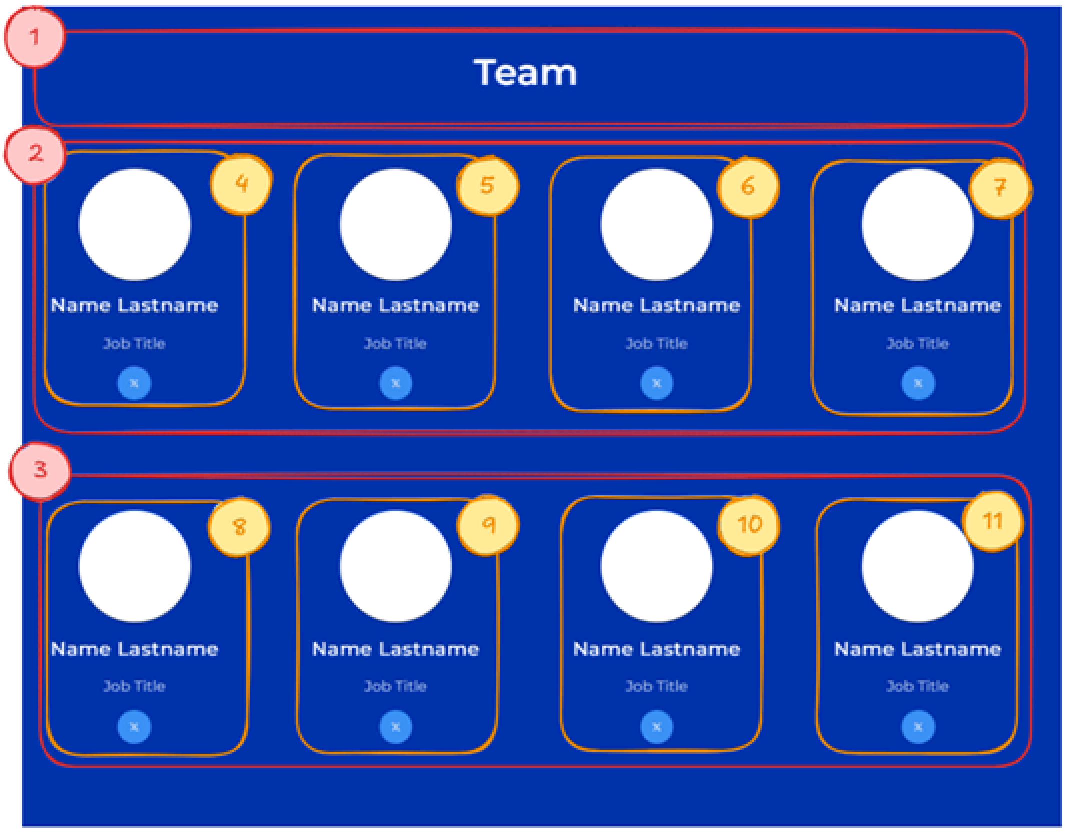


Figure 44: Aegis sixth section without flex and grid

- **Team Title:** Like the previous tokenomics section, the title (number 1) is vertically centered using the `text-align: center` property.
- **Team Information Containers:** This section, represented by numbers 2 and 3, serves as the containers that hold all the team information elements. For these containers, the `text-align: justify` property is applied, ensuring that the information cards within the container are justified. This works because each information card is declared as an inline-block element.
- **Team Information Cards:** In this section, each card is defined as a list item within unordered lists, styled as inline-block elements. The cards automatically distribute themselves without the need to explicitly define gaps, like how it is normally managed in Flexbox. The cards are numbered from 4 to 11.

VI.12) Seventh Section

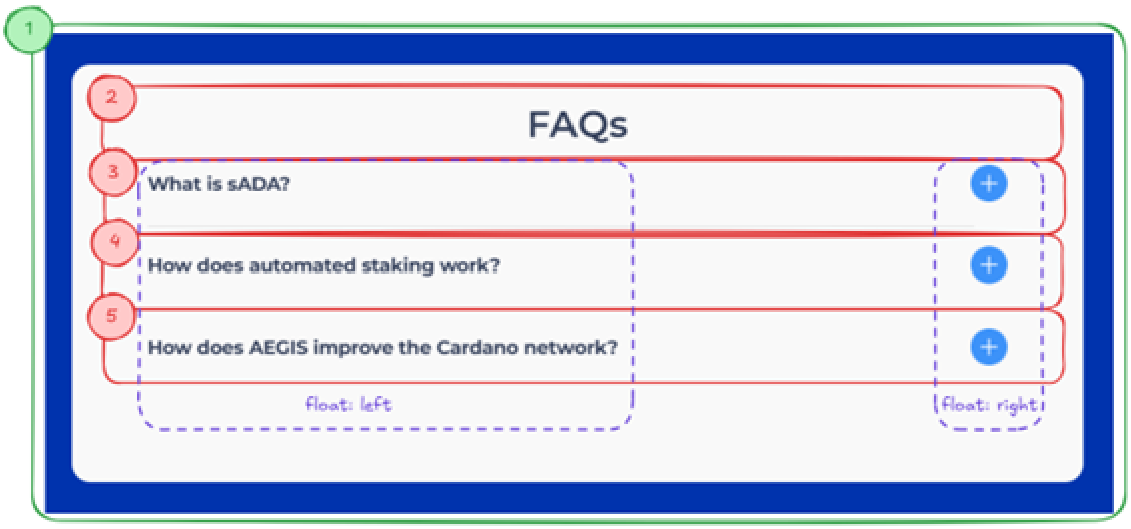

1
2
FAQs
3
What is sADA?
4
How does automated staking work?
5
How does AEGIS improve the Cardano network?
float: left
float: right

Figure 45: Aegis seventh section without flex and grid

- **Background Section:** Number 1 contains the FAQ section along with the elements that define it.
- **FAQ Title:** Number 2 contains the FAQ title is vertically centered using the text-align: center property.
- **FAQ Texts:** Numbers 3 to 5 contain two different elements, the one represented at the left side is positioned using float: left, while the buttons on the right side are represented as float: right.

VI.13) Aegis Footer Section

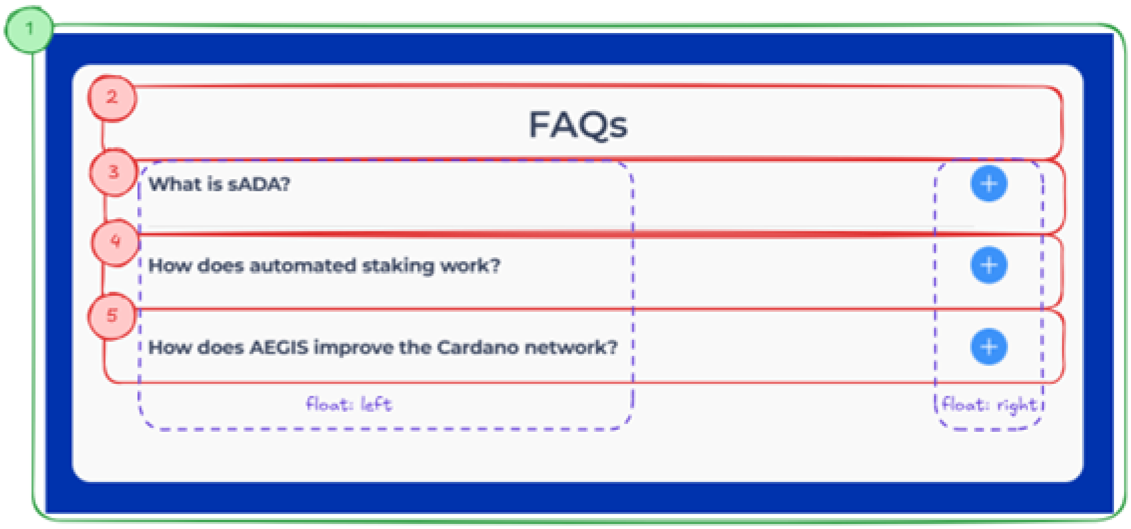

1
2
FAQs
3
What is sADA?
4
How does automated staking work?
5
How does AEGIS improve the Cardano network?
float: left
float: right

Figure 46: Aegis footer section without flex and grid

- **Background Section:** Number 1 contains the Footer section along with the elements that define it.
- **Footer Containers:** Numbers 2 and 3 serve as the footer container which holds the respective elements holding necessary texts on what defines this section.
- **Footer Information:** Numbers 4 to 10 are the elements that represent the information found in the footer. Numbers 4 to 6 and 8 are positioned using `float: left` and Numbers 7, 9, and 10 are positioned using `float: right`.

VII) Github Conventional Commits

VII.1) Overview

The Conventional Commits specification is a lightweight convention built on top of commit messages. It establishes a set of rules for creating clear and consistent commit histories, making it easier to write automated tools and processes around commits. In this section, I will discuss the commonly used conventional commits on GitHub. For more detailed information, you can refer to the official documentation here.

VII.2) Prerequisites

Before diving into GitHub Conventional Commits, developers should have a foundational understanding of how GitHub works and how version control systems, like Git, are used to manage code.

VII.3) Conventional Commits Structure

Conventional commits are structured based on the following **format: <`type`> [optional scope]: <`description`>**. For the following, I will show what each type entails and give an example on the message that can be associated with each:

Github Commit Messages	Description	Examples
fix	A commit that addresses a bug or fixes an issue.	fix: resolve login error when user submits invalid credentials
feat	A commit that introduces a new feature or functionality.	feat: add search bar to navigation header
build	A commit that affects the build system or dependencies (e.g., updating tools or libraries).	build: update webpack to version 5.0.0
chore	A commit for routine tasks or maintenance work that doesn't affect application functionality.	chore: deleted asset files
ci	A commit related to continuous integration (e.g., changes to build scripts or CI configurations).	ci: add deployment pipeline for staging

Github Commit Messages	**Description**	**Examples**
docs	A commit that updates documentation (e.g., README files, API docs).	docs: update installation instructions
style	A commit that corrects styling issues without affecting the application's functionality (e.g., formatting, whitespace).	style: fix inconsistent indentation in main.css
refactor	A commit that refactors or improves the codebase without adding new features or fixing bugs.	refactor: simplify carousel hook
perf	A commit that improves performance (e.g., optimization).	perf: reduce response time by optimizing database queries
test	A commit that adds or modifies tests (e.g., unit tests, integration tests).	test: add unit tests for login validation

VIII) Material UI

VIII.1) Overview

Material UI is an open source React component library designed to bring Google's Material Design principles to life. It offers a rich set of prebuilt components that are production-ready, making it an excellent choice for building modern, responsive user interfaces. With its extensive customization options, Material UI allows developers to seamlessly integrate unique design systems on top of its components. Our company leverages Material UI to deliver consistent, visually appealing, and highly functional interfaces across all our projects, ensuring a polished user experience. More information on this document can be found in this article.

VIII.2) Prerequisites

Before diving into MUI, developers are expected to have a foundational understanding of the following technologies: **HTML, CSS, JavaScript, Typescript, and React**.

VIII.3) Version

The current version of MUI during the inception of this documentation is **v.6.3.1**, any information discussed in this documentation is subject to change.

VIII.4) MUI Theme Customization

Every company has its own unique theme that aligns with its brand identity. At SAIB, our primary customization is based on a specific color palette, though this may vary depending on individual project requirements, such as client-specific projects. For instance, Crashr follows a different theme, yet it maintains a consistent design language that reflects coherence and often conveys a deeper meaning. This approach ensures a harmonious visual identity across all projects while allowing flexibility to meet diverse needs.

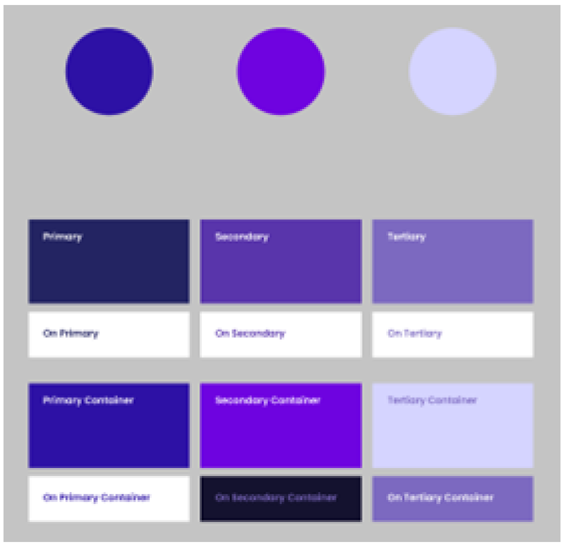
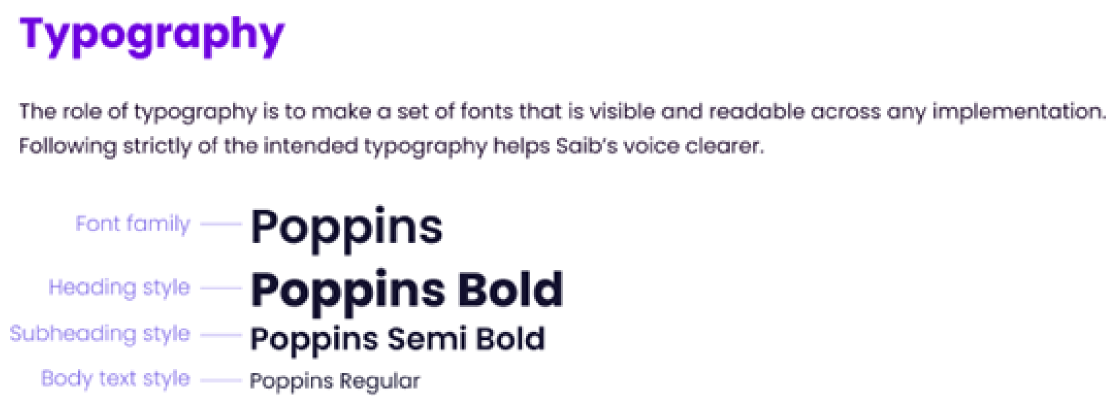


Figure 47: SAIB palette representing its color theme



Typography

The role of typography is to make a set of fonts that is visible and readable across any implementation. Following strictly of the intended typography helps Saib's voice clearer.

Font family	Poppins
Heading style	**Poppins Bold**
Subheading style	**Poppins Semi Bold**
Body text style	Poppins Regular

Figure 48: SAIB palette representing its typography

To create a theme in Material UI, developers can use the `createTheme` function, which allows customization of the palette, typography, and other design elements as outlined in the documentation. The generated theme is then passed as a prop to the `ThemeProvider` component, which applies the theme to the entire React component tree it wraps. For optimal results, `ThemeProvider` should be placed at the root of your component hierarchy to ensure consistent theming across your application. For more information, refer to this section of this documentation. For the following example, developers can refer to this repository as a project reference which is also implemented using Material UI (MUI), Gatsby, and React.

Below is a code snippet which shows the implementation of theming:

```
const theme = createTheme({
  palette: {
    primary: {
      main: red[500],
    },
  },
});



function App() {
  return <ThemeProvider theme={theme}>...</ThemeProvider>;
}
```

As a sidenote, when using Tailwind CSS with Material UI, developers can leverage the breakpoints property provided by Material UI's theme. This is particularly useful because the breakpoints in Material UI differ from those in Tailwind CSS. Maintaining consistency is important, but adapting to one set of breakpoints over the other is not mandatory as it ultimately comes down to personal preference or company standards.

MUI Breakpoints			**TailwindCSS Breakpoints**		
xs	**extra-small**	0px	**sm**	**small**	640px
sm	**small**	600px	**md**	**medium**	768px
md	**medium**	900px	**lg**	**large**	1024px
lg	**large**	1200px	**xl**	**extra-large**	1280px
xl	**extra-large**	1536px	**2xl**		1536px

Here is how I was able to change the breakpoints using the `createTheme` function:

```
breakpoints: {
  values: {
    xs: 0,
    sm: 640,
    md: 768,
    lg: 1024,
    xl: 1280,
  },
},
```

VIII.5) MUI Building Custom Palettes

One common limitation in MUI is the restricted number of predefined palettes available for creating themes. By default, MUI provides only a limited set of palettes such as **primary, secondary, error, info, success, and warning**. This can pose challenges in scenarios where additional or more descriptive color palettes are required to convey meaning effectively.

For instance, in a design like the tokenomics section shown below, there are five distinct colors. However, with MUI's default configuration, each palette allows defining only three-color variants (e.g., main, light, and dark). This restriction can make it difficult to fully customize or extend themes for projects that require more nuanced or varied color schemes.

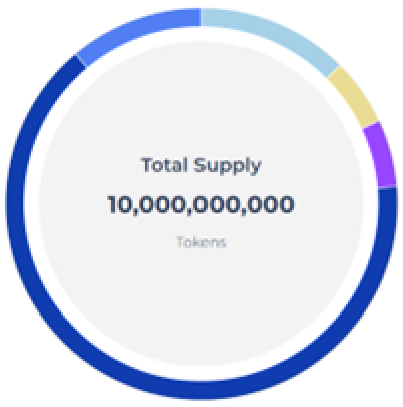


Figure 49: Search interface of open graph website

The tokenomics component is from MUI which can be found in this section of the documentation.

For typescript, here is a way to create custom palettes:

1. **Module Declaration:** Based on the code below, this extends the existing Palette interface from `@mui/material/styles` to include a new property called tokenomics. The module declaration ensures that TypeScript recognizes tokenomics as part of the Palette interface, preventing type errors. The tokenomics property contains six custom colors (darkgray, darkblue, navyblue, lightblue, yellow, purple), each represented as a string.

```
declare module "@mui/material/styles" {
    interface Palette {
        tokenomics: {
            darkgray: string;
            darkblue: string;
            navyblue: string;
            lightblue: string;
            yellow: string;
            purple: string;
        };
    }
}
```

2. **Optional Palette Options:** The **PaletteOptions** interface is similarly extended to include an optional tokenomics property. This is necessary because createTheme uses **PaletteOptions** for theme configuration, so adding tokenomics ensures compatibility when defining a custom theme.

```
interface PaletteOptions {
  tokenomics?: {
    darkgray?: string;
    darkblue?: string;
    navyblue?: string;
    lightblue?: string;
    yellow?: string;
    purple?: string;
  };
}
```

3. **Creating the Theme:** A new theme is created using **createTheme**, with the custom tokenomics color palette defined under the palette key. The colors are specified as hexadecimal values, representing the required custom shades.

```
const theme = createTheme ({
  palette: {
    tokenomics: {
      darkgray: "#F3F3F3",
      darkblue: "#0F3CAE",
      navyblue: "#517FF3",
      lightblue: "#A5D0E8",
      yellow: "#E9DC94",
      purple: "#9747FF",
     },
   },
});
```

VIII.6) MUI Light and Dark Mode

This functionality is available in Material UI and allows websites to implement a toggle for switching between two modes: light mode and dark mode. It is a common feature that enhances user experience by adapting the interface to user preferences or environmental conditions. More details can be found in the relevant section of the documentation. The image below illustrates a comparison between light and dark modes as implemented on a website.

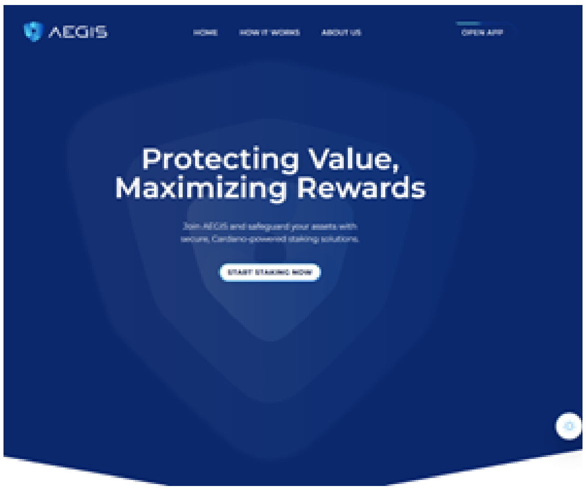


Figure 50: Aegis website light mode

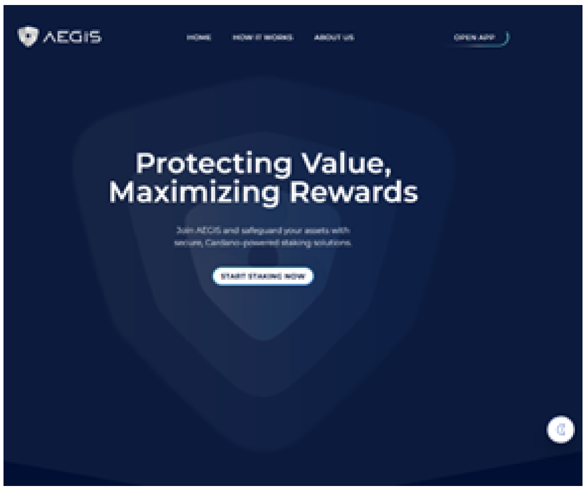


Figure 51: Aegis website dark mode

To achieve this, we use the **colorSchemes** property to define multiple color schemes for a theme, such as light and dark modes. Within each scheme, we specify the palette and associate it with the corresponding colors for that theme. Here's an example:

```
const theme = createTheme({
  colorSchemes:
  {
    light:
    {
      palette:
      {
        background: { default: "#FFFFFF" },
      }
    }
    dark:
    {
      palette:
      {
        background: { default: "#000F33" },
      }
    }
  }
```

To enable dark and light mode in React, we first import the **useColorSchemes** hook from Material UI in another file:

```
import { useColorScheme } from "@mui/material";
```

Using the **useColorSchemes** hook, we can access the current mode (light or dark) and update it as needed. This can also be referenced in this part of the documentation.

```
const { mode, setMode } = useColorScheme();
```

Next, we create a function for toggling between light and dark modes:

```
const darkMode = () => {
  setMode(mode === "dark" ? "light" : "dark");
};
```

Finally, we implement a button that toggles the mode. The button updates the theme based on the current mode and displays an appropriate icon:

```
<div onClick={darkMode}>
  <img
    src=
    {
      mode === "dark"
        ? firstSectionImages.moonIcon.publicURL
        : firstSectionImages.sunIcon.publicURL
    }
  />
</div>
```

And now based on this code, the icon of the button changes depending on its mode which can be reflected here:

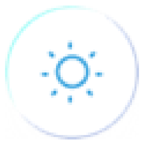


Figure 52: Aegis website light mode icon

Figure 53: Aegis website dark mode icon

VIII.7) MUI Editing with SX Props

The sx prop is a special feature in MUI that allows you to apply styles using standard CSS attributes combined with MUI's theme-aware styling solution. It leverages the MUI System to provide a powerful and flexible way to style components. Below is an example demonstrating how the sx prop can be used to access colors defined in the theme:

```
<Box
  component="section"
  sx={{
    backgroundColor: theme.palette.primary.main,
  }}
/>
```

The SX prop also supports pseudo-classes and nested selectors, such as `&:hover` enabling powerful and intuitive styling for dynamic and interactive states. Below is also a code example of how this can be applied in MUI:

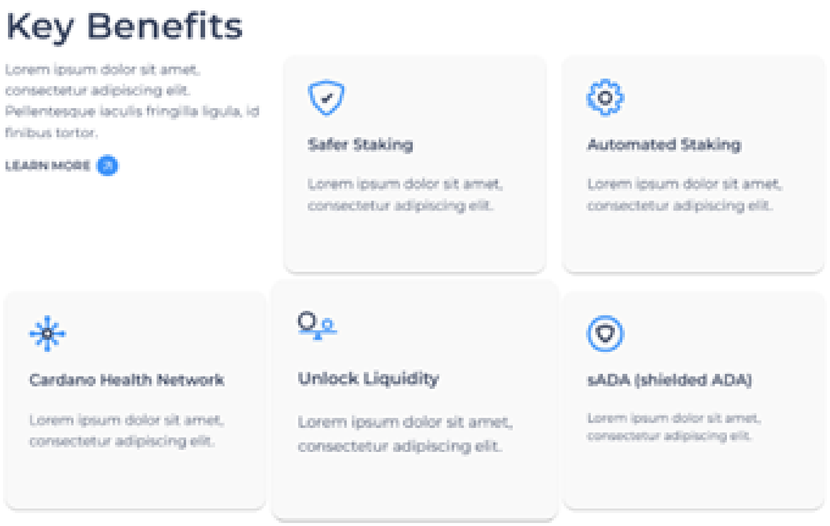
```
<Box
  component="section"
  sx={{
    backgroundColor: theme.palette.primary.main,
    '&:hover': {
      backgroundColor: theme.palette.secondary.main,
    },
  }}
/>
```

VIII.8) Relevant MUI Components

This section discusses two key MUI components that were primarily used in the development of the Aegis website.

- **Box Component:** The Box component is highly versatile and allows developers to create layout components such as sections, articles, or divs with easy access to the theme. It provides a convenient wrapper that can be styled dynamically, making it ideal for building flexible and customizable layouts.

- **Card Component:** The Card component is used to display content in a structured, visually appealing format. It can be easily elevated to add shadows, creating a sense of interactivity and depth. This makes the Card component perfect for showcasing content like blog posts, product details, or any other information that benefits from a clean, modular presentation. Below shows an example of which uses a card component in the Aegis website.


Key Benefits

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque iaculis fringilla ligula, id finibus tortor.

LEARN MORE

Safer Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Automated Staking

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Cardano Health Network

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Unlock Liquidity

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

sADA (shielded ADA)

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Figure 54: Key benefits section that uses card components

IX) TailwindCSS Guidelines

IX.1) Overview

Tailwind CSS is a utility-first CSS framework that enables rapid UI development by providing low-level utility classes. These classes are designed to be composable, making it easy to build complex and custom designs without writing traditional CSS. In this section of the documentation, the relevance of TailwindCSS implementation in this company will be outlined. More information will be outlined in the tailwind documentation.

IX.2) Prerequisites

Before diving into TailwindCSS, developers are expected to have a foundational understanding of the following technologies: **HTML, CSS, Responsive Design, and Javascript**.

IX.3) Version

The current version of TailwindCSS during the inception of this documentation is **v.4**, any information discussed in this documentation is subject to change.

IX.4) Utility First Approach

Tailwind focuses on small, reusable utility classes such as `flex`, `p-4`, `text-center`, and `bg-blue-500`, rather than predefined components. Its inline format enhances maintainability across projects, as styles are defined directly within the element or component, making it more convenient and efficient to create and manage designs. Here is a code that demonstrates an implementation of tailwindcss as a reference:

```html
<div class="flex flex-col items-center justify-center p-4 bg-blue-500 text-white
rounded-lg shadow-lg max-w-sm mx-auto">
  <h1 class="text-center text-2xl font-bold">
  Welcome to TailwindCSS
  </h1>
</div>
```

IX.5) Arbitrary Values

Tailwind CSS supports arbitrary values, allowing you to specify custom values for properties that aren't predefined in the configuration. This feature provides fine-grained control over styles, enabling you to use any value (e.g., pixels, percentages, rems, etc.) directly within your utility classes without modifying the configuration file. Arbitrary values are typically defined using the syntax shown below:

```
<div data-current class="opacity-[75rem] />
```

However, in the latest version of TailwindCSS, dynamic values now support a new set of arbitrary values, replacing some predefined behaviors from older versions. Here's an example demonstrating the updated syntax:

```
<div data-current class="opacity-75 />
```

IX.6) Responsive Design

Every utility class in Tailwind can be applied conditionally at different breakpoints, making it convenient to create responsive interfaces directly in HTML instead of relying on separate CSS files. Below are the five default breakpoints in Tailwind CSS, designed with a mobile-first approach. Each breakpoint specifies the minimum screen width at which it becomes active:

Breakpoint Prefix	Minimum Width	CSS
sm	**40rem(640px)**	`@media (min-width: 640px) { ... }`
md	**48rem(768px)**	`@media (min-width: 768px) { ... }`
lg	**64rem(1024px)**	`@media (min-width: 1024px) { ... }`
xl	**80rem(1280px)**	`@media (min-width: 1280px) { ... }`
2xl	**96rem(1536px)**	`@media (min-width: 1536px) { ... }`

By default, styles applied by rules like `md:flex` will apply at that breakpoint and stay applied at larger breakpoints. Below is an example which shows how responsive design works in the context of tailwindcss.

```
<div class="flex flex-col items-center justify-center p-4 bg-blue-500 text-white
rounded-lg shadow-lg max-w-sm md:max-w-md lg:max-w-lg mx-auto">
  <h1 class="text-center text-xl md:text-2xl lg:text-3xl font-bold">
    Welcome to TailwindCSS
  </h1>
</div>
```

IX.7) Mobile First Implementation

In a mobile-first approach with Tailwind CSS, styling for mobile devices uses the unprefixed utility classes. The sm: prefix doesn't mean "for small screens" but rather "starting from the small breakpoint and up." This means mobile styles are the default, while breakpoint prefixes define styles for larger screen sizes.

⊗ Don't use sm: to target mobile devices:

```
<div class="sm:text-center"></div>
```

✓ Use unprefixed utilities to target mobile, and override them at larger breakpoints

```
<div class="text-center sm:text-left"></div>
```

IX.8) New Gradient Updates

As discussed in **Section 3.10** regarding the implementation of the Crashr button, TailwindCSS **v4** introduces support for radial and conic gradients, expanding beyond the previously supported linear gradients. This update allows for greater flexibility and creativity when working with gradient-based designs, overcoming the limitations of earlier versions. For example, the following codepen implementation demonstrates the use of a conic gradient in CSS:

```
.gradient-border-button {
  background: conic-gradient(transparent 25%, #40DCC8, #6073F6 99%, transparent);
}
```

Now in tailwindcss **v4**, this can now be implemented as:

```
<div class="bg-conic from-[#FFFFFF] via-[#40DCC8] to-[#6073F6]"></div>
```

Linear gradients also improved as they now support angles which allows them to abstract from the standard left-to-right and vice versa implementations. This is also represented based on the code snippet provided below:

```html
<div class="bg-linear-45 from-indigo-500 via-purple-500 to-pink-500"></div>
```

X) Gatsby Guidelines

X.1) Overview

Gatsby is a powerful open-source static site generator (SSG) built on React. It enables developers to create fast, modern websites and web applications with ease. By leveraging React, GraphQL, and a rich plugin-based architecture, Gatsby allows for the development of high-performance, scalable, and SEO-friendly static websites.

In this section of the documentation, I will explain the role of Gatsby in building the AEGIS website, which is hosted in this repository. Specifically, I will discuss how Gatsby's unique features, such as its GraphQL data layer and plugin ecosystem, were utilized to streamline development and optimize performance.

X.2) Prerequisites

Before diving into Gatsby, developers are expected to have a foundational understanding of the following technologies: **HTML, CSS, JavaScript, TypeScript, React, Node.JS, and GraphQL**.

X.3) Version

The current version of Gatsby during the inception of this documentation is **v5**, any information discussed in this documentation is subject to change.

X.4) Video Reference

This video reference, discussed by Clark (CEO), provides an in-depth exploration of the relationship between TCP, DNS, Server-Side Rendering (SSR), and Gatsby. It offers a detailed, low-level explanation of how these technologies interact, why they are used, and their impact on building modern web applications. Watching this video will give you a deeper appreciation of the underlying mechanisms that make Gatsby a powerful tool for developers.

X.5) Gatsby Reference

Gatsby's configuration is primarily managed through the **gatsby-config.js** file, which is in the root directory of a Gatsby project. This file acts as the central hub for defining a project's settings, plugins, and metadata. Below is an explanation which details its key components and structure in typescript:

1. **siteMetadata:** An object used to store metadata about the site, such as its title, description, author, and other custom fields. This metadata can be queried via GraphQL and used throughout the site. The code below is used as a reference on how site metadata is implemented using gatsby:

   ```
   const config: GatsbyConfig = {
     siteMetadata: {
       title: `My Gatsby Site`,
       siteUrl: `https://www.yourdomain.tld`,
     },
   }
   ```

2. **plugins:** An array which lists the plugins your Gatsby project uses. Plugins enhance the site's functionality, such as sourcing data, optimizing images, or enabling SEO. Each plugin can be added as a string or as an object (custom settings). Each plugin can be added as a string or as an object (custom settings).

   ```
   const config: GatsbyConfig = {
     plugins: [
       "gatsby-plugin-postcss",
       "gatsby-plugin-image",
       "gatsby-plugin-sharp",
       "gatsby-transformer-sharp",
       {
         resolve: `gatsby-source-filesystem`,
         options: {
           name: `images`,
           path: `${__dirname}/src/images/`,
         },
       },
     ],
   }
   ```

Developers can also add Google Fonts and favicons by configuring the plugins section in the gatsby-config.js file. This allows for custom configurations tailored to the project's requirements, as demonstrated in the example code below:

```
const config: GatsbyConfig = {
  plugins: [
    {
      resolve: "gatsby-plugin-google-fonts",
      options: {
        fonts: [
          "montserrat:100,200,300,400,500,600,700,800,900:latin,latin-    ext",
        ],
        display: "swap",
      },
    },
    {
      resolve: "gatsby-plugin-manifest",
      options: {
        icon: "src/images/icons/aegis-icon.svg",
      },
    },
  ],
}
```

X.6) Querying Data with GraphQL

This section explains how data is queried in Gatsby from its data layer. The process involves several steps that work together seamlessly.

- **Source Plugins:** Data in Gatsby comes from multiple sources, including the filesystem, content management systems (CMS), private APIs, and databases. These are referred to as "source plugins." Each source plugin is designed to interact with a specific data source and bring that data into Gatsby's data layer.
- **Data Layer:** The data from various sources is integrated into a unified data layer. This layer serves as a central location for all of Gatsby's data, regardless of where it originally came from.
- **GraphQL Queries:** To access this data, Gatsby uses GraphQL queries. Developers write these queries to extract specific data from the data layer. The queries are run during the build process, and the resulting data is then made available in the components that request it.

Below is an illustration of how this process works under the hood:

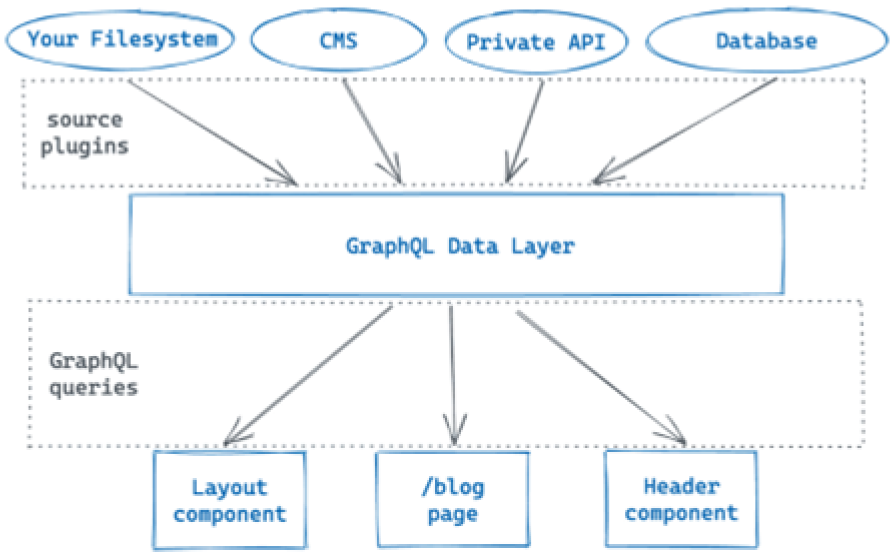
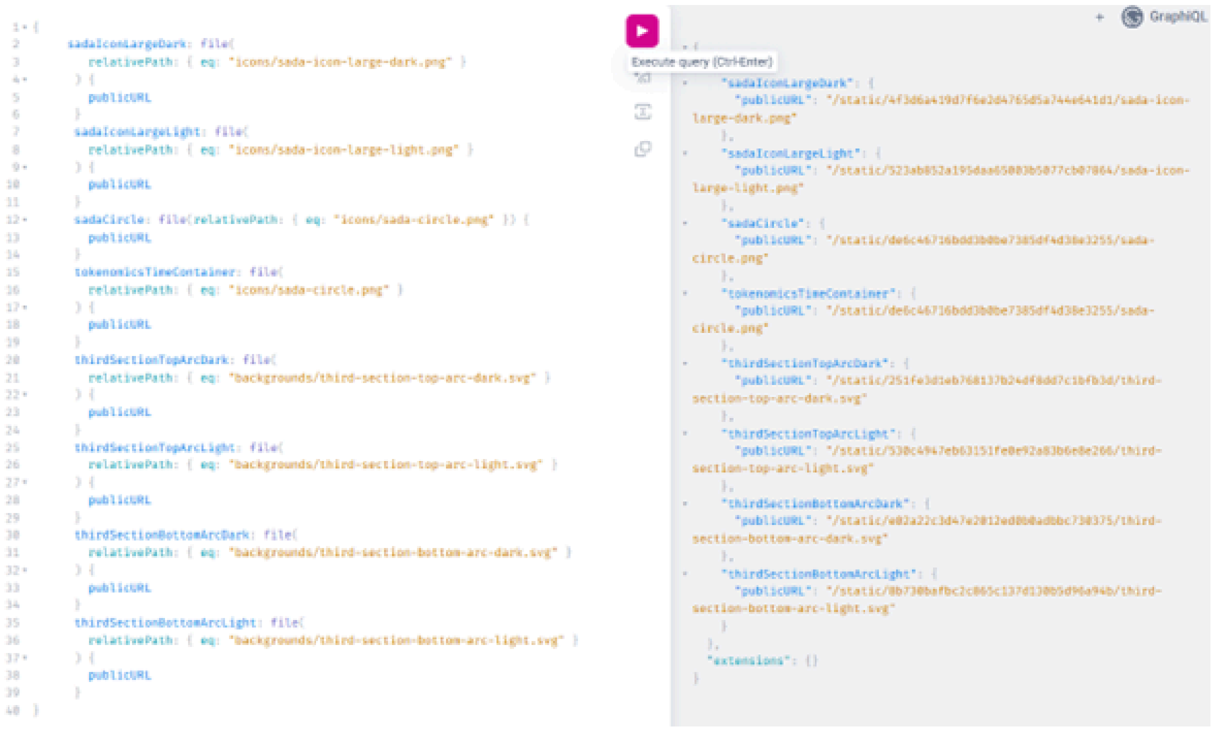


Figure 55: GraphQL data layer abstraction

X.7) Exploring the Data Layer

To check if the data from source plugins is available in Gatsby's GraphQL data layer, developers can use GraphiQL, an interactive tool that helps structure and test GraphQL queries. GraphiQL provides an interface where developers can experiment with queries, ensuring they are correctly formatted and return the expected data. This is a great way to validate your queries before integrating them into the actual code, ensuring smooth data retrieval and integration.

For example, you can use GraphiQL to query file data, like images or assets, stored in Gatsby's data layer. The following GraphQL query retrieves URLs for various image files from the data layer:



```
{
      sadaIconLargeDark: file(
        relativePath: { eq: "icons/sada-icon-large-dark.png" }
       ) {
        publicURL
       }
       sadaIconLargeLight: file(
        relativePath: { eq: "icons/sada-icon-large-light.png" }
       ) {
        publicURL
       }
       sadaCircle: file(relativePath: { eq: "icons/sada-circle.png" }) {
        publicURL
       }
       tokenomicsTimeContainer: file(
        relativePath: { eq: "icons/sada-circle.png" }
       ) {
        publicURL
       }
       thirdSectionTopArcDark: file(
        relativePath: { eq: "backgrounds/third-section-top-arc-dark.svg" }
       ) {
        publicURL
       }
       thirdSectionTopArcLight: file(
        relativePath: { eq: "backgrounds/third-section-top-arc-light.svg" }
       ) {
        publicURL
       }
       thirdSectionBottomArcDark: file(
        relativePath: { eq: "backgrounds/third-section-bottom-arc-dark.svg" }
       ) {
        publicURL
       }
       thirdSectionBottomArcLight: file(
        relativePath: { eq: "backgrounds/third-section-bottom-arc-light.svg" }
       ) {
        publicURL
       }
}
```

```
{
      "sadaIconLargeDark": {
        "publicURL": "/static/4f3d6a419d7f6e2d4765d5a744e641d1/sada-icon-
large-dark.png"
      },
      "sadaIconLargeLight": {
        "publicURL": "/static/523ab852a195daa65083b5077cb07864/sada-icon-
large-light.png"
      },
      "sadaCircle": {
        "publicURL": "/static/de6c46716bdd3b0be7385df4d38e3255/sada-
circle.png"
      },
      "tokenomicsTimeContainer": {
        "publicURL": "/static/de6c46716bdd3b0be7385df4d38e3255/sada-
circle.png"
      },
      "thirdSectionTopArcDark": {
        "publicURL": "/static/251fe3d1eb768137b24df8dd7c1bfb3d/third-
section-top-arc-dark.svg"
      },
      "thirdSectionTopArcLight": {
        "publicURL": "/static/530c4947eb63151fe0e92a83b6e0e266/third-
section-top-arc-light.svg"
      },
      "thirdSectionBottomArcDark": {
        "publicURL": "/static/e02a22c3d47e2012ed0b0adbbc730375/third-
section-bottom-arc-dark.svg"
      },
      "thirdSectionBottomArcLight": {
        "publicURL": "/static/0b730bafbc2c065c137d1305d96a94b/third-
section-bottom-arc-light.svg"
      }
    },
    "extensions": {}
}
```

Figure 56: Fetching data in Gatsby through graphiQL

X.8) Fetching Data with Gatsby's useStaticQuery Hook

useStaticQuery is a React hook provided by Gatsby that enables developers to execute GraphQL queries at build time and retrieve data from Gatsby's data layer. This hook is primarily used in components to fetch static data, such as images, markdown content, and other files, that are available when the site is built. By using **useStaticQuery**, Gatsby can pre-fetch and optimize the data during the build process, making it immediately available on the frontend without the need for additional requests or runtime fetching. Below is an example of how it can be used to fetch an image file:

```
const fourthSectionImages = useStaticQuery(graphql`
  query {
   aegisCheckMark: file(relativePath: {eq: "icons/aegis-check-mark.png"})
    {
        publicURL
    }
  }
`)
```

In this example, the **useStaticQuery** hook executes a GraphQL query to search for the aegis checkmark image with its relative path. The query retrieves the **publicURL** of the file, which provides the image URL for use within the component. This allows developers to easily reference static files, such as images, without needing to manage complex data fetching at runtime.

X.9) Gatsby Image Plugin

This section explains the configurations necessary for integrating images into the Gatsby image pipeline, which is fed through the GraphQL data layer. The Gatsby Image plugin provides two key components for displaying responsive images on your site:

- StaticImage
  - **Description:** This is used when the image is static, meaning it doesn't change each time the component is used. This is ideal for images that remain the same across all instances of the component.
  - **Examples:** Site logo, hero image on the homepage, and icons or other static branding elements
- GatsbyImage
  - **Description:** This is used for images that are dynamic and passed into the component as props. This component allows for images to change based on data, making it suitable for content-driven websites where images are frequently updated or vary per page.
  - **Examples:** Blog post hero images, author avatars, and product images in an e-commerce site

XI) Blazor Guidelines

XI.1) Overview

Blazor is a .NET front-end web framework that enables both server-side rendering and client-side interactivity within a unified programming model. It is component-based, with components typically written as Razor markup pages using the **.razor** file extension.

In this section, I will discuss the relevance of Blazor in the context of our company's projects and share key techniques and best practices I have learned through my experiences. There also exists a website iteration which uses the blazor framework and it can be found on this repository.

XI.2) Prerequisites

Before diving into Blazor, developers are expected to have a foundational understanding of the following technologies: **HTML, CSS, C# and .NET**.

XI.3) Version

At the time of writing this documentation, the latest version of Blazor is part of ASP.NET Core in .NET 9.0. Please note that the information provided here is subject to change as newer versions of Blazor and .NET are being released. Developers are encouraged to stay updated with the official Microsoft documentation for the most accurate and current information.

XI.4) Blazor Web Assembly vs. Server

For starters, it is essential to understand the difference between Blazor WebAssembly and Blazor Server. Both provide a component-based architecture for building web applications, but they differ significantly in how they render and handle interactions. Below is a comparison:

- **Blazor WebAssembly:** Blazor WebAssembly allows for fully client-side applications, where the application is downloaded to the user's browser and runs directly on WebAssembly. This approach eliminates the need for a continuous server connection, making it suitable for scenarios where offline functionality or low-latency performance is required. The initial load time can be higher since the entire application is downloaded, but once loaded, it provides a seamless and responsive user experience.

- **Blazor Server:** Blazor Server, on the other hand, relies on server-side rendering where the UI interactions are handled on the server over a real-time SignalR connection. This model enables interactive server-side rendering (interactive SSR), allowing a rich user experience similar to a client-side app without the need for extensive API development. The server prerenders the page content, sending the HTML UI to the client as quickly as possible. Event handlers for UI controls are processed on the server, ensuring a responsive feel for users while maintaining centralized server control over the application logic. However, this approach requires a consistent server connection to function properly.

XI.5) Components and Razor Syntax

Blazor applications are built using Razor components, which are commonly referred to as simply "components." A component represents a reusable UI element, such as a page, dialog, or data entry form. These components are .NET C# classes compiled into .NET assemblies, making them powerful and flexible building blocks for your application.

Razor syntax is typically used to write these components, combining HTML markup with C# code for seamless client-side UI logic and composition. Razor files, identified by the **.razor** file extension, enable developers to design dynamic and interactive web applications with ease. Below is an example of a simple Razor component:

```
@page "/welcome"
<PageTitle>Welcome!</PageTitle>
<h1>Welcome to Blazor!</h1>
<p>@welcomeMessage</p>
@code {
    private string welcomeMessage = "We ❤ Blazor!";
}
```

At the top of the Razor file, directives like `@page` define the behavior and structure of the component. For example, `@page "/welcome"` specifies the route for the component.

The Razor file includes standard HTML markup to define how the UI is rendered. For instance, elements like `<h1>` and `<p>` render text and other content on the page.

The `@code` block contains the C# code defining the logic for the component. This can include variables, methods, event handlers, and parameters. The `@code` block can be separated into a `.cs` file for better organization, provided it shares the same namespace as its corresponding `.razor` file.

To maintain a clean and organized project structure, it is recommended to separate Razor files and their associated logic files as follows:

📁 - MainLayout

  📄 - MainLayout.razor

  📄 - MainLayout.razor.cs

XI.6) Entry Point

The entry point for a Blazor application, whether it's a **Blazor Server** or **WebAssembly** app, is defined in the **Program.cs** file, similar to how it is in a Console application. When the application starts, the system creates and runs a web host instance, using default configurations specific to web applications. The web host is responsible for managing the lifecycle of the Blazor app and setting up essential host-level services. These services include configuration, logging, dependency injection, and the HTTP server.

While this setup is typically boilerplate and often left unchanged, it forms the foundation of how the Blazor application is run. It is important to note that the **Program.cs** file structure and entry point setup are relevant for both **Blazor Server** and **Blazor WebAssembly** applications, though some specifics, such as the hosting model, may differ. In short, while the entry point concept is common for both, the implementation details and hosting models differ depending on the app type.

XI.7) Dependency Injection

In this section, we will discuss Dependency Injection (DI) in Blazor, particularly in relation to theming with MudBlazor. DI is a design pattern that allows a class to receive its dependencies from an external source, promoting loose coupling and making the application more modular, maintainable, and testable.

In Blazor, DI is commonly used to inject services, configurations, and other dependencies into components. Instead of managing dependencies internally, components request them from the DI container, centralizing their creation and enhancing the testability of the application. DI helps manage shared services, such as MudBlazor's theme providers, and other configurations, like user authentication or application settings, across the application without needing to pass them through each component manually.

Here's an example of how DI is configured in a Blazor application inside the **Program.cs** file using MudBlazor and custom services like ThemeService:

- **`builder.Services.AddRazorComponents().AddInteractiveServerComponents():`** adds interactive server components for Blazor.
- **`builder.Services.AddMudServices():`** adds MudBlazor services, such as the ThemeProvider.
- **`builder.Services.AddControllers():`** adds MVC controller services.
- **`builder.Services.AddSingleton<ThemeService>():`** registers ThemeService as a singleton service

Below are the types of dependency injection:

- Singleton
  - **Description:** A single instance of the service is created and shared across the entire application. It is useful for managing global state or services that don't need to change.
  - **Examples:** For theming, you would typically register a service like **ThemeService** as a singleton to ensure the theme settings are consistent across the application. `builder.Services.AddSingleton<IThemeService, ThemeService>  ();`
- Transient
  - **Description:** A new instance of the service is created every time it is requested. It is suitable for lightweight, stateless services.
  - **Examples:** If you have a service that calculates data on each request, register it as transient to ensure that each request gets a fresh instance.

    ```
    builder.Services.AddTransient<ICalculationService,
    CalculationService>();
    ```

- Scoped
  - **Description:** A service is created once per request or per scope, which in Blazor refers to the lifecycle of a component or a user interaction. It is useful for maintaining state within a single session.
  - **Examples:** If you need a service to maintain state during a component's lifecycle (e.g., a user session), register it as scoped. `builder.Services.AddScoped<IUserSessionService, UserSessionService>();`

In addition to injecting services, dependency injection (DI) can also be used to pass variables between components, whether from parent to child or vice versa. This is particularly useful for scenarios like implementing theming, where the theme settings (like dark or light mode) can be managed in a centralized service and shared across different components. Below is an example structure that illustrates how DI can be used to manage theming in a Blazor application.

📁 - Components

  📁 - MainLayout

    📄 - MainLayout.razor

    📄 - MainLayout.razor.cs

  📁 - Settings

    📄 - Settings.razor

    📄 - Settings.razor.cs

📁 - Services

  📄 - ThemeService.cs

📄 - Program.cs

First, we define the **ThemeService** class, which will manage the theme state (whether the app is in dark mode or not). This service will be placed in the **ThemeService.cs** file under the services folder.

```
namespace Aegis.Web.Services;
public class ThemeService
{
  public bool IsDarkMode { get; set; } = true;
}
```

Next, in **Program.cs**, we need to register ThemeService as a singleton because the theme's palette values are static and should be shared across the app. A singleton ensures that there is only one instance of ThemeService throughout the app.

```
using Aegis.Web.Services;
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<ThemeService>();
var app = builder.Build();
```

In the **Settings.razor** file, we add a button that will toggle the theme. This button will trigger the **ChangeTheme** method when clicked.

```
<MudButton @onclick="ChangeTheme">Toggle Theme</MudButton>
```

In the **Settings.razor.cs** file, we implement the **ChangeTheme** method. This method will change the theme by toggling the IsDarkMode property of **ThemeService**.

```
public class ChangeTheme
{
    ThemeService.IsDarkMode = !ThemeService.IsDarkMode;
}
```

In **MainLayout.razor**, we add the **MudThemeProvider** component. The **MudTheme-Provider** allows the theme to be dynamically updated based on the IsDarkMode property from the ThemeService. The theme is injected into the layout, and it automatically reflects the changes when toggled.

```
@inject ThemeService ThemeService
<MudThemeProvider @bind-
IsDarkMode="@ThemeService.IsDarkMode" Theme="CustomTheme">
    <!-- Other layout components go here -->
</MudThemeProvider>
```

By using dependency injection, the theme state becomes more manageable and can be shared across different components without the need for complex state passing or dependency management.

XII) Mud Blazor Guidelines

XII.1) Overview

MudBlazor is a Blazor component library based on Material Design principles. It is designed to be simple, lightweight, and easy to extend, making it an excellent choice for .NET developers. Unlike many other front-end libraries, MudBlazor relies on minimal JavaScript, aligning well with the Blazor framework's .NET-centric approach.

In .NET applications, MudBlazor is the recommended framework for front-end development within this organization. This section will cover the steps for implementing MudBlazor and its associated best practices.

XII.2) Prerequisites

Before diving into MudBlazor, developers are expected to have a foundational understanding of the following technologies: **HTML, CSS, C#, .NET, and Blazor**.

XII.3) Version

At the time of writing this documentation, the latest version of MudBlazor was **v8**. Note that the information provided here may become outdated as newer versions are released. Developers are encouraged to refer to the official MudBlazor documentation for the latest updates and detailed information.

XII.4) Relevance of Theming in MudBlazor

The importance of implementing themes on a website has been explained in the MUI section. However, there are notable differences in how theming is implemented in MudBlazor compared to MUI. In the following sections, I will discuss how theming can be applied in MudBlazor, including support for both light and dark modes.

XII.5) Default Themes and Palettes

MudBlazor provides default themes with predefined color schemes for most components. However, a key difference between MUI and MudBlazor lies in the flexibility of naming palettes.

- **MUI:** Developers can create custom palette names and define their own color schemes, providing extensive flexibility.
- **MudBlazor:** MudBlazor currently limits developers to a set of 66 predefined palette names. While you can customize these palettes by changing the colors or adding gradients, this can sometimes lead to semantic confusion, as the names cannot be altered.

Despite this limitation, MudBlazor's predefined palettes remain useful, as most websites adhere to a specific color scheme. Using these palettes ensures a cleaner implementation and aligns with Material Design principles. Below is an example which shows the different palettes available in MudBlazor:

Name	Type	Default Light	Default Dark	CSS Variable
Black	MudColor	rgba(39,44,52,1)	rgba(39,39,47,1)	–mud-palette-black
Primary	MudColor	rgba(89,74,226,1)	rgba(119,107,231,1)	–mud-palette-primary
Secondary-Darken	String	rgb(255,31,105)		–mud-palette-secondary-darken
Secondary-Lighten	String	rgb(255,102,15,3)		–mud-palette-lighten
HoverOpacity	Double	0.06		–mud-palette-hover-opacity

Once the palette has been implemented, the standard way to apply its colors is at the component level. This is achieved by using the **Color** property provided by MudBlazor components which is demonstrated based on the code snippet below:

```
<MudButton Color="Color.Primary">Primary Button</MudButton>
```

One efficient way to apply its colors to elements is demonstrated using TailwindCSS interoperability. This approach highlights one of MudBlazor's strengths compared to MUI.

- **MUI:** Developers must use MUI-specific components to access and apply the color palette.
- **MudBlazor:** Colors from the palette can be directly used within standard HTML tags by referencing the associated CSS variables.

This flexibility allows developers to seamlessly combine MudBlazor with other frameworks like TailwindCSS. Below is an example where a CSS variable from the MudBlazor palette is used to set the background color of an HTML element:

```
<div class="relative h-auto xl:h-screen 2xl:h-auto bg-[var(--mud-palette-
primary)]">
```

XII.6) MudBlazor Theme Customization

This section covers how to implement theming in the Blazor framework using MudBlazor. The **ThemeProvider** in MudBlazor is responsible for defining all aspects of the application's theme, including colors, shapes, sizes, and shadows. This section also explains the implementation of light and dark modes. More details can be found throughout this documentation.

To maintain clean and organized code, it is recommended to separate the Blazor files into two distinct parts:

- **MainLayout.razor:** Contains the Razor markup and HTML components for the layout.
- **MainLayout.razor.cs:** Contains the logic for managing the theme and palette.

In the **MainLayout.razor.cs:** file, we initialize the MudBlazor components by importing the necessary namespace:

```
using MudBlazor;
```

Next, we define the theme by creating a new **MudTheme** object. To support both light and dark modes, we define two separate palettes: **PaletteLight** and **PaletteDark**. Additionally, typography settings can be included to define custom fonts for the application.

```
public MudTheme CustomTheme = new()
  {
     PaletteLight = new PaletteLight()
  {
     Background = "#FFFFFF",
     Primary = "#0B286D",
  },
PaletteDark = new PaletteDark()
  {
     Background = "#000F33",
     Primary = "#0B1A3D",
  },
};
```

To manage the current theme mode (light or dark), declare a bool variable:

```
public bool IsDarkMode = false;
```

We also create a method for switching the state of the Boolean which is also interpreted as:

```
public void ThemeSwitcher()
{
   IsDarkMode = !IsDarkMode;
}
```

In the **MainLayout.razor** file, we use the MudThemeProvider component to apply the theme. Ensure the MudThemeProvider is placed at the top level, so all child components inherit the theme.

```
<MudThemeProvider @bind-IsDarkMode="@IsDarkMode" Theme="CustomTheme" />
```

To allow users to toggle between light and dark modes, we add a button that invokes the **ThemeSwitcher** method. This button can be placed in any component while accessing the theme switcher method using dependency injection which was discussed in the Blazor documentation.

```
<MudButton @onclick="ThemeSwitcher" />
```