Dashboard Web Components (React, Vue etc) for Workflow and Task Runs

Personal Information

<Redacted>

Motivation & Technical Skills

<Redacted>

Project Details

Brief Background

The Global Alliance for Genomics and Health

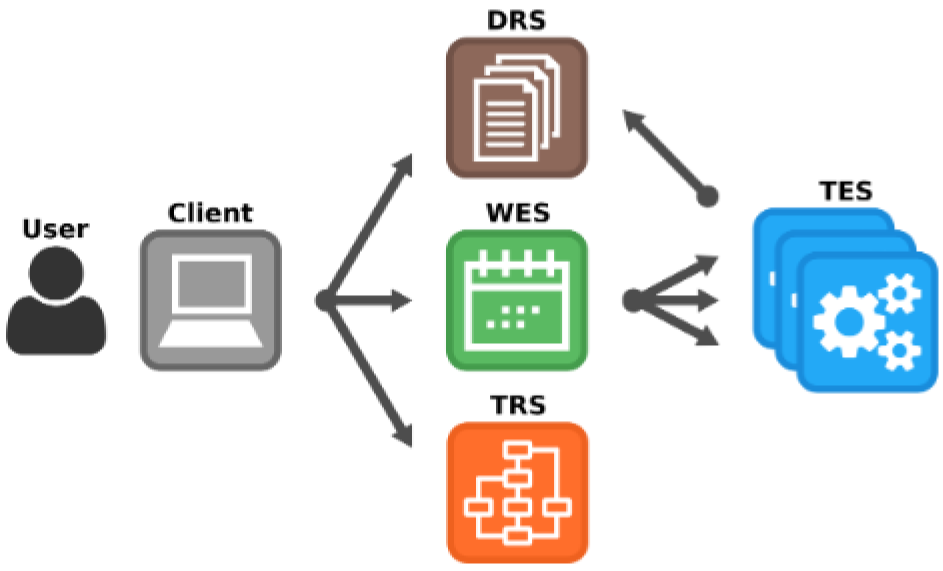
The mission of the Global Alliance for Genomics and Health (GA4GH) is to establish frameworks and standards for working with health-related and genomic data. Healthcare practitioners and researchers worldwide rely on this data to make informed decisions and improve treatments. Since the data, tools, and workflows involved are valuable resources, it's crucial that they can be located, accessed, and reused in a way that is interoperable. The GA4GH standards promote and maintain data principles across the biomedical data exchange and analysis field.

The GA4GH Cloud Work Stream

The GA4GH Cloud Workstream (CWS) aims to enable genomics and health communities to fully utilize modern cloud environments. Its primary focus is to enable the deployment of algorithms on cloud platforms by defining, sharing, and executing portable workflows through the development of standardized approaches. Standards under discussion include workflow definition languages, tool encapsulation, cloud-based task and workflow execution, and cloud-agnostic abstraction of secure data access.

There are currently 4 API standards that allow a person to:

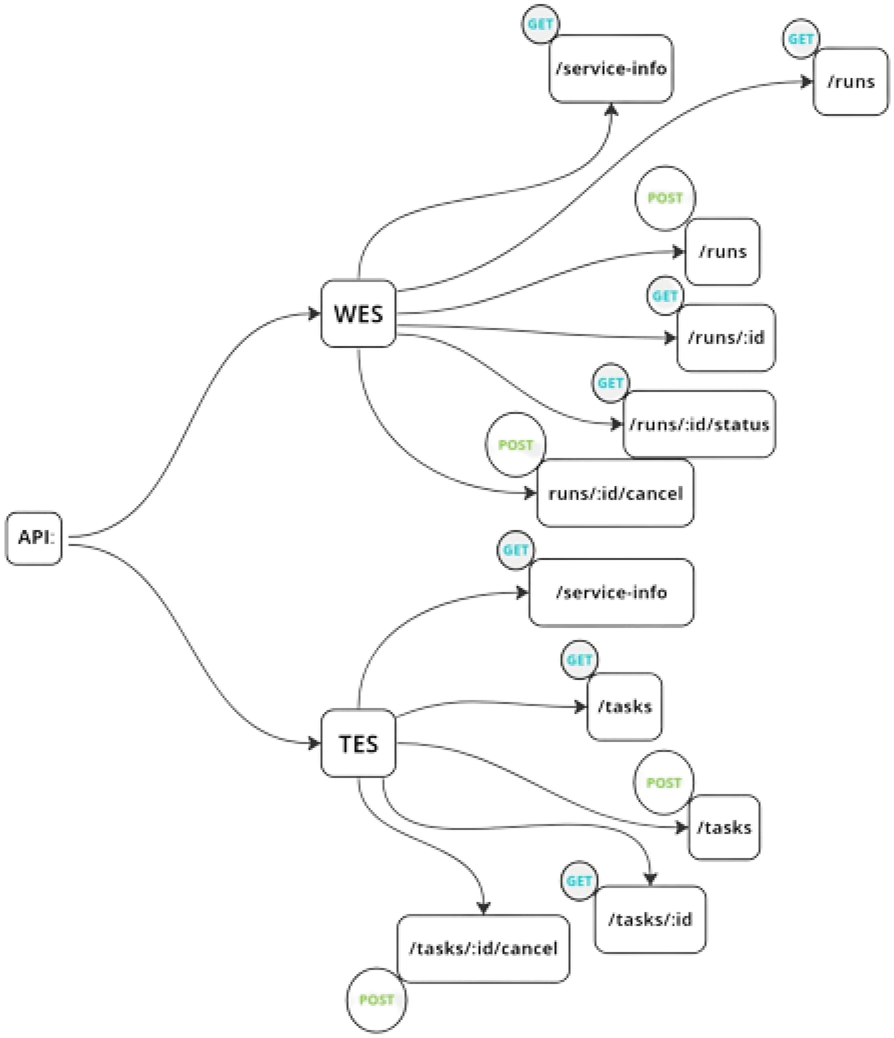
- Share tools/workflows (Tool Registry Service : TRS)
- Execute individual jobs on the cloud using standard API (Task Execution Service : TES)
- Run full workflows (for example : CWL, WDL, Snakemake, Nextflow etc) on execution platform (Workflow Execution Service : WES)
- Read/Write data objects across clouds in an agnostic way (Data Repository Service : DRS)



The Global Alliance for Genomics and Health (GA4GH) defines two web APIs for the execution of computational analysis workflows and individual containerized tasks, the Workflow Execution Service (WES) and the Task Execution Service (TES) API specifications, respectively.

The project involves the implementation of reusable Web Components Clients for the Workflow Execution Service (WES) and Task Execution Service (TES) API specifications, as defined by the Global Alliance for Genomics and Health (GA4GH). These components will enable users to trigger workflow and task runs and monitor their progress/status. The request payloads and API responses will be dynamically rendered based on the corresponding WES and TES OpenAPI schemas. The monitoring of available workflow runs and tasks will be displayed in paginated tables, with both basic and extended (uncollapsed) views.

The ultimate goal of the project is to integrate the Web Components into the Krini web portal, providing users with a user-friendly interface to trigger tasks and workflow runs and monitor their progress. The GA4GH's standards ensure the analysis workloads that are run through Krini, allowing for the data and tools to be found, accessed, and reused in an interoperable manner.



ELIXIR Cloud & AAI

ELIXIR Cloud is a cloud-based infrastructure provided by the European life science infrastructure ELIXIR (European Life Science Infrastructure for Biological Information). The ELIXIR Cloud provides a range of services and tools for storing, sharing, and analyzing large volumes of data generated by life science research. These services include computing, data storage, and data management capabilities.

AAI stands for "Authentication and Authorization Infrastructure." It is a set of tools and policies that enable users to securely access and use resources across different services and infrastructures. AAI is critical for enabling secure and seamless collaboration between researchers and institutions, as well as for ensuring the privacy and security of sensitive data. ELIXIR Cloud utilizes AAI to provide secure access to its resources and services.

Project-related section

- ### Overview of the project:

  The goal of this project is to develop reusable Web Components using UI frameworks implementing customizable design (such as by using Design Token etc) for a user-friendly interface for the GA4GH Workflow Execution Service (WES) and Task Execution Service (TES) APIs, which will be finally integrated on the Krini web portal. Its use-case is to support heavy genomic and biomedical computations through an intuitive and responsive interface.

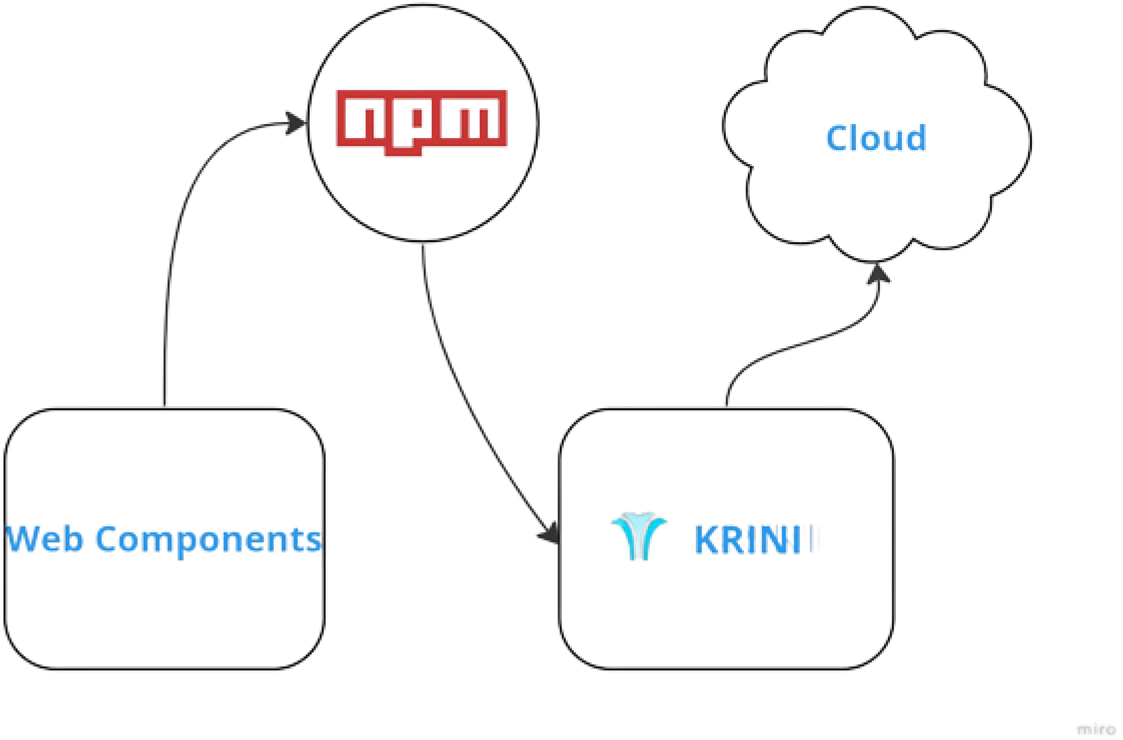
- ### Technical details

  The components can be built using FAST (Why Fast?) , a lightweight framework for creating Web Components in Open Web Components. Testing (IT and Unit both as explained in the implementation section) will be conducted using the Test runner or Testing Library, Jest. For design, Figma will be used to create a user-friendly and visually appealing interface. Finally, the components and API will be integrated into the Krini portal's React front-end for seamless usage.

- ### Implementation strategy

  The implementation strategy involves creating two portals for WES and TES, respectively. The first portal will handle the submission of workflow and task runs, while the second portal will be responsible for monitoring the runs. The monitoring portal will provide detailed information about each run, including its status and progress, and allow users to cancel runs if necessary.

While the two portals will be separate components, there will be some interaction between them. For example, the creation of a new run can be initiated from a modal within the monitoring portal, which will streamline the user experience and improve efficiency.



Finally, the Web Components will be packaged and published on npm to seamlessly integrate into the Krini portal.

User interface design

The user interface design for both the WES and TES APIs will have a similar underlying concept, with one modal for creating new runs and another for monitoring existing runs. The monitoring modal will feature a collapsible view of all runs, which will be populated by fetching data from the corresponding API. When a user clicks on a run, they will be able to view more detailed information about it, and they will have the ability to cancel or re-run the run if necessary.

The "re-run" and "create new run" features will each render a separate form, with the only difference being that the "re-run" form will be pre-populated with information from the selected run, while the "create new run" form will be blank. The form for submitting a new run will include fields for inputs such as attachment files, dropdown menus for selecting the workflow type, and parameters for the workflow.

Overall, the user interface design will be intuitive and user-friendly, with a clear layout and simple navigation. The forms and modals will be designed with the user's needs in mind, providing easy access to all necessary information and functionality

Note : Mockup Figma design is given in the implementation section.

- ## Potential challenges

One of the potential challenges that this project may face is ensuring compatibility across various frameworks, browsers, and devices. Even though FAST is intended to be flexible and lightweight, there may still be compatibility issues with different browser versions or devices that have varying screen sizes or resolutions. To ensure that the Web Components work effectively on all platforms, it may be necessary to conduct additional testing and make adjustments. Such as conducting manual testing in different browsers or if need be using libraries such as Puppeteer to fire up different browsers all at once to test the same component across various browsers.

Note : For this project manual testing should suffice, as the suite of components are discrete .

Implementor needs to be aware of integration techniques of FAST components with different frameworks , eg some adjustments need to be made, like using @microsoft/fast-react-wrapper for seamless integration of the component with applications based on React.The wrapper facilitates the consistent passing of data and events between React and FAST Elements etc.

```
//exporting Web Component class
import { FASTElement, customElement, html } from '@microsoft/fast-element';

@customElement({
  name: 'my-component',
  template: html`...`,
  styles:...
})
export class _MyComponent extends FASTElement {}

// In React module
import {_MyComponent} from './path'
import { provideReactWrapper } from '@microsoft/fast-react-wrapper';
import React from 'react';

export const MyComponent = provideReactWrapper(React).wrap(_MyComponent);
```

User story

As a researcher (end-user), I want to be able to monitor, run, cancel, and rerun large-scale computational analyses on genomics data using the GA4GH Workflow Execution Service (WES) and Task Execution Service (TES) APIs through a user-friendly and intuitive web interface. This will allow me to easily execute complex bioinformatics pipelines (selecting input files, choosing from pre-configured analysis workflows, and customizing the parameters and settings for my jobs) all without having to write code, using complex command-line tools or without having to worry about managing the underlying infrastructure and to track the progress of my jobs in real-time. Specifically, I would like to be able to submit WES and TES requests manually (through an easy and user-friendly interface), retrieve status and output information for my jobs, and be notified of any errors or failures that occur during the execution process. This will help me to accelerate my research and generate meaningful insights from large and complex genomic datasets.

Project Goal & Milestones

It is the aim of this project to:

- Develop a user-friendly web interface that allows researchers to submit WES and TES requests, customize job parameters, and track job progress in real-time.
- Ensure seamless integration with WES and TES APIs, with proper handling of errors and notifications to users.
- Implement responsive design and compatibility across different browsers and devices.
- Integrate the component on the Kirini web portal.
- Documentation

The following milestones can be defined:

Note : These just define the checklists of milestones that need to be completed, for more chronological and comprehensive detail look at the Gantt Chart and timeline.

1. **Milestone 1,SETUP (MS1) :** Setting up project and file structure, configuring the testing environment, and web-component compatibility.
   - ☐ Set up a well-organized file structure and configuration for the project like setting up Linter and Formatter.
   - ☐ Set up basic Design Tokens.
   - ☐ Configure the testing environment using industry-standard tools such as Jest.
   - ☐ Establish a strict contributor guideline for the project that can be followed by future contributors.

   Rationale : This so as to lay out a strict contributor guideline which can be further imposed on future contributors.

   Note : Milestones such as documentation and setup are project long milestones .

2. **Milestone 2, POST COMPONENT (MS2):** Creating Web Components for POST Workflow/ Task runs.
   - ☐ Create a basic form for submission for runs.

- ☐ Integrate API and create notifications or pop upfor bad API responses.
- ☐ Style using decided Design Tokens and Figma design.
- ☐ Optimize for Accessibility, ensuring that the component is accessible to users with disabilities by following accessibility guidelines and using semantic HTML.

Rationale : Choosing to create these components first due to the greater complexity of the get-runs Web Component According to the design implemented (shown below), and also due to the possibility of either whole or part of those components to be called for the get-runs component.

3. **Milestone 3, TESTING POST (MS3):** Testing Post WES and TES run components.
   - ☐ Test event handlers used such as onClick, onChange, etc.
   - ☐ Test component compatibility with different screen sizes and browsers, such as chrome, opera, firefox, and edge.
   - ☐ Check for optimization if needed, like Minimizing Dom manipulations.

Rationale : Due to the similarity among both the components, the implementation can be run simultaneously hence after a working model is up, testing will be much easier instead of going through the route of creating and testing both components one after another.

Note : Testing and production go hand in hand, milestones containing them will go on simultaneously.

4. **Milestone 4, NPM RELEASE 1 (MS4):** Releasing the first version of the npm package and integrating the components on the Kirini portal.
   - ☐ Creating the first build version of the npm package.
   - ☐ Testing the package locally before publishing, sorting any issues if needed.
   - ☐ Publishing the first version.

Rationale : Integrating the Web Components before the whole suite of components is done so as to ensure smooth integration of the components and troubleshoot future integration issues that might come up.

---

get-runs ⇒ |

| ⇒ Phase A

| ⇒ Phase B

Dividing runs components in 2 Phases, namely **Phase A** and **B** .

Phase A: This phase will focus on the basic functionality of the API, I.e. listing runs in a paginated and uncollapsible manner where single run deletion is possible. This phase ensures that all the requirements of the end user are met.

Phase B: This phase focuses on making the experience smoother and more flexible by adding rerun, and run creation by using post-run components with passed arguments.

---

5. **Milestone 5, PHASE A RUNS (MS5):** Creating and Testing the get-runs component (Phase A).
   - ☐ Creating an uncollapsible view for runs.
   - ☐ Adding pagination (this will be passed as an attribute where an implementer can define how many runs are required in on-page view) and delete functionality.
   - ☐ Styling.
6. **Milestone 6, PHASE A TESTING (MS6):** Testing get-runs component (PHASE A).
7. **Milestone 7, PHASE B RUNS (MS7):** implementing the ideas for Phase B of get-runs components.
   - ☐ Creating a rerun button, which will open a basic pre-populated form.
   - ☐ Adding a cloning run feature, if the form is selected, the run can be cloned and re-run simultaneously.
   - ☐ Styling.
   - ☐ Adding CI/CD pipeline.

8. **Milestone 8, PHASE B TESTING (MS8):** Testing the get-runs component (PHASE B) and publishing the next version of the npm package.
9. **Milestone 9, INTEGRATION (MS9):** Integrating the final Web Components to Kirini.

– STRETCH GOAL –

Adding bulk deletion of runs, this method will improve the productivity of researchers, where an option to select multiple runs from the table will be added, which will subsequently delete each run by calling the cancel API .

- ☐ Creating bulk deletion (Reference Gmail bulk deletion).
- ☐ Adding a spell checker such as cSpell to improve code's readability and maintainability.

Note : Many sub-milestones are repetitive, so it's only mentioned once and not every time it's needed.

Note : Sub milestones are listed with a checkbox but have an inherent numbering of format 1.2, 1.3, 1.4, and so- on.

Implementation Details

Note : Click here for an example repo I made of Web Components Using Fast . And here for hosted website (vanilla HTML) using those components.

PRE-Coding:

Google's official TypeScript style guide, which provides a set of guidelines and best practices for writing clean, consistent, and maintainable TypeScript code. It covers various aspects of TypeScript programming, including naming conventions, variable and function declarations, interfaces and classes, and code formatting. I'll be following this for best coding practices.

Note : A possible way is to configure Eslint using Google's style guide, to enforce certain standard conventions.

WHY FAST?

I chose Fast for this project because of its strong support for Design Tokens (explained below) and its focus on performance and scalability. Fast provides a robust set of APIs and tools for building web components that can be easily styled and customized using design tokens, which makes it an ideal choice for projects that require consistent branding and design. Additionally, Fast is optimized for speed and scalability, which is important for projects that need to handle large amounts of data or require high responsiveness. While other web component libraries like Lit and Polymer are also great choices, I believe that Fast's support for design tokens and its focus on performance make it the best fit for this project.

Styling:

In order to achieve a consistent and maintainable design for the web application, I will be utilizing Design Tokens to define the visual properties of the application. Design Tokens are a way to abstract design values, such as colors, typography, and spacing, into reusable variables that can be applied consistently across the application.

These Design Tokens will be based on the W3C standard, taking reference from LDS Design Tokens.
To establish a more scalable Design Tokens system, we can categorize them based on their characteristics, such as color, typography, spacing, shadows, and others. These categories can be treated as independent modules that export their respective design tokens, like:

```
// Define the default colors for the brand
export const colors = {
  description: 'These are the default colors used for brand identity',
  colors: {
    'color-primary': {
      value: '#0077B6', // Dark blue
      description: 'Primary color for the brand',
      type: 'color',
    },
    'color-secondary': {
      value: '#F7F7FF', // Off-white
      description: 'Secondary color for the brand',
      type: 'color',
    },
    'color-accent': {
      value: '#FFA500', // Orange
      description: 'Accent color for the brand',
      type: 'color',
    }
  },
};
```

Rationale : This systematic and modular way will not only help maintain and scale these design tokens easily but also in future if needed , an API can be created in similar format which will easily integrate with the previous code without breaking it.

Note : For further details look into my example implementation [here](#).

Note : The styles presented are only mockups, and are subject to change based on project requirements, as well as feedback and decisions from mentors.

```
Then we can easily export these Design Token to be used in different web
components.
/* Importing individual modules that export design tokens for colors,
spacing, shadows, and borderRadii … etc*/
import { colors } from './colors/colors.js';
import { spacing } from './spacing/spacing.js';
import { shadows } from './shadows/shadows.js';
import { borderRadii } from './borderRadii/borderRadii.js';

// Importing types for design token items and design tokens
import { DesignTokenItem, DesignTokens } from './designTokenInterface.js';

/* Creating an empty object called "tokens" to hold the combined design
tokens*/
const tokens: Record<string, string> = {};

/* Combining all the design tokens into a single "designTokens" object,
categorized by type*/
const designTokens: DesignTokens = {
  colors: colors.colors,
  spacing: spacing.spacing,
  shadows: shadows.shadows,
  borderRadii: borderRadii.borderRadii,
};
```

```
/* Looping through each category and token and adding them to the "tokens"
object with a key that combines the category and token name*/
for (const [category, token] of Object.entries(designTokens)) {
  for (const [tokenName, tokenValue] of Object.entries<DesignTokenItem>(
    token
  )) {
    tokens[`${category}-${tokenName}`] = tokenValue.value;
  }
}

// Exporting the combined design tokens as a single object
export { tokens };
```

This will emit tokens such which can be used in css of Web Components.

```
...
Background-color : var(colors-color-primary)
...
```

Making default styles modular and flexible:

Since the goal of the project is to make styling more flexible and modular, we can make the default styling flexible as well instead of going around each file and programmatically changing the CSS.

Variables export:

Using a ts file we can export a JSON with all the Design Tokens (Design Tokens as shown above) with default values this way we wouldn't need to hardcode styles in every module. In future if the default values are needed to be changed it would just need to be changed in a single file instead of each module.

PartialCSS:

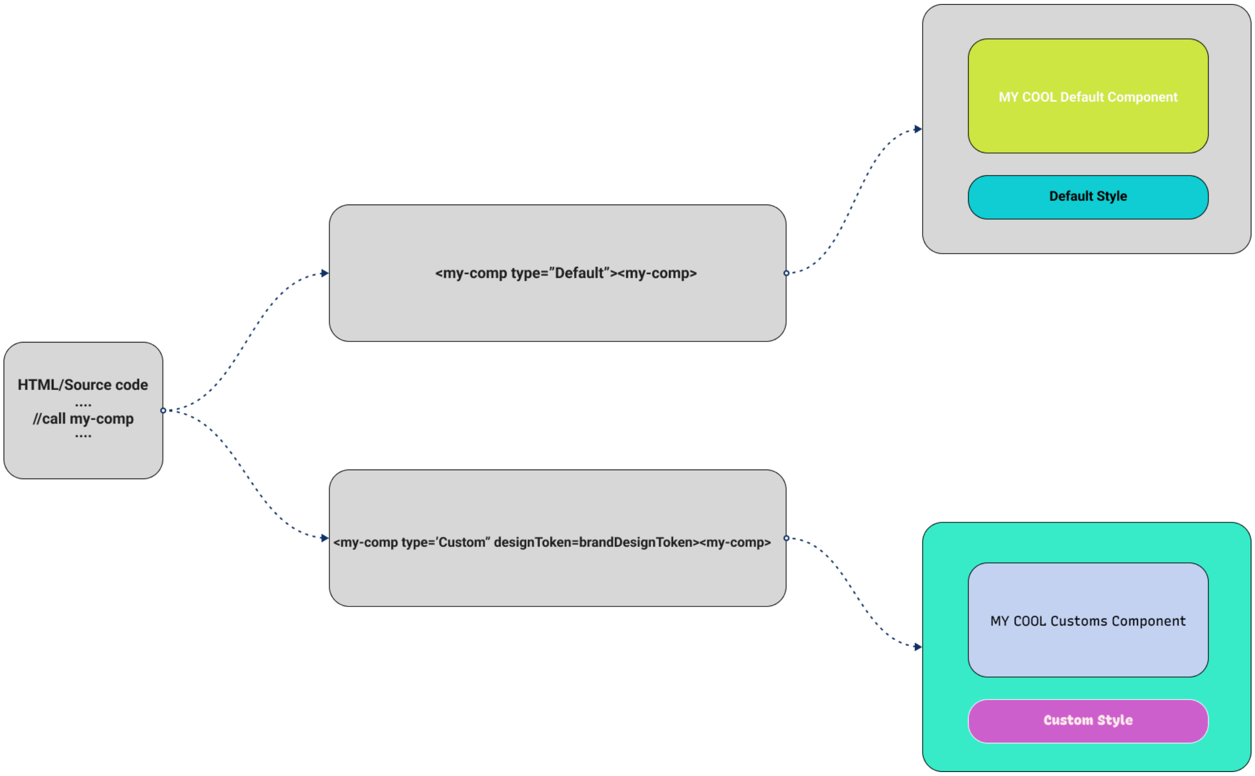
Using PartialCSS from @microsoft/fast-element we can define recurring styles so that we don’t have to apply the same css multiple times in different modules .

As @microsoft/fast-foundation provides support to emit a custom CSS property, this can help an implementer customize components based on their design system as mentioned in FAST’s documentation :

```
DesignToken.create<number>({
    name: "my-token",
    cssCustomPropertyName: "my-css-custom-property-name" // Emits to
--my-css-custom-property-name
});
```

Customizing the design according to a Brand’s Design System

These components initially will have ga4gh default CSS styling, i.e. the token will be registered with a default style (look at design below or as per the mentors guidance) , but they can be styled differently by passing a stringified JSON as an attribute while calling the component.


HTML/Source code
....
//call my-comp
....
<my-comp type="Default"><my-comp>
MY COOL Default Component
Default Style
<my-comp type='Custom" designToken=brandDesignToken><my-comp>
MY COOL Customs Component
Custom Style

Note : type = "Default" is not required to call a default component, it is an example and here it renders the word Default in the component.

Note :

HTML tags inherently do not support a lot of data types to be passed as attributes (see W3 documentation for more info), To pass Design Tokens as attributes to an HTML tag, we need to stringify them first. However, most modern frameworks like React and Angular have built-in functionality that automatically stringifies the attribute before passing them and converts them back to an object using JSON.parse(). It's important for implementers to be aware of the framework they're using. For vanilla HTML projects, the Design Tokens should be passed as a stringified JSON object.

```
....
<!-- when calling in vanilla HTML projects -->
<my-comp designToken='{"key":"value"}'></my-comp>
....
```

while for frameworks that support such conversion an object can be passed without errors.

```
  return (
    <div>
      //No need to send an already stringified JSON.
      <my-comp designToken={{key:value}}></my-comp>
    </div>
  );
```

Creating Design Token

```
import { .. } from '@microsoft/fast-element';

import { DesignToken } from '@microsoft/fast-foundation';

import { DefaultStyles } from '../defaultStyles.js';

const styles = css`
  .header {
    background-color: var(--bgPrimaryColor);
  }
`;

const template = html<NameTag>`...`;

@customElement({
    ..
})
export class NameTag extends FASTElement {
  @attr jsonData: string = '{}';
  @observable parsedJsonData?: parsedDataType =DefaultStyles

  //fetch the passed design token
  connectedCallback() {
    super.connectedCallback();
    this.parsedJsonData = JSON.parse(this.jsonData);
  }

  //Whenever the Design Tokens change , re write the css variables
  parsedJsonDataChanged() {
    //Store the component for which the Design Tokens need to be created.
    const currentComponent = this.$fastController.element;

    // Check if parsed JSON data is available.
    if (this.parsedJsonData) {
      // Iterate over each key-value pair in the parsed JSON data using
         Object.entries.

      Object.entries(this.parsedJsonData).forEach(([key, value]) => {
        // Create a new Design Token using the key.
        // Data type wouldn’t always be string, this is just an example.
        const newToken = DesignToken.create<string>(key);
        // Set the value of the new Design Token for the current component.
        newToken.setValueFor(currentComponent, value);
      });
    }
  }
}
```

Current issue with fast foundation’s Design Token:

As mentioned in open Issue #6602 of fast repo, on importing FAST’s Design Token (or cloning the repo which utilizes FAST’s design token), one might encounter the above mentioned issue, this issue can be solved by replacing lines 2252, 2254, and 2256 in node_modules/@microsoft/fast-foundation/dist/fast-foundation.d.ts with the following:

```
declare function create<T extends string | number | boolean | symbol |
any[] | Uint8Array | ({ createCSS?(): string; } & Record<PropertyKey, any>)
| null>(nameOrConfig: string): CSSDesignToken<T>;

declare function create<T extends string | number | boolean | symbol |
any[] | Uint8Array | ({ createCSS?(): string; } & Record<PropertyKey, any>)
| null>(nameOrConfig: Omit<DesignTokenConfiguration,
"cssCustomPropertyName"> | (DesignTokenConfiguration &
Record<"cssCustomPropertyName", string>)): CSSDesignToken<T>;

declare function create<T extends string | number | boolean | symbol |
any[] | Uint8Array | ({ createCSS?(): string; } & Record<PropertyKey, any>)
| null>(nameOrConfig: DesignTokenConfiguration &
Record<"cssCustomPropertyName", null>): DesignToken<T>;
```

Note : credit: hawkticehurst

WHY open-wc?

Open wc is an open-source framework for building Web Components that comes with utilities and testing, demoing, building, and publishing Web Components. Its native support for FAST as a base library will help produce better production-grade code and reduce future coding time.

Building:

GET Component:

When implementing the get-runs component, it's important to consider the needs of the end user. The component will consist of two phases, as explained above, Phase A (Basics) and Phase B (Complete).

The layout of the component will be designed with usability in mind. The list view will be paginated to avoid overwhelming the user with too much information at once. Each run will be represented by a tile that displays the most essential information about the run, such as the run ID, status, and a delete button. Clicking on the tile will expand it to show additional details about the run, such as the input data and processing results.

The buttons will have callback functions to invoke their handle functions, re-fetching the API and rendering the API data again. This becomes simple using the attr and observable decorator of Fast. These Ts functions will then also need to be tested (as explained below).

To help the user find the desired runs more easily, the component will include basic filtering and searching capabilities. The user will be able to filter the list by run status (e.g. completed, failed, running) or search for a specific run by ID. This will make it easier for the user to quickly find the runs they are interested in.

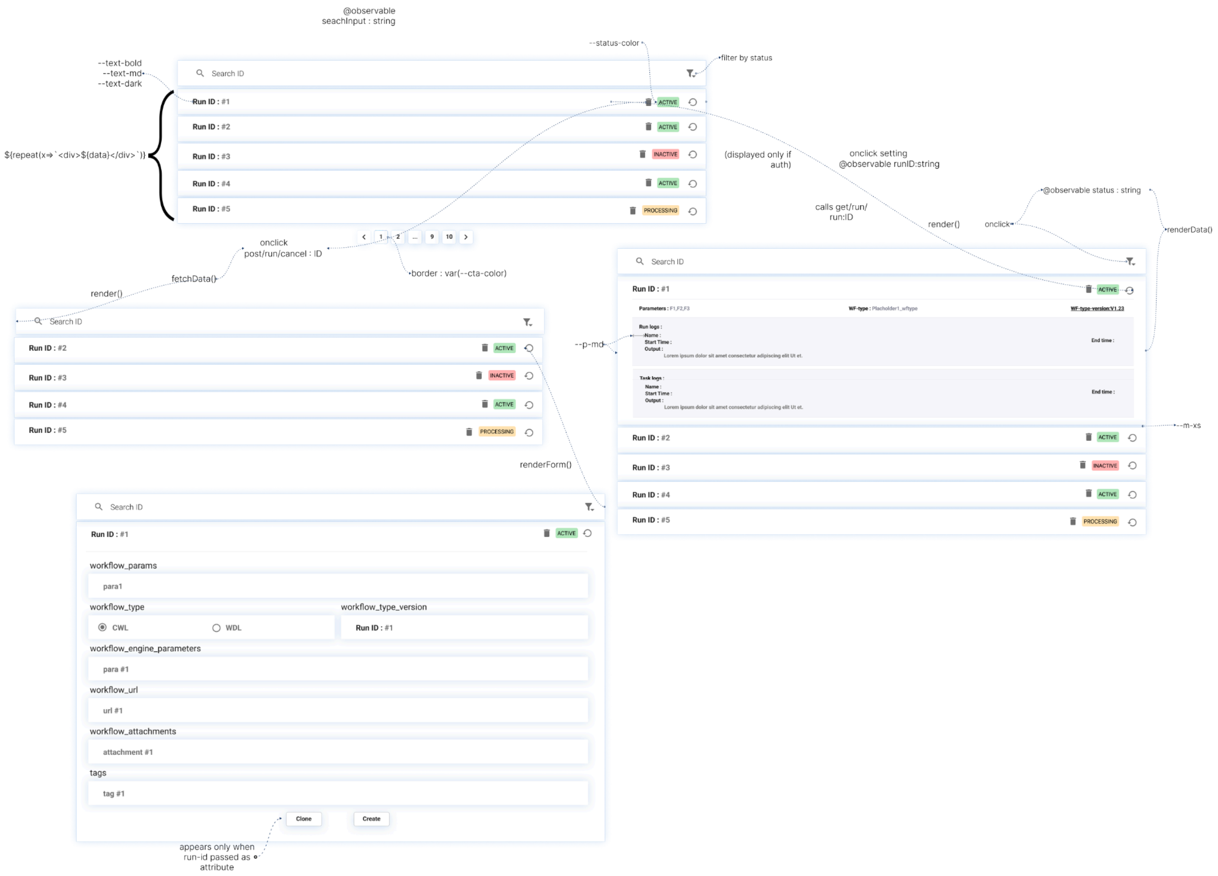
Delete feature will be subsequently added which on selection will use the selected run id and call cancel API.

```
import { html, css } from '@microsoft/fast-element';

const styles = css``;

const template = html<MyFastStyledComp>`
  <template>
    <button @click="${this.handleSort}">Sort</button>
    // paginated data, data="${this.data}"
  </template>
`;

export class MyFastStyledComp extends FASTElement {
  @observable data: any = {};

  // You need to define fetchDATA and setDATA functions
  fetchDATA() {
    // Implementation goes here
  }

  setDATA() {
    // Implementation goes here
  }
  connectedCallback() {
    // You need to call fetchDATA and setDATA functions to load data
    this.fetchDATA();
    this.setDATA();
  }
}

customElements.define('my-fast-styled-comp', MyFastStyledComp);
```

Note: For simple examples of list view look into my example implementation here or visit this vanilla HTML website to preview my implementation.

Upon clicking the "re-run" button, a form will appear that will render the post-run form and populate it with the current data. The form will include two buttons: one for submitting the re-run with modified data, and the other for cloning and re-running the task. This will result in two runs - one with updated values and another with the previous data. As there is no native re-run API available, the task will need to be canceled before it can be re-run.

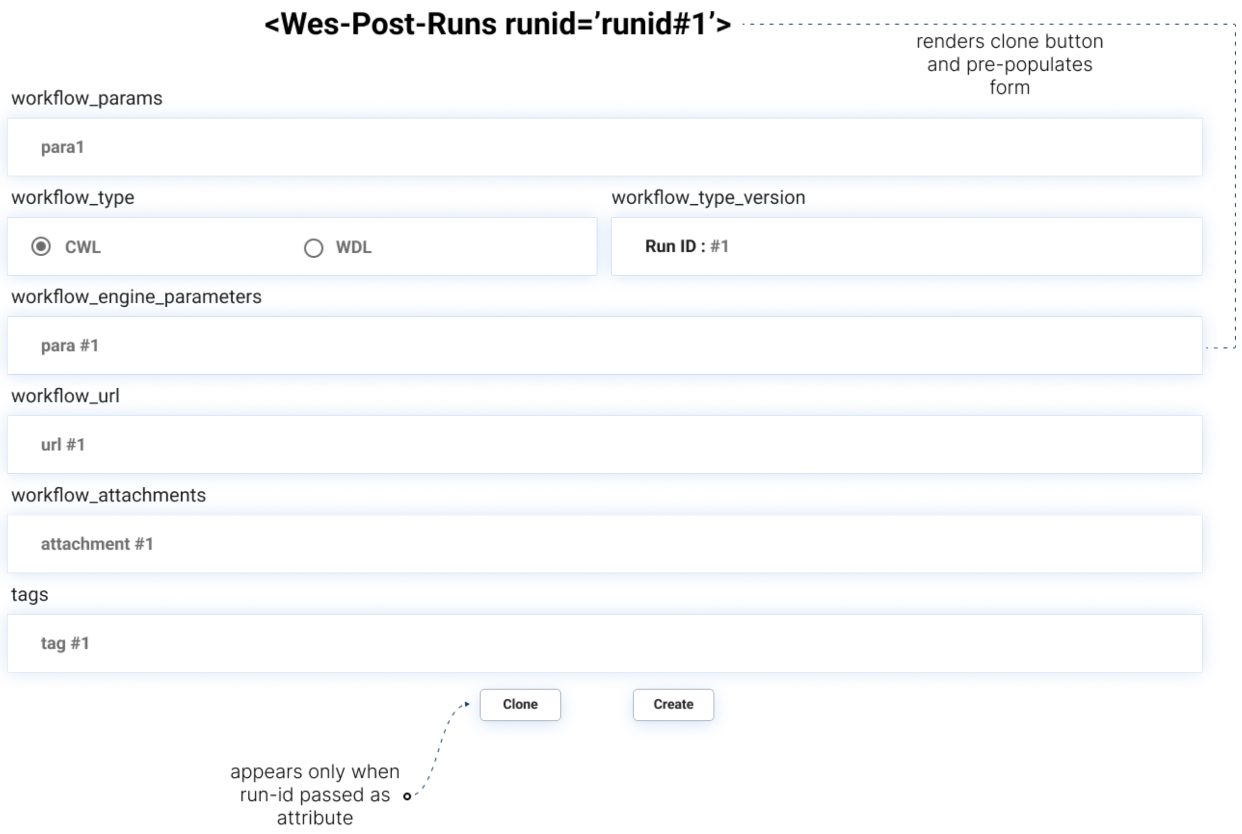
<Wes-Get-Runs>



For a better view click here.

POST Component:

The purpose of this component is to enable the posting of data to initiate a run. This will be achieved by means of a form that will capture the necessary inputs. Depending on the parameters passed, there will be two buttons that will automatically populate the form. In certain cases, a clone button will also be displayed, which not only creates a new run but also duplicates the previous run.



<Wes-Post-Runs runid='runid#1'>

renders clone button and pre-populates form

workflow_params: para1

workflow_type: ● CWL ○ WDL

workflow_type_version: Run ID : #1

workflow_engine_parameters: para #1

workflow_url: url #1

workflow_attachments: attachment #1

tags: tag #1

appears only when run-id passed as attribute

For a better view click here

Note: For simple examples of data binding / forms look into my example implementation here or visit this vanilla HTML website to preview my implementation.

Testing:

Using @web/test-runner for testing the web component. Using the standard testing practices and optimized for open-wc (as recommended by them in their docs) , Web Components will be thoroughly checked, hence making them robust and industry ready.

Look into my example testing here.

Unit Testing

The testing strategy for this component will consist of Unit testing as well as Integration testing. Unit testing to test each of the component's functions and methods in isolation. This will ensure that each function works as intended and will help prevent regressions as the component evolves.

```
import { ...} from '@microsoft/fast-element';
import { DesignToken } from '@microsoft/fast-foundation';
import { tokens } from '../../design-tokens/designToken.js';

const template: ViewTemplate<MyUnitTestedComponent> = html`
  <div class="container">
    <label for="name-input">Enter your name:</label>
    <input
      id="name-input"
      type="text"
      class="input-field"
      value.bind="${x => x.inputValue}"
      @input="${x => x.handleChange()}"
    />
    <button class="submit-button" @click="${x => x.handleClick()}">
      Submit
    </button>
    <div class="output-text">Hello, ${x => x.displayValue}!</div>
    <div class="flex abs">Styled Tested:Unit</div>
  </div>
`;

const styles = css`
  ....
`;

@customElement({
  name: 'my-unit-tested-component',
  template,
  styles,
})
export class MyUnitTestedComponent extends FASTElement {
  @observable displayValue: string = '';

  @observable inputValue: string = '';

  // Implementing Design Token . . .
  //handling value change
  handleChange() {
    this.inputValue = (
      this.shadowRoot?.getElementById('name-input') as HTMLInputElement
    ).value;
  }

  handleClick() {
    this.displayValue = this.inputValue;
  }
}
```

In Unit testing , We will have a test file which will test all the possible branches of the Dom tree and functional call to see if everything meets the expectation .

```
import { fixture, html, expect } from '@open-wc/testing';
import { MyItTestedComponent } from '../../src/index.js';

window.customElements.define('my-it-tested-component',
MyItTestedComponent);

// Use the "describe" function to group the tests
describe('MyItTestedComponent', () => {

  // Testing counting button
  it('should increment count when clicking the "+" button', async () =>
    const el = await fixture<MyItTestedComponent>(
      html`<my-it-tested-component></my-it-tested-component>`
    );
    //selecting the button
    const button = el.shadowRoot?.querySelector('.btn:first-child');
    const initialCount = el.count;

    //clicking the button and expecting the count to increase
    if (button instanceof HTMLElement) {
      button.click();
    }
    expect(el.count).to.equal(initialCount + 1);
  });
  it('should decrement count when clicking the "-" button', async () => {
    const el = await fixture<MyItTestedComponent>(
      html`<my-it-tested-component></my-it-tested-component>`
    );
    const button = el.shadowRoot?.querySelector('.btn:last-child');
    const initialCount = el.count;

    if (button instanceof HTMLElement) {
      button.click();
    }

    expect(el.count).to.equal(initialCount - 1);
  });
});
```

Click for more info.

Integration Testing

For integration testing, I will test how the component interacts with other components in the system. I will create a suite of tests that will check how the component behaves under different conditions, such as when certain input values are provided or when the user performs certain actions. These tests will help ensure that the component integrates correctly with the rest of the system and that any interactions between components work as expected.

```
import { expect, fixture, html } from '@open-wc/testing';
import { MyItTestedComponent } from '../../src/index.js';

window.customElements.define('my-it-tested-component',
MyItTestedComponent);

describe('MyItTestedComponent', () => {
  let component: MyItTestedComponent;

  beforeEach(async () => {
    component = await fixture(
      html`<my-it-tested-component></my-it-tested-component>`
    );
  });
  it('renders a heading and two buttons', async () => {

    // Find the heading and buttons in the component's shadow DOM
    const heading = component.shadowRoot!.querySelector('.heading');
    const buttons = component.shadowRoot!.querySelectorAll('button');

    // Use testing assertions to check if the heading and buttons exist
    expect(heading).to.exist;
    expect(buttons.length).to.equal(2);
  });

  it('increments and decrements count when buttons are clicked', async ()
=> {

  // Find the increment and decrement buttons in the component's shadow DOM
    const incrementButton = component.shadowRoot!.querySelector(
      '.btn:nth-of-type(1)'
    ) as HTMLButtonElement;
    const decrementButton = component.shadowRoot!.querySelector(
      '.btn:nth-of-type(2)'
    ) as HTMLButtonElement;

    // Using testing assertion to check if the button work as intended
    expect(component.count).to.equal(0);
    incrementButton.click();
    expect(component.count).to.equal(1);
    decrementButton.click();
    expect(component.count).to.equal(0);
  });
});
```

Click here for more info

Linting:

Eslint and Prettier will be used for linting and formatting the code and analyzing code for potential errors, bugs, and other issues to ensure that the code is written in a consistent, readable, and maintainable manner.

Google's typescript linting and formatting configurations can be leveraged as an example or adopted for this project.

Documentation :

Documenting the components is an essential part of making a library usable and user-friendly. To ensure that our library is well-documented, we will establish a comprehensive documentation strategy that covers all the components.

The documentation will follow a consistent format, such as Markdown, to ensure that it is easy to read and understand. It will not only describe how to use the components and their attributes, but will also include examples of different use cases and scenarios to make the documentation more practical and applicable. Additionally, the documentation will provide guidelines and best practices for using the components and styling them, listing out all the Design Tokens that can be swapped for custom values .

Post production:

CI/CD (Continuous Integration and Continuous Delivery/Deployment) pipelines are a set of processes and tools that allow for automated building, testing, and deployment of software applications. The pipeline consists of various stages, each with a specific function, including building and testing.

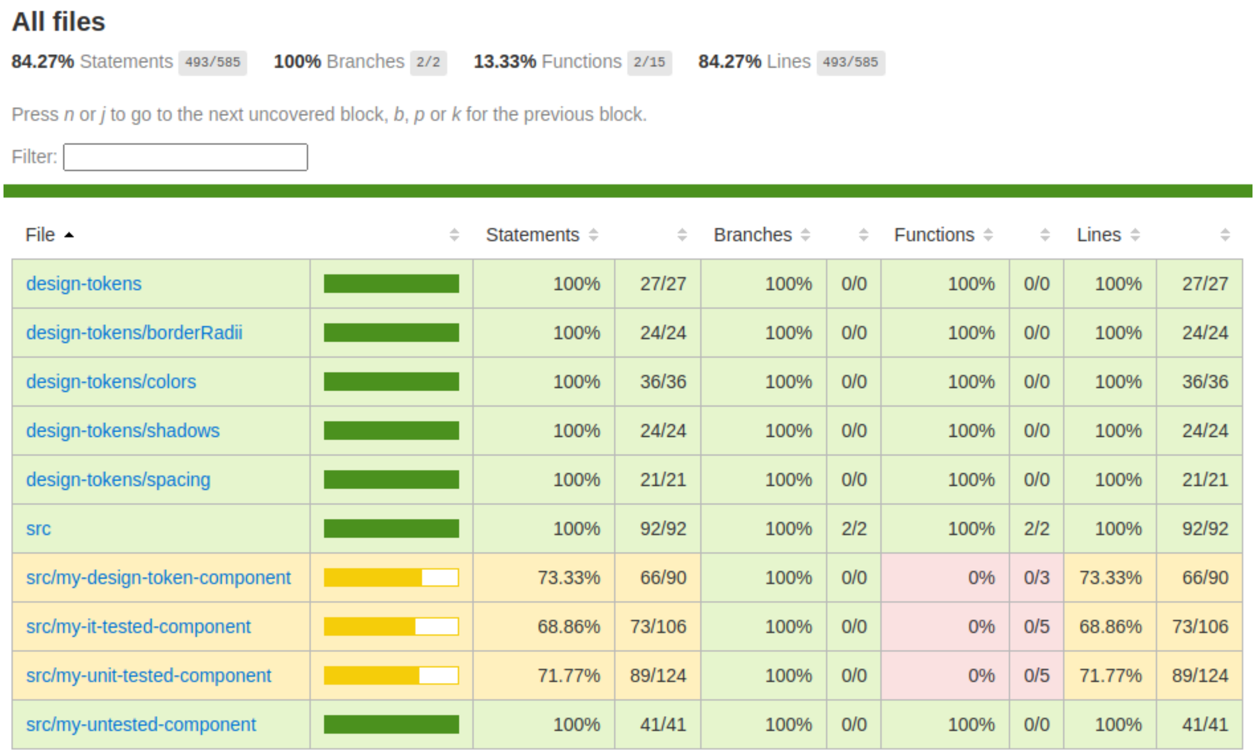
Upon discussion with mentors I will decide upon which CI/CD pipeline to choose for our project such as GitHub Actions, Travis CI, Jenkins etc.

– Stretch Goal –

A CI/CD pipeline can be implemented that automatically generates a code coverage report for each pull request and hosts it as a static HTML page. This would simplify future merges by providing developers with an easy way to assess the test coverage of their code changes.

To achieve this, we can use Istanbul, a tool that generates code coverage reports when the --coverage flag is enabled. Istanbul instruments the code to track which lines are executed during a test run and generates reports in various formats such as HTML, text, or JSON.

With this pipeline in place, we can ensure that all pull requests have sufficient test coverage, improving the overall quality of the codebase and making it easier to maintain and extend over time.



All files

84.27% Statements 493/585 **100%** Branches 2/2 **13.33%** Functions 2/15 **84.27%** Lines 493/585

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

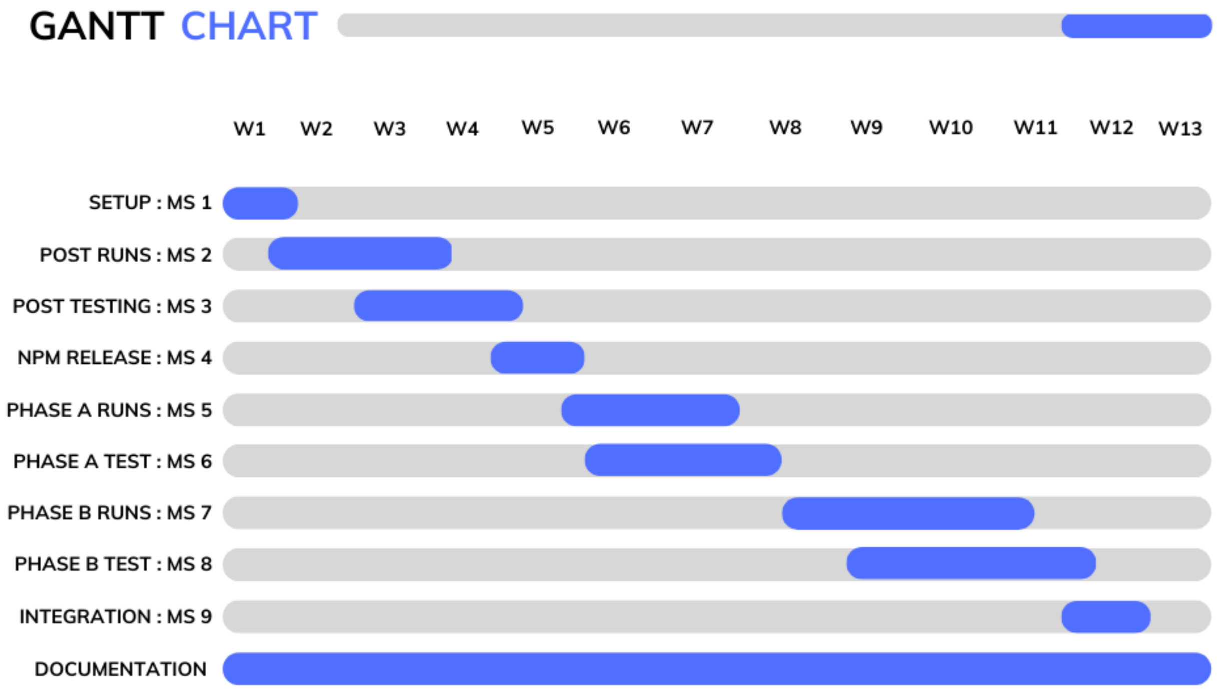
File ▲		Statements		Branches		Functions		Lines	
design-tokens	🟩	100%	27/27	100%	0/0	100%	0/0	100%	27/27
design-tokens/borderRadii	🟩	100%	24/24	100%	0/0	100%	0/0	100%	24/24
design-tokens/colors	🟩	100%	36/36	100%	0/0	100%	0/0	100%	36/36
design-tokens/shadows	🟩	100%	24/24	100%	0/0	100%	0/0	100%	24/24
design-tokens/spacing	🟩	100%	21/21	100%	0/0	100%	0/0	100%	21/21
src	🟩	100%	92/92	100%	2/2	100%	2/2	100%	92/92
src/my-design-token-component	🟨	73.33%	66/90	100%	0/0	0%	0/3	73.33%	66/90
src/my-it-tested-component	🟨	68.86%	73/106	100%	0/0	0%	0/5	68.86%	73/106
src/my-unit-tested-component	🟨	71.77%	89/124	100%	0/0	0%	0/5	71.77%	89/124
src/my-untested-component	🟩	100%	41/41	100%	0/0	100%	0/0	100%	41/41

Contingency Plan:

In case of severe health issues or adverse conditions out of my hand, my first step would be to communicate openly and honestly with my mentors. I will discuss and determine the best course of action together. If required, I will put in extra effort to complete all tasks within the allotted time frame or continue working until all deliverables are met to everyone's satisfaction if need be, even after the submission period of GSoC.

Timeline

Note : The project's timeline for completing code modules is not following a strict chronological order. Instead, it is structured to prioritize the needs of the end users. The project goals have been divided into segments with the end user's perspective in mind, allowing the completion of the most important components first.



- **Community bonding:** May 04 - May 28

  During this time, I will communicate with the mentors, discuss suggestions, and standard practices (going through google's style guide), get more familiar and acquainted with modules/tech needed and build up the repo, so as to streamline the process and build a base for the project.

- **Week 1:** May 29 - June 04

  Completing ***MS1,*** project setup (~ 2 days)

  Working on ***MS2.1: basic form*** and ***MS2.2 : error response***, setting up for WES. (~ rest)

- **Week 2:** June 05 - June 11

  Completing ***MS2.3: styling*** for WES (~ 3 days)

  Completing ***MS2.4: accessibility optimization*** for WES (~ rest)

- **Week 3:** June 12 - June 18

  Completing ***MS2.1***, ***MS2.2***, ***MS2.3, MS2.4 : same for TES*** (~ 5 days)

  Setting up testing, Starting basic tests and completing ***MS3.1: unit tests*** (~ rest)

- **Week 4:** June 19 - June 25

  Completing ***MS3: IT testing*** (~ 3 days)

  Integrating the component locally on Krini, completing ***MS4.1*** and ***MS4.2*** (~ rest)

- **Week 5:** June 26 - July 02 (PHASE A for WES)

  Releasing the first npm version completing ***MS4.3*** (~ 2 days)

  Implementation for ***MS5.1: uncollapsible view*** (~ 2 days)

Completing ***MS5.2: pagination*** (~ rest)

- **Week 6:** July 03 - July 09 (PHASE A for TES)

  Completing ***MS5.3: styling*** (~ 2 days)

  Completing ***MS5.1: TES uncollapsible view*** (~ 3 days)

  Completing ***MS5.2: TES pagination*** (~ rest)

- **Week 7:** July 10 - July 16 **(Phase 1 evaluation)**

  MS5.3:TES styling (~ 2 days)

  MS6: phase A testing (~ rest)

- **Week 8:** July 17 - July 23

  Completing ***MS7.1: rerun feature*** (~ 4 days)

  Completing ***MS7.2: clone feature*** (~ rest)

- **Week 9:** July 24 - July 30

  Completing ***MS7.3*** and finishing up with WES GET component (~ 3 days)

  Starting Phase B for get-runs TES , Completing ***MS7.1*** and ***MS7.2*** (~ rest)

- **Week 10:** July 31 - Aug 06

  Completing ***MS7.3*** and finishing up with WES GET component (~ 3 days)

Starting Phase B for get-runs TES , Completing ***MS7.1*** and ***MS7.2*** (~ rest)

- **Week 11:** Aug 07 - Aug 13

  MS7.4: CI/CD pipeline (~ 3 days)

  MS8: testing (~ rest)

- **Week 12:** Aug 14 - Aug 20

  MS8: testing (~ 2 days)

  MS9: Integrating with Krini (~ 2 day)

  Completing Documentation (~rest)

- **Week 13:** Aug 21 - Aug 28 **(Submission)**

  Buffer week, discussing with mentors and tweaking if necessary.