Framework Face Off

Jonatan Valen, Daniel Nyvoll & Vegard Vasset

December 11, 2023

Abstract

In our comparative study titled "Framework Face Off," we conducted an analysis of the three front-end frameworks: React, Vue, and Svelte. This study aimed to determine their suitability for developing a weather application, evaluating aspects such as ease of integration, performance, documentation, community support, and learning curve. React, with its robust ecosystem and strong community, excels in dynamic data handling. Vue offers ease of use and a friendly learning curve, while Svelte stands out for its efficient performance and less boilerplate code. Our findings provide vital insights into each framework's capabilities, aiding developers in making informed choices for their web development projects.

1 Introduction

In the dynamic world of web development, choosing the right front-end framework is crucial yet challenging, especially for beginners in the field. With the rapid evolution of the digital landscape, developers, particularly those inexperienced with front-end frameworks, face a challenging decision. In our study, which we conducted without any prior knowledge of front-end frameworks, we hypothesize that a thorough comparative analysis of prominent frameworks like React, Svelte, and Vue can clarify their strengths and weaknesses. This approach aims to aid beginners in making an informed decision about which framework to learn and use. The decision of selecting a front-end framework is critical. It profoundly influences the development process, application performance, and end-user satisfaction. As the web becomes increasingly integral to our daily lives, ensuring the optimal performance and usability of web applications is not just a technical concern but a necessity for enhancing digital experiences globally.

To address this, our project aims to conduct a comprehensive comparative study of React, Svelte, and Vue. We intend to evaluate these frameworks across crucial parameters like documentation, runtime performance, code quality, and cross-browser compatibility. Our approach involves developing a web application using each framework, adhering to test-driven development principles to ensure consistency and fairness. This methodical evaluation will not only compare these frameworks but also provide insights into their suitability for various development scenarios.

To expand on this, our project will also place significant emphasis on the subjective experience of the development process. As newcomers to the realm of front-end frameworks, we will document our

personal journey with React, Svelte, and Vue. By reflecting on the intuitiveness of the frameworks, the clarity of documentation, and the ease of overcoming challenges, we aim to provide a unique perspective on the learning curve and developer ergonomics of each framework. This personal narrative will complement our objective metrics, offering a comprehensive view of the frameworks from the standpoint of novice developers. This experiential component could be invaluable for beginners who are evaluating these tools for their own use.
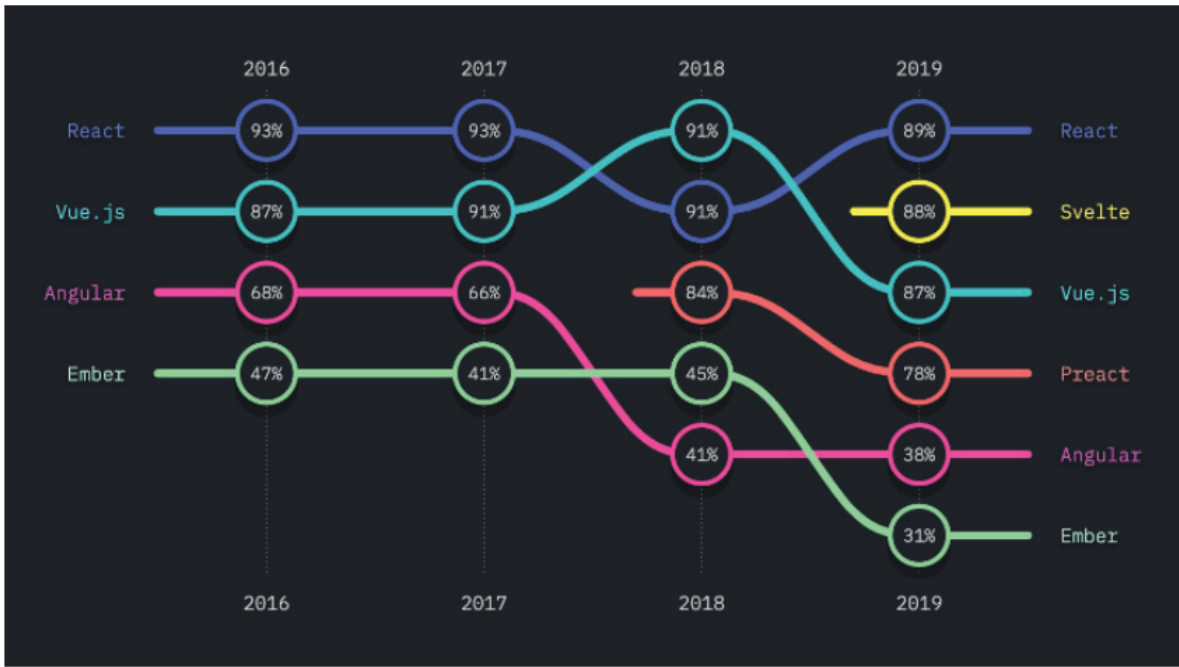
2   Background

Figure 1: Representing the popularity trends of various web development frameworks over a period from 2016 to 2019.

React, known for its robust ecosystem and backing by Facebook, has consistently held a significant share of popularity, while Vue.js has emerged as a strong contender with its approachable learning curve and incremental adoptability. Meanwhile, Svelte, the newest among them, presents an innovative approach by shifting work to compile time, leading to faster runtime performances.

To ensure a consistent standard of functionality across the web pages developed using each framework, Test-Driven Development (TDD) was employed as an aspect of our comparative analysis. This approach guaranteed that, although different frameworks were used, each implementation adhered to the same functional specifications. By integrating TDD into our workflow, we were able to provide a more objective measure of each framework's capabilities, taking into account not only the end product but also the development process itself.

In our analysis of front-end frameworks, we selected a weather application as our use case, primarily because it allows for practical experimentation with public APIs. This choice provided a tangible

means to assess and compare the capabilities of various frameworks in handling real-time data, user interface dynamics, and responsiveness. The weather application scenario was particularly suited for this comparison as it involved fetching and displaying data from external sources, a common requirement in modern web development. This setup enabled us to evaluate how different the frameworks React, Svelte and Vue.js manage data binding, state management, and overall performance. Additionally, the integration with public APIs served as an excellent benchmark for understanding the ease of use, flexibility, and scalability offered by each framework, making our comparison comprehensive and relevant to current industry trends.

Central to our approach was the utilization of client-side operations, primarily fetching data from various APIs. The decision to not set up a dedicated backend allowed us to concentrate on the frontend functionalities, exploring how each framework handles tasks like API calls, data processing, and DOM manipulation. By simulating real-world scenarios where frontend applications interact with existing APIs, we were able to assess the efficiency, ease of integration, and responsiveness of each framework in handling client-side operations.

2.1   Related Work

Several studies and articles have compared the leading JavaScript frameworks, offering valuable insights for developers and researchers alike. The paper "Evaluating the Performance of Web Rendering Technologies Based on JavaScript: Angular, React, and Vue" presents a thorough comparative analysis focusing on performance metrics such as build size, time to interactive, and DOM manipulation time. It underscores Vue's superior performance in DOM manipulation and React's efficiency in interactivity, while noting Angular's larger bundle size. This study serves as a benchmark for understanding the technical strengths and limitations of these frameworks. (IEEE et al.)

An article on the DEV Community website further expands on this comparison by examining factors such as popularity, community support, and real-world applications of React, Vue, Angular, and Svelte (Wichmann). It highlights the extensive communities and resources available for React and Angular, the moderate popularity of Vue, and the emerging presence of Svelte in the industry. The article also delves into performance assessments, aligning with the findings of the aforementioned study by highlighting the efficiency of Vue and Svelte.

Another critical aspect of these frameworks is the learning curve, as discussed in the Medium article by Faulkner Project, "Vue vs React vs Svelte" (Faulkner). This article provides a practical perspective for new developers, emphasizing the user-friendly nature of Vue and Svelte compared to the more complex React. It also discusses real-world applications of these technologies, providing examples of companies that utilize them, thereby offering insights into their practicality in various industry settings.

Knowledge about different frontend frameworks is a critical asset for developers, preparing them to make discerning choices in a professional context and to deploy these tools adeptly in the creation of web solutions.

2.2 Frameworks

2.2.1 About Vue:

Vue.js is a modern, progressive JavaScript framework used for building user interfaces and single-page applications. It's known for its simplicity and gentle learning curve, which have contributed to a growing popularity among developers. Vue.js is approachable, performant, versatile, and designed to be incrementally adoptable. The core library is focused on the view layer only but can be easily integrated and scaled for complex applications using additional libraries and tooling (Copes, 2018).

The framework facilitates the development process by offering a component-based architecture, allowing developers to create encapsulated and reusable components. Vue.js uses a declarative rendering model, where the developer writes templates in HTML that are bound to the underlying component state using a reactive and composable data binding system.

The design and implementation of Vue.js involve a balance of technical details and engineering trade-offs, which are well-documented and provide developers with a solid foundation for building applications. With Vue.js, components' data and DOM are reactively and efficiently updated, making state management straightforward and the development process more intuitive (HcySun, 2023, 1-574).

2.2.2 About React:

React, a JavaScript library developed by Facebook, has significantly transformed web application development. Its primary strength lies in its component-based architecture, which enables developers to build self-contained, state-managing components. This approach not only facilitates the creation of complex and reusable user interfaces but also as stated by (Novac et al. 1) that; "It is easy to build interactive applications with React.js because it makes views for each state of the application. When data is changed the framework React.js will perform the update and render the appropriate components."

The library's declarative nature is another important part of its functionality. In React, developers define the desired state of the user interface, leaving the responsibility of actual DOM updates to React itself even though the paper detecting code smells in React-based Webapps says that "React beginners usually use plain vanilla JavaScript to direct access a HTML DOM element, which can cause inconsistencies between React's virtual DOM and the real DOM." (Ferreira and Valente 5). This method ensures efficient rendering processes, as React compares the component's current state with its previous state and updates only the parts that have changed, not the entire page. Such an approach is particularly advantageous for the maintainability of large-scale applications, as it significantly enhances performance and streamlines the development workflow.

Moreover, React's lifecycle management provides developers with fine control over the behavior of components. This control is crucial for managing the various states in the application lifecycle, from initialization to updating and unmounting, allowing developers to execute specific code at each stage. The comprehensive ecosystem surrounding React, which includes tools like React Native for mobile app development and numerous third-party libraries, further extends its capabilities.

2.2.3 About Svelte:

Svelte, a groundbreaking framework in web development, was introduced by Rich Harris in 2016. It challenges traditional web development norms by shifting much of the work typically done in the browser to compile time. This results in highly optimized JavaScript code, setting Svelte apart from other frameworks like React and Vue. Svelte's approach is component-based, but it diverges notably in its operational philosophy, especially with its direct updates to the DOM instead of using a virtual DOM, leading to faster rendering and more efficient memory usage.

According to Bhardwaz and Godha, "Svelte also offers a more intuitive and straightforward way to build web applications. Instead of using a complex set of APIs, Svelte's syntax is easy to understand and follow, making it a great choice for developers who are new to web development." (Bhardwaz and Godha, 2023, 1). Its syntax is cleaner and more intuitive, requiring less boilerplate code, which enhances code readability and maintainability. Svelte's reactive programming model is integrated into its language, streamlining state management and reducing reliance on external libraries.

Building upon the strengths of Svelte, SvelteKit ("Introduction • Docs • SvelteKit") is a framework thats designed to create more complex and scalable web applications. SvelteKit extends Svelte's capabilities, providing server-side rendering, file-based routing, and full-stack capabilities. This makes SvelteKit a more complete solution for modern web development, allowing for the creation of both static sites and server-rendered applications with the same ease and efficiency that Svelte offers.

3 Design and Implementation

3.1 Test Driven Development

The implementation of Test-Driven Development (TDD) in our project commenced with the creation of unit tests before the actual coding of the application. We used testing frameworks like Jest and Vitest to write tests for core functionalities. For instance, we wrote tests to ensure the correct conversion of location names to geographic coordinates, a fundamental feature for our weather application. These tests validated the functionality of our geocoding utility and weather API services, ensuring that, for example, inputting 'New York' would correctly yield the expected latitude and longitude. Our project's testing phase was designed to maintain identical logic across Svelte, React, and Vue, allowing us to compare each developer's experience in achieving the required test outcomes. The journey to passing the tests provided insights into the user-friendliness of each framework, the quality and accessibility of their documentation, the richness of resources available, and the level of community support. These factors collectively offered a measure of each framework's approachability and practicality from a developer's perspective, influencing our overall evaluation of the tools.

3.1.1 Svelte Example Test

```
1  import { it, expect } from 'vitest';
2  import { geocodeTextToLatLon, celciusTemp, fetchWeatherData } from './Weather';
```

```
3   it('geocodes a location name to latitude and longitude', async () => {
4
5       const { lat, lon } = await geocodeTextToLatLon('New York');
6
7       expect(lat).toBeCloseTo(40.7128, 2);
8       expect(lon).toBeCloseTo(-74.0060, 2);
9     });
```

Utilizes Vitest for testing. Focuses on component behavior and interaction with mocked functions.

3.1.2 React Example Test

```
1   import { render, screen } from '@testing-library/react';
2   import App from './App';
3   import WeatherComponent from './WeatherComponent';
4
5   test('renders learn react link', () => {
6     render(<App />);
7     const linkElement = screen.getByText(/learn react/i);
8     expect(linkElement).toBeInTheDocument();
9   });
10
11  jest.mock('./GeocodingUtils', () => ({
12      geocodeTextToLatLon: async (locationName) => {
13        if (locationName === 'New York') {
14          return { lat: 40.7128, lon: -74.0060 };
15        } else {
16          throw new Error('Invalid location');
17        }
18      },
19    }));
```

Leverages Jest and React Testing Library. Tests user interface and component integration with mocked dependencies.

3.1.3 Vue Example Test

```
1   import { tempConverter, getCoordinatesFromName, fetchWeatherData } from
        '../services/weatherAPI'
2
3   describe('getCoordinatesFromName', () => {
4       it('converts a location name to latitude and longitude', async () => {
5           const locationName = 'New York';
6           const expectedLat = 40.7128;
7           const expectedLon = -74.0060;
8
9           const { lat, lon } = await getCoordinatesFromName(locationName);
```

```
10
11
12      expect(lat*1).toBeCloseTo(expectedLat, 2);
13      expect(lon*1).toBeCloseTo(expectedLon, 2);
14    });
```

Employs Vue Test Utils and Jest. Concentrates on user event simulation and component state management.

3.2 React Implementation

The React application is constructed with a focus on modular design, leveraging the power of React's component-based architecture. Each component is tailored to fulfill specific functionalities, contributing to a better user experience. The application effectively utilizes state management through React hooks and engages with external APIs, ensuring dynamic content delivery and interaction.

Routing and Application Structure:

```
1   const PlaceholderApp = () => {
2       return <p>This app section is not yet implemented.</p>;
3     };
4
5     const App = () => {
6      return (
7        <Router>
8          <div>
9            <Header />
10           <Routes>
11             <Route path="/weather" element={<WeatherApp />} />
12             <Route path="/active-transportation" element={<ActiveTransportationApp
                    />} />
13             <Route path="/car" element={<PlaceholderApp />} />
14             <Route path="/public-transport" element={<PlaceholderApp />} />
15             <Route path="/" element={<p>Welcome to your app. Please select a section
                    from the menu.</p>} />
16           </Routes>
17         </div>
18       </Router>
19     );
20   };
21
22   export default App;
```

The code snippet above is part of the App.js file. It illustrates the App component, which is responsible for the applications routing structure. It utilizes the Router component to define the applications navigation. Inside the Router, various Route components map URL paths to corresponding React components. For instance, the path /weather is associated with the WeatherApp component, while

/active-transportation is linked to the Active-transportation component. There are also placeholder routes, which serve as temporary stubs for future implementation, denoted by the PlaceHolderApp component.

React root rendering:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

This code snippet is from the index.js file. It showcases the root rendering process of the React application. It uses React.StrictMode, which is a tool for highlighting potential problems in an application. It do not render any visible UI, but activates additional checks and warnings for its descendants. It is then passed to REACTDOM.createRoot, signifying the entry point of the application. This is where React elements are rendered into the DOM, with the root DOM node being provided by document.getElementByID('root'). This is what kickstarts the React application's rendering process.

Reactive System and State Management:

The application's use of React hooks across several components like LocationMap.jsx, SearchBar.jsx, and Weather App.jsx. It provides a modern approach to state management. This setup ensures a responsive and user-friendly interface, capable of efficiently handling state changes and asynchronous operations.

For instance, in SearchBar.jsx React's useState hook is employed to manage the state of the user's search input. This allows for an instantaneous and responsive feedback loop between the user input and the application's state. An example from the code illustrates this:

```
const SearchBar = ({ onSearch }) => {
    const [searchTerm, setSearchTerm] = useState("");

    const handleInputChange = (event) => {
      setSearchTerm(event.target.value);
    };

    const handleKeyPress = (event) => {
      if (event.key === "Enter") {
        onSearch();
      }
    };
```

In the code snippet above we have the following key elements: useState hook. This is used to declare the state variable "searchTerm" and its updater function "setSearchTerm". It is initialized with an empty string, and the state holds the current value of the search input. handleInputChange is an event handler function that updates the searchTerms state when with the value entered by the the user in the search input field handleKeyPress is another event handler that listens for the "Enter" key press to

trigger the search function passed as a prop.

The component is designed to be reactive and responsive to user input, utilizing the useState hook for management and providing a clean and intuitive interface for users to perform search queries. Interaction with External APIs:

```
1   const search = async () => {
2       setIsLoading(true);
3
4       const element = document.getElementsByClassName("cityInput");
5
6       if (element[0].value === "") {
7         setIsLoading(false);
8         return 0;
9       }
10
11      // Get coordinates using Mapbox's Geocoding
12      let geocodeUrl =
            `https://api.mapbox.com/geocoding/v5/mapbox.places/${element[0].value}.json?access_token=${map
13
14
15      try {
16        let geoResponse = await fetch(geocodeUrl);
17        let geoData = await geoResponse.json();
18        const [lon, lat] = geoData.features[0].geometry.coordinates;
19
20        setLongitude(lon);
21        setLatitude(lat);
22
23        let url =
            `https://api.openweathermap.org/data/2.5/weather?lat=${lat}&lon=${lon}&units=Metric&appid=${
24
25        let response = await fetch(url);
26        let data = await response.json();
27
28        console.log(data);
29
30        setTemperature(Math.floor(data.main.temp) + "c");
31        setLocation(data.name);
32        setHumidity(data.main.humidity + "%");
33        setWindSpeed(Math.floor(data.wind.speed) + " km/h");
34        setDescription(data.weather[0].description);
```

The application integrates with external APIs to enhance its feature set. In the code snippet above we see an example on how it is done in WeatherApp.jsx.

An asynchronous function search is defined to handle retrieval of location coordinates and weather data. The application first uses Mapbox's Geocoding API to convert a location input into geographical coordinates. It constructs a request URL with the user's input from the cityInput field. Upon receiving the response, the application parses the JSON to extract latitude and longitude coordinates, which are then used to set the application's state with setLongitude and setLatitude.

With the coordinates obtained from Mapbox, the application proceeds to fetch weather data from the OpenWeatherMap API. It constructs a new request URL, including the latitude and longitude, and specifies the desired units as metric. The resulting weather data is obtained asynchronously, and the application updates the state with various weather details such as temperature, location name, humidity, wind speed, and a description of the current weather.

Once the data is retrieved and the state is updated, these values are used to update the UI, providing users with real-time weather information for the queried location. This is done using all the "set" functions. When these functions are called with new data retrieved from the API, they trigger a re-render of the component.

3.3   Vue Implementation

Vue Router:

Implementing routing functionality in Vue was remarkably straightforward and intuitive, primarily due to the seamless integration and alignment of Vue Router with the core Vue.js framework. Vue Router's declarative routing configuration, which leverages a simple and clear JavaScript object syntax, made it particularly easy to define and manage routes. This approach, combined with Vue's component-based architecture, allowed for an efficient and hassle-free setup of navigational structures within the application. Additionally, the comprehensive and well-organized documentation of Vue Router provided excellent guidance, further simplifying the implementation process.

Here is a code snippet of the implementation of Vue Router.

```
import { createRouter, createWebHistory } from 'vue-router';


import WeatherPage from '@/components/WeatherPage.vue';
import TravelPage from '@/components/TravelPage.vue';



const routes = [
  {
    path: '/',
    redirect: '/weather'
  },
  {
    path: '/weather',
    component: WeatherPage
  },
  {
    path: '/travel',
    component: TravelPage
  },
];
const router = createRouter({
    history: createWebHistory(),
    routes,
  });
```

```
25
26
27  export default router;
```

Component-based Development:

The encapsulation of Vue components proved to be an intuitive and streamlined process, particularly advantageous for those new to frontend development. Components like SearchBar.vue or WeatherDisplay.vue function as standalone units, each with their dedicated HTML, CSS, and JavaScript. This encapsulation enables a focused approach on individual parts of the application, mitigating concerns about unintended interactions with other segments. The encapsulation was quickly recognized as beginner-friendly at the onset of development.

For example, WeatherDisplay.vue encapsulates the template, script, and style in a single file. The HTML structure, reactive data properties, methods, and component-specific styles are all consolidated, simplifying both development and troubleshooting.

The use of the 'scoped' attribute in the <script>-tag is notably advantageous. It guarantees that the defined CSS styles are confined to the component, avoiding accidental style conflicts.

The facility to transfer data and methods via props and to issue events lends to the high reusability and maintainability of the components. Components like SearchBar.vue can be confidently utilized across various parts of the application, ensuring consistent and independent functionality.

The organization and modularity offered by Vue's component system greatly reduce the complexity in developing an intricate web application, enabling a foundational understanding of each part of the application before their integration, which is substantially beneficial for learning frontend development effectively.

Reactive Data System:

The reactive data system in Vue.js offers a robust and streamlined mechanism for binding data to the user interface, facilitating the development of dynamic interfaces that update in real time. This system represents a paradigm shift away from the manual manipulation of the DOM, automating UI updates in response to state changes within the application.

In the context of the WeatherDisplay.vue component, the employment of Vue's reactive properties significantly simplifies the presentation of weather data. Through the declaration of a reactive data property within the component's data function, a direct linkage is established between the data and the template. Consequently, any modifications to the weather data are promptly and automatically mirrored in the UI, negating the need for extraneous code. This not only enhances efficiency but also underscores Vue's capability to streamline front-end development, particularly for those new to the framework.

Code snippets from WeatherDisplay.vue component, displaying current weather, demonstrating the use of reactive data binding:

```
1  <template>
```

```
2    <div v-if="weatherData">
3      <!-- Current weather display section -->
4      <div class="current-weather">
5        <h3>Current Weather for {{ placeName }}</h3>
6        <p><strong>Temperature:</strong> {{
             weatherData.properties.timeseries[0].data.instant.details.air_temperature
             }}C</p>
7        <p><strong>Wind:</strong> {{
             weatherData.properties.timeseries[0].data.instant.details.wind_speed }}
             m/s</p>
8        <p><strong>Humidity:</strong> {{
             weatherData.properties.timeseries[0].data.instant.details.relative_humidity
             }}</p>
9      </div>
10
11
12
13  <script>
14  export default {
15    props: ['weatherData', 'placeName', 'searchQuery'],
```

The placeName interpolates the placeName data property into the heading. Whenever placeName
changes, the content within the header tag will automatically update to reflect the new value.

The temperature, wind speed, and humidity are also dynamically rendered using data binding.

Moreover, the v-if directive provides a declarative approach to controlling the visibility of DOM elements. It is used to conditionally render the container div for the weather data, based on the presence
and truthiness of weatherData. This technique enhances the interface's reactivity and user experience
by only presenting relevant information when data is available. The effectiveness of directives like
v-if in managing the DOM underscores the strengths of Vue.js in creating interactive and reactive web
applications.

Working with external APIs:

Engaging with external APIs within a Vue.js context proved to be an enriching educational experience, particularly in understanding and managing asynchronous operations. Initially, concepts such
as asynchronous programming and API response handling appeared complex. However, the utilization of axios, a promise-based HTTP client, considerably simplified these tasks.

A practical application of this was demonstrated in the getCoordinatesFromName function, designed
to transform a location name into geographic coordinates through the OpenStreetMap API. This function showcases the streamlined handling of HTTP requests facilitated by axios.

```
1  export async function getCoordinatesFromName(locationName) {
2      console.log('Fetching coordinates for:', locationName);
3      const baseUrl = 'https://nominatim.openstreetmap.org/search';
4        try {
5          const response = await Axios.get(baseUrl, {
6            params: {
```

```
 7        q: locationName,
 8        format: 'json',
 9        limit: 1
10      }
11    });
12    if (response.data && response.data.length > 0) {
13      const data = response.data[0];
14      return { lat: data.lat, lon: data.lon };
15    }
16  } catch (error) {
17    console.error("Error fetching coordinates:", error);
18  }
19  return null;
20 }
```

The use of async/await syntax in conjunction with axios greatly enhanced the readability and maintainability of the code. It allowed for writing asynchronous functions in a style reminiscent of synchronous code, thus simplifying the understanding and implementation. The getCoordinatesFromName function needed only the location name to perform its operation, handling the complexities of asynchronous requests and error management internally.

Following the acquisition of coordinates, the fetchWeatherData function was employed to retrieve weather data.

These functionalities were integrated into a searchLocationByName function, providing a better and user-friendly method for fetching and displaying weather data based on location names. This process underscored the importance of logical structuring in API calls and illustrated the effectiveness of JavaScript's advanced features in external API interactions. Such experiences highlighted the significance of adeptly handling asynchronous operations and leveraged the capabilities of modern JavaScript to enhance the application's functionality.

```
 1 export async function searchLocationByName(locationName) {
 2    const coordinates = await getCoordinatesFromName(locationName);
 3    if (coordinates) {
 4
 5        return await fetchWeatherData(coordinates.lat, coordinates.lon);
 6    } else {
 7        console.error("Couldn't get coordinates for the provided location.");
 8        return null;
 9    }
10  }
```

3.4   Svelte Implementation

The Svelte application encapsulates a modular design, mirroring the practices of component-based frameworks. In this Svelte-based project, each component is crafted to fulfill a discrete functional role, contributing to an integrated and seamless user experience. The application showcases a ro-

bust implementation of reactive state management, capitalizing on Svelte's innate reactivity and store capabilities.

```svelte
<script>
    import Menu from "$lib/Components/Menu.svelte";
    import { Hamburger } from 'svelte-hamburgers';
    import Infobox from "$lib/Components/Infobox.svelte";
    import Geocoding from "$lib/Weather/Geocoding.svelte";
    import Map from "$lib/Components/Map.svelte";
    import WeatherInfo from "$lib/Weather/WeatherInfo.svelte";
    // import { selectedLocationName } from "$lib/store.js";

        let open = false;

    </script>


    <body>
        <Hamburger bind:open --layer-height="2rem" --layer-width="10rem"/>
        <Menu bind:open />
        <Infobox>
            <Geocoding />
                <WeatherInfo/>
            <Map />
        </Infobox>
    </body>
```

The core of the application's routing is managed within the +page.svelte file, demonstrating Svelte's file-based routing mechanism. This approach is distinctly Svelte, where routes are implicitly defined through the file structure, streamlining the mapping of URL paths to Svelte components. The Menu.svelte component is responsible for navigating to different parts of the application, such as the weather view or the active transportation information section. The +page can be seen above

```js
import { LngLat } from 'mapbox-gl';
import { writable } from 'svelte/store';

export const selectedLocationName = writable('');
export const selectedCoordinates = writable(LngLat.convert([0, 0]));
export const temperature = writable(0);
export const windSpeed = writable(0);
export const map = writable();
```

The application's reactive system is powered by Svelte's reactive stores, as seen in the store.js file above. Svelte's reactivity is straightforward and less boilerplate-heavy compared to other frameworks, as any variable declared as writable in the store becomes reactive across the application. In Svelte, the $ symbol is used to denote a store value and to establish a reactive subscription to the store's changes. When the value of a store changes, any component that uses the $ syntax to refer to that store will automatically re-render to reflect the new value. The figure show the initialisation of the map by using the dollar sign:

```
1  onMount(() => {
2      mapboxgl.accessToken =
          'pk.eyJ1IjoiaW5mMzE5IiwiYSI6ImNsbjIwNGxtajA0OHMyaW5yZ2pwdjdzdmQifQ.dcde7lQq6RN9tl5V91lMjg';
3
4      $map = new mapboxgl.Map({
5        container: 'map',
6        style: 'mapbox://styles/inf319/cln2208t6031101nzbzvshgt1',
7        zoom: 12,
8      });
9    });
```

Svelte's architecture is distinguished by its lifecycle hooks, which provide developers with fine-grained control over the behavior of components during their lifecycle events. Among these, onMount is a particularly crucial lifecycle hook used to run code when a component is first introduced into the DOM as seen in the figure above. The onMount function is typically employed for initialization tasks such as fetching data, setting up subscriptions, or directly interacting with the DOM, tasks that should only be executed once when the component is created.

Interacting with external APIs is a critical feature of the application. The code snippet provided indicates a function that interfaces with the Mapbox API, searching for locations based on user input. The seamless integration of API calls within Svelte components is indicative of the framework's promise to keep things simple and concise. The fetchData function is shown in the code below.

```
1  <script>
2  async function fetchData() {
3      try {
4        const response = await
            fetch('https://api.met.no/weatherapi/locationforecast/2.0/compact?lat=${$selectedCoordinate
5
6        if (response.ok) {
7          const data = await response.json();
8          $temperature =
              data.properties.timeseries[0].data.instant.details.air_temperature;
9          $windSpeed = data.properties.timeseries[0].data.instant.details.wind_speed;
10         return data;
11       } else {
12         throw new Error('Failed to fetch data. Status: ${response.status}');
13       }
14     } catch (error) {
15       throw error;
16     }
17   }
18 </script>
```

Svelte's straightforward approach to incorporating asynchronous operations, showing how data fetched from an external weather API is utilized to update the reactive variables in the store, consequently refreshing the UI with new data. The Svelte syntax allows for co-locating markup, script, and style, which is visible in the comprehensive <script>and <style>, making the codebase more maintain-

able and approachable.

4 Evaluation

In the final analysis of our front-end framework comparison for a weather application, several results stand out. React's build time is the longest at 9.7 seconds, which might be because of its comprehensive ecosystem that includes numerous libraries and plugins. While this can increase the initial load time, it offers a wealth of resources for API integration and the ability to efficiently handle dynamic data updates a key aspect for real-time weather information.

Vue presents itself as a strong contender with the shortest build time of 4.9 seconds. This efficiency could reflect a more streamlined build process or a lighter framework footprint. Vue's performance advantage is evident during the initial development stages, contributing to rapid prototyping and early momentum in application development.

Svelte's build time stands at 6.44 seconds, positioning it between React and Vue. This intermediate value is reflective of Svelte's unique architectural approach, where much of the workload is shifted to compile time, resulting in optimized and concise final code. The Svelte store system, particularly its import/export functionality, plays a significant role in state management within the application, allowing for real-time updates across components without significant overhead.

These measures of build time and application progress, coupled with qualitative feedback from developers, paint a comprehensive picture of each framework's strengths and weaknesses. They reveal not just the technical capabilities, but also the practical implications for developer productivity and application performance.

As emerging developers at the confluence of technological innovation and environmental responsibility, our assessment of carbon footprint and sustainability within the landscape of front-end development frameworks is more than academic. it is an ethical imperative. Recognizing that the digital solutions we craft today have a tangible impact on tomorrow's world, we diligently compared the sustainability profiles of Svelte, React, and Vue. Our criteria for evaluation were twofold: ease of use, which influences the speed and efficiency of development, and the carbon footprint, particularly influenced by daily build times. Svelte, with its lean build process and highly efficient runtime performance, emerged as the most sustainable framework in our study. Its build time, at 6.44 seconds, suggests a lower daily energy consumption, which, when scaled across the industry, could contribute significantly to energy conservation efforts. Coupled with its user-friendly approach, Svelte stands as a testament to our commitment to a sustainable future, blending ease of use with an environmentally conscious development process.

4.1 Documentation and Learning Curve

In our study, we found React's documentation to be exceptionally comprehensive. Covering everything from basic concepts to advanced topics like hooks and context, it served as a vital resource in our learning process. The structured format and the inclusion of tutorials made it accessible, yet the
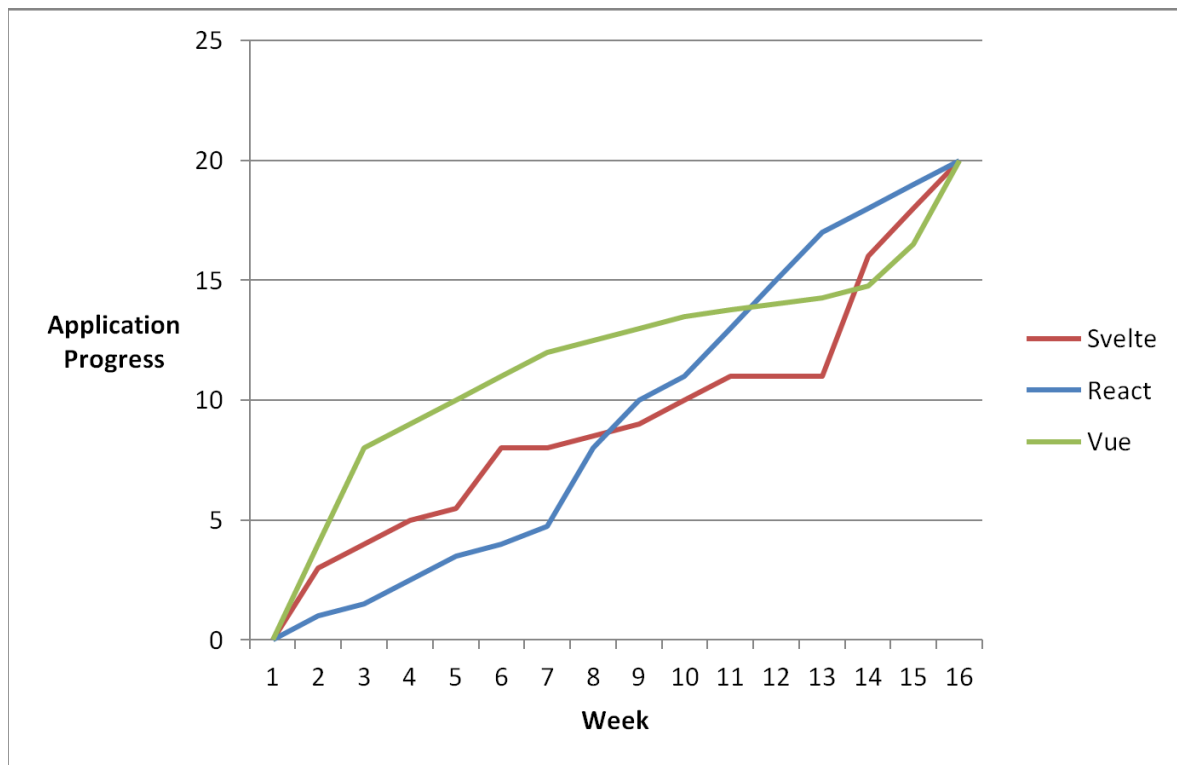
Figure 2: Shows application progress during the 16 week period.

sheer depth of information sometimes posed a challenge for us as beginners. We experienced a steep learning curve with React, particularly at the start. However, as we delved deeper, the framework's capabilities unfolded, offering us a more profound understanding of advanced frontend development concepts. The active community support was very helpful, providing us with additional guidance and resources.

Our engagement with Vue was marked by its notably user-friendly documentation. The clarity and conciseness of the guides helped us grasp core concepts with ease, and the practical examples were particularly beneficial. This aligns with the gentle learning curve we experienced with Vue. For beginners like us, Vue's intuitive component structure and declarative rendering approach made the learning process smoother and more straightforward. Additionally, the vibrant Vue community was a helpful resource, offering us a platform to seek assistance and share insights.

In our exploration of Svelte, the simplicity and directness of its documentation stood out. It efficiently covered the basics, allowing us to quickly understand the framework's essentials. The tutorials and examples, though less extensive in advanced topics compared to React or Vue, were straightforward and suited our initial learning phase. We found Svelte's learning curve to be moderated by the strong community support, especially on the Discord platform. This bustling hub was like a busy coffee shop for us, full of interactive learning opportunities and real-time problem-solving, which proved to be an invaluable aspect of our learning journey with Svelte.

4.2 Advantages and Disadvantages

4.2.1 React

In our comparative analysis, we found several advantages of using React. Its large ecosystem provides abundant libraries for API integration and UI components, which are particularly suitable for displaying weather data and maps. We also appreciated the strong community support, which offered ample resources and knowledge for troubleshooting API or Mapbox integration issues. Another significant advantage we noted was React's efficient handling of dynamic data updates, a crucial feature for real-time weather information display.

However, we encountered some disadvantages with React. Beginners might find the learning curve steep, especially when grasping JSX and component lifecycle, potentially delaying development. The framework's tendency towards more verbose setup might slow initial development, particularly when handling multiple API responses. Additionally, React's fast-paced changes could lead to frequent need for updates, adding to the maintenance workload, especially in the context of API and Mapbox integrations.

4.2.2 Vue

In our study, Vue demonstrated several advantages. We found its integration with various APIs and Mapbox straightforward, making it well-suited for modular weather application development. Its simplicity and readability significantly expedited our development process and eased maintenance. Vue's detailed documentation was also a notable advantage, providing extensive resources for best practices in API and map service integration.

On the downside, Vue's smaller market share presented some challenges. The limited examples specific to weather applications due to a smaller community were a noticeable disadvantage. We also found that the community size might result in fewer specialized resources for addressing complex API or Mapbox challenges. A particular complexity we observed in Vue was in the management of data between components that lack a direct parent-child relationship, which indicated that while props and events are effective for direct lineage communication, they are less so for components situated far apart in the hierarchy.

4.2.3 Svelte

Our examination of Svelte revealed several strengths. The framework's efficient code structure allowed for faster implementation of API calls and map integrations. We were impressed by Svelte's performance, particularly its capability for fast UI updates, which is essential in displaying real-time weather data. Additionally, its simplicity was a significant advantage, easing the learning curve and accelerating the development process.

Nevertheless, Svelte also has its drawbacks. Its smaller community and ecosystem limit the availability of resources and examples for weather application-specific challenges. Moreover, we found that

Svelte's maturity is less proven in complex, real-time data-driven applications, such as a weather app with mapping features, which could be a concern for more intricate projects.

5   Discussion

Framework	Performance	Learning curve	Ease Of Use	Popularity	Community	Virtual DOM	Mobile Support
React.js	Good	Steep	Moderate	High	Strong	Yes	Good
Vue.js	Good	Easy	Easy	High	Strong	Yes	Good
Svelte.js	Excellent	Moderate	Easy	Rising	Strong	NO	Good

Figure 3: Comparison table

The table above presents a comparison chart for the frontend frameworks we have been working with. It assesses them on performance, learning curve, ease of use, popularity, community support, virtual DOM usage, and mobile support. Svelte.js is noted for its excellent performance and ease of use, while React.js and Vue.js are marked as good performers with strong community support. All three frameworks provide good mobile support, but only React.js and Vue.js utilize a virtual DOM, an approach that Svelte.js does not adopt.

What have we learned?

Throughout this project, our understanding of frontend frameworks has deepened significantly. We've learned the importance of choosing the right tool for the job: Vue for its simplicity and rapid proto-typing, React for its robust ecosystem suitable for complex applications, and Svelte for its innovative approach and performance benefits. The exploration of these frameworks highlighted the trade-offs between ease of use, flexibility, and performance, an essential consideration for any web development project.

What would we have done differently?

In hindsight, a key area we would approach differently is the consideration of deployment aspects and how these frameworks integrate within a larger system with a custom-defined backend. We realize that the choice of frontend framework can significantly impact the overall architecture, scalability, and maintenance of web applications. Delving into how each framework interacts with different backend technologies would have provided a more holistic view of their capabilities in a real-world scenario. Additionally, exploring server-side rendering or static site generation options for each framework could have given us valuable insights into their performance and SEO advantages.

Group work and work distribution

The group dynamics and work distribution were one of the strengths of this project. Each team member engaged early and showed enthusiasm, which fostered a collaborative and productive environment. We divided the work evenly, with each member focusing on a specific framework, allowing for a deep

dive into each technology. This approach not only enhanced individual learning but also enriched our group discussions, as each member brought detailed insights and unique perspectives about their respective frameworks. Our regular weekly meetings and open communication ensured that the project progressed smoothly and cohesively.

Future work – Reflections on further development

In our future work, we plan to delve deeper into the advanced capabilities of each framework and how they interact with different backend systems. We're particularly interested in how they fit into microservices architectures and serverless environments. We'll look at how they do in real deployment situations, focusing on load times, server-side rendering, and user experience. Alongside these technical evaluations, we also see the importance of user testing. By gathering feedback on how real users interact with the applications, we can understand the practical impact of each framework on user satisfaction and engagement. This feedback will be crucial in developing a set of best practices for building complex and user-friendly frontend systems.

The comparative analysis of ReactJS, Vue.js, and Svelte, as detailed in the article "Comparison and Analysis of Popular Frontend Frameworks and Libraries: An Evaluation of Parameters for Frontend Web Development" by Kaur and Tiwari, highlights the importance of examining how these frameworks approach UI design—a key aspect in frontend development (Kaur and Tiwari 5). React and Vue, while both being free, open-source, and performance-efficient, adopt distinct methodologies in their UI design processes. Svelte, on the other hand, also presents its unique approach to UI construction, which merits a closer examination.

Exploring the different resources available for Svelte, such as sveltematerialui.com by (Perrin), could provide valuable insights into its UI capabilities. This exploration would not only help in understanding Svelte's design philosophy and practical application but also offer a perspective on how it compares with the UI design approaches of React and Vue. Such an analysis is crucial for developers to make informed decisions about which framework best suits their project's UI requirements.

6 Conclusion

In conclusion, our comparative study of React, Vue, and Svelte frameworks reveals distinct advantages in each. React excels with a robust ecosystem and dynamic data handling, Vue offers streamlined build processes and user-friendly design, while Svelte stands out for its compile-time optimization and efficient performance. These findings, coupled with our developmental experiences, underscore the importance of choosing the right framework based on project needs, performance considerations, and sustainability. Our analysis provides valuable insights for future developers, highlighting each framework's potential to shape efficient, user-centric, and environmentally conscious web applications.

7 References

1. Acosta, Jan. n.d. "Test-driven development." IBM. Accessed December 6, 2023. `https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/`.

2. Bhardwaz, Saumyamani, and Rohan Godha. 2023. "Svelte.js: The Most Loved Framework Today." no. 1, 7.

3. Copes, Flavio. 2018. "The Vue Handbook: a thorough introduction to Vue.js." freeCodeCamp. `https://www.freecodecamp.org/news/the-vue-handbook-a-thorough-introduction-to-vue-js-1e`

4. Faulkner, Rosie. 2023. "Vue vs React vs... Svelte?!." Medium. `https://medium.com/@faulknerproject/vue-vs-react-vs-svelte-5f93d70d2618`.

5. Ferreira, Fabio, and Marco T. Valente. 2023. "Detecting code smells in React-based Web apps." no. 1, 16.

6. HcySun, Yang. 2023. Vue.JS Framework. 1st ed. Singapore: Springer Singapore.

7. IEEE, Raimundo N. Diniz-Junior, et al. 2022. "Evaluating the performance of web rendering technologies based on JavaScript: Angular, React, and Vue." `https://ieeexplore.ieee.org/document/9959901`.

8. "Introduction • Docs • SvelteKit." n.d. SvelteKit. Accessed December 5, 2023. `https://kit.svelte.dev/docs/introduction`.

9. Kaur, Gursheen, and Raj G. Tiwari. 2023. "Comparison and Analysis of Popular Frontend Frameworks and Libraries: An Evaluation of Parameters for Frontend Web Development." no. 1, 7.

10. Novac, Cornelia M., et al. 2021. "Comparative study of some applications made in the Vue.js and React.js frameworks." no. 1, 4.

11. Perrin, Hunter. n.d. Svelte Material UI. Accessed December 8, 2023. `https://sveltematerialui.com/`.

12. Wichmann, Nick. 2021. "React vs Vue vs Angular vs Svelte." DEV Community. `https://dev.to/hb/react-vs-vue-vs-angular-vs-svelte-1fdm`.