universität
uulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Case-study for SvelteKit in a Modern Business Application

Abschlussarbeit an der Universität Ulm

Vorgelegt von:
Hannah Lappe
hannah.lappe@uni-ulm.de
922114

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Rüdiger Pryss

Betreuer:
Dr. Marc Schickler
Jens Scheible
Martin Käser

2023

Abstract

SvelteKit is a modern full stack JavaScript framework. It provides many useful features for modern web development such as routing, server side rendering, and build optimization. In this work we investigate how SvelteKit performs when developing business applications. To this end, we reimplemented an existing business application in SvelteKit and compared original implementation with our SvelteKit implementation in terms of performance and developer experience.

In this work we give an overview of Svelte's and SvelteKit's features and concepts. We present the existing business application and our reimplementation. Finally, we provide the results of our comparison as well as various findings we discovered during our study.

Our results show that SvelteKit can be a great fit for business application where custom UI design is an important factor. Furthermore, it can provide fast performance. On the other hand, SvelteKit currently struggles with an immature library ecosystem as well as missing feature support.

Fassung November 2, 2023

Satz: PDF-LaTeX 2$_\varepsilon$

Contents

1 Introduction

Over the past decade, single page applications (SPAs) have become the most popular approach for developing websites. This development was fueled by JavaScript frameworks such as React, Angular and Vue.js which made it easy to develop feature rich web applications. But modern web applications need to satisfy a growing number of technical requirements. Features, such as server side rendering (SSR), code splitting, application routing, and state management have become ubiquitous. This abundance of requirements has led to a point where it has become difficult to start new projects from scratch.

To address this issue, in recent years the JavaScript ecosystem saw the emergence of so-called meta frameworks. Meta frameworks are JavaScript libraries that try to fit the role of opinionated "batteries included" frameworks, known from other programming languages, such as Ruby on Rails or Django. They provide out of the box support for many of the features required for building modern web applications. They promise that developers need to spend less time thinking about software architecture and can spend more time thinking about business requirements. Furthermore, meta frameworks can use JavaScript for both frontend and backend, which allows for a tighter integration between the two. Therefore, they can enable a more streamlined development experience, as well as reducing overall application complexity.

SvelteKit is an up-and-coming meta framework that is rapidly gaining popularity. It has multiple advantages over traditional SPA frameworks. SvelteKit allows per page control over the rendering strategy. This allows to render different parts of an application with SSR, client side rendering (CSR) or at build time with pre-rendering. Furthermore, SvelteKit uses various compile time optimizations to produce a more efficient and performant JavaScript bundle. Moreover, it provides functionality, such as reactive state management, component composition, and streamlined data fetch-

ing, which help in writing concise and readable code. Overall, SvelteKit promises to make development of modern web applications faster, while simultaneously improving developer experience.

1.1 Problem Statement

SvelteKit claims to speed up the development process by handling many common technical requirements of modern web development, but these claims are largely anecdotal. Furthermore, business applications tend to differ in their technical requirements compared to regular public facing web applications. Thus, it is unclear if SvelteKit would be a good fit for the development of modern business applications.

1.2 Purpose of this Study

This study examines SvelteKit's claimed benefits in the domain of modern business applications. To this end we compare multiple implementations of the same business application in terms of performance, development effort, and code complexity. Furthermore, we provide further considerations when using SvelteKit in production.

In this study, we investigate how SvelteKit's claimed benefits translate to the development of business applications. To this end we reimplement parts of an application that is used to schedule rides for a chauffeur service provided by a federal agency. We compare the resulting applications with the original implementation in terms of performance and development effort.

1.3 Thesis Structure

This thesis is structured in the following way: Chapter 2 will lay out fundamentals required for this thesis. This includes an overview of Svelte and SvelteKit, as well as SAP OpenUI5. In Chapter 3 we will provide an overview of related research. Afterwards, Chapter 4 will provide details about the specific code artifacts and their implementation. Following that, in Chapter 5 we will give our evaluation results. Finally, in Chapter 6 we will discuss our results and provide directions for future work.

2 Fundamentals

In this chapter we provide overviews for multiple different topics which will be relevant in the following thesis. We give a definition for enterprise applications and their use-case. Furthermore, present an overview of different rendering strategies used for web applications. Afterwards, we introduce Svelte and SvelteKit, including an overview of features and concepts used in these technologies. Finally, we lay out our methodology used for this research.

2.1 Enterprise Applications

An enterprise application is software which is used by companies, rather than individual users. Their purpose is to assist with tasks commonly found in the management and organization process of an enterprise. Common tasks for enterprise applications can be:

- enterprise communication
- business intelligence (BI)
- customer relationship management (CRM)
- human resource management (HRM)
- enterprise resource planning (ERP)

Requirements of an enterprise application can differ vastly depending on its use case, targeted user group, company size, company location, and scope [1, 2]. While many ready-made software solutions exist which provide support for common tasks such as ERP, HRM, and CRM, it is often not possible to map all processes of an enterprise to an off-the-shelf component. To this end custom software is often required, which is specifically developed for the given use-case.

2.2 Rendering Strategies

The rendering of web pages has evolved significantly over the years, with various techniques emerging to enhance user experience. In this section we provide an overview of relevant rendering strategies ordered by historical appearance.

2.2.1 Static HTML

Static HTML represents the most basic form of web page rendering. In this approach, the entire HTML content is generated on the server and sent to the client as a complete document. Static HTML offers exceptional performance, as there is minimal processing required on the client and server side. However, it lacks dynamic content updates and interactivity, making it suitable primarily for simple, non-interactive websites.

2.2.2 Server Side Rendering

Server-side Rendering (SSR) involves rendering web pages on the server, where both HTML and initial data are generated. Unlike with the static HTML approach, the server renders the HTML on demand. The server delivers a fully populated HTML page to the client, improving initial load times and search engine optimization (SEO). SSR is well-suited for content-heavy and dynamic websites, as it combines good performance with dynamic data capabilities. However, it may still require additional client-side processing for interactive features.

2.2.3 Client Side Rendering

Client-side rendering (CSR) shifts the rendering process to the client's browser. Initially, the server sends a minimal HTML template and JavaScript code. The browser then takes responsibility for rendering and fetching data, making CSR ideal for highly interactive web applications. However, CSR can result in slower initial page load times and worse SEO due to delayed rendering and indexing.

2.2.4 Static Site Generation

Static-site Generation (SSG) is a more extreme form of SSR. Instead of rendering the content when the page is requested, it is instead rendered when the site is initially built. This results in a page with similar performance to static HTML and therefore improved SEO. But, because the page has to be rebuilt every time content should change, it is only usable for content that rarely changes. This approach can also cause scaling issues in projects with a large amount of different pages. When using SSR these pages could be handled by a dynamic route which fetches the pages content from a headless content management system. With SSG model on the other hand, for each page a separate HTML file is generated during build-time. In extreme cases, this can mean large storage requirements.

2.2.5 Hydration

When server rendered or static web pages are sent to the client side they initially are not interactive beyond the scope of default HTML functionality. Hydration refers to the process of turning this "dumb" web page into a fully interactive client-side Application. With hydration an application can benefit from the fast render times and improved SEO of SSR and SSG, while simultaneously providing the user experience improvements of client-side applications.

2.2.6 Progressive Enhancement

Progressive enhancement is a web development strategy that focuses on building web application in a way that ensures a baseline level of functionality and accessibility for all users, regardless of their device, browser, or network capabilities. This approach involves creating a solid foundation of core content and functionality that works on virtually all web environments and then progressively enhancing it with additional features, styles, and interactivity for users with more advanced or modern browsers and devices.

2.3 Svelte

Svelte was created in 2016 by Rich Harris as a successor to Ractive.js [3]. It is a framework for developing web user interfaces, similar to React, Angular, and Vue.js. Harris describes Svelte as a framework not to run code but to think about code [3]. Svelte's key difference compared to other web UI frameworks is, that it uses a compiler. This makes it possible to shift many rendering steps from the run time into the build time. Therefore, Svelte can ship smaller and more efficient bundles.

Svelte uses its compiler to overload existing JavaScript syntax with special meaning. This allows for language extensions while still allowing it to be processed by existing JavaScript tooling (e.g. syntax highlighters).

2.3.1 Features

In the following we provide an overview of how Svelte's syntax is structured and showcase some of Svelte's features.

Svelte Files

Svelte is a custom language that is very similar to HTML, JavaScript, and CSS, which is colocated inside a `.svelte` file. In fact, plain HTML syntax is also valid Svelte syntax (Listing 1).

```
app.svelte
<h1>Hello World!</h1>
```

Listing 1: HTML is valid Svelte-syntax.

Similarly to HTML, JavaScript can be added inside `<script>` tags. But Svelte further extends this syntax to enable closer integration between JavaScript and HTML. Outside the script-block it is possible to use curly braces to write arbitrary JavaScript expressions. These expressions can reference variables and functions defined inside the script block (Listing 2).

```
<script>
  const name = 'world';
</script>

<div>Hello {name.toUpperCase()}!</div>
```

Listing 2: Example for JavaScript expressions in Svelte.

Similar to React, variables can also be used in element attributes, and Svelte even provides a shorthand when variable name and attribute name are the same (Listing 3).

```
<script>
  const src = './some/img.svg';
</script>

<img src={src}>
<img {src}> <!-- equivalent to <img src={src}> -->
```

Listing 3: Syntax for using variables in element attributes.

Reactivity

Reactive programming is a programming paradigm that enables development of declarative event-driven applications [4]. In reactive programming the developer only declares what to do, whereas when to do it is managed by the language. To this end, the language has to track dependencies and automatically propagate changes across these dependencies when they happen.

Svelte provides a strong system for reactive programming. Variable changes are automatically tracked and trigger UI updates. Therefore, it is possible to use regular assignment operations to mutate variables, which is not possible in frameworks such as React (Listing 4).

```
<script>
  let count = 0;

  function increment() {
    count += 1;
  }
</script>

<div>count: {count}!</div>
<button on:click={increment}>click me!</button>
```

Listing 4: Basic example for mutation of variables in Svelte.

To achieve this, Svelte's compiler replaces assignment operations with operations that automatically trigger changes of the reactive dependency chain at compile time. The increment function in the prior example would be compiled to an invalidation statement that tells the UI to update (Listing 5).

```
function increment() {
  $$invalidate('count', count += 1)
}
```

Listing 5: Compiler output for the `increment` function defined in Listing 4

Beyond reactive *UI updates* following mutation of a variable, Svelte also provides syntax to trigger *computations* following mutation of variables (Listing 6).

```
<script>
  let count = 2;

  $: doubled = count * 2;

  function increment() {
    count += 1;
  }
</script>

<button on:click={increment}>click me!</button>
<div>count: {count}!</div>
<div>doubled: {doubled}!</div>
```

Listing 6: Example of a reactive variable in Svelte. `doubled` will automatically be recalculated when `count` changes.

`$:` marks a statement as reactive. Every time a value, that is used in a reactive statement, changes, the statement is reevaluated. This can be used to declare reactive variables, as seen in the prior example, but it can also be used to run arbitrary code blocks (Listing 7).

```
<script>
  let count = 0;

  $: {
    console.log(`new value of count: ${count}`);
    console.log('This will also be executed when count changes');
  }

  $: if(count > 9) {
    console.log('It is over 9!');
    count = 0;
  }
</script>
```

Listing 7: Svelte example that runs statements when `count` changes

Templates

HTML does not have a way of expressing logic, thus Svelte introduces its own syntax. To conditionally render some markup, the markup has to be wrapped inside an if block (Listing 8).

```
<script>
  let count = 0;
</script>

{#if count > 9}
  <div>It's over 9!!</div>
{:else}
  <div>The number is pretty low</div>
{/if}
```

Listing 8: Example usage of Svelte's if block.

UIs also often work with lists of data. To this end, Svelte provides an each block, which can be used to handle iterable values such as arrays (Listing 9).

```
<script>
  let array = [1, 2, 3, 4];
</script>

{#each array as value}
  <div>{value}</div>
{/each}
```

Listing 9: Example usage of Svelte's each block.

Components

Since the emergence of React in 2013 [5], component-driven user interfaces have become a widespread approach to web development. In Svelte, a component is represented by a singular `.svelte` file, where the file name determines the component name (Listing 10).

```
MyComponent.svelte
<script>
  export let value;
</script>

<div>hello, {value}!</div>
```

Listing 10: Basic Svelte component that defines the input `value`.

To define inputs that should be passed into the Component, Svelte repurposes the `export` keyword. In the prior example `value` is defined as an input of the Component `MyComponent`. The Component can than be imported in another Svelte file and be used like a regular element (Listing 11).

```
app.svelte
<script>
  import MyComponent from './MyComponent.svelte';

  let value = 'world';
</script>

<MyComponent value={value} />
```

Listing 11: Example usage of a custom Svelte component.

Events

In a prior example (Listing 6) we already showed usage of `on:click`. This directive is used to listen to mouse clicks. Svelte makes it possible to listen for arbitrary events on an element with the `on` keyword (Listing 12). Furthermore, Svelte provides an event dispatcher to enable components to send their own custom events (Listing 13).

```
<script>
  function handleMove(event) {
    // ...
  }
</script>

<div on:pointermove="{handleMove}" ></div>
```

Listing 12: Usage of the `on:` directive to listen to `pointermove` events.

```
MyComponent.svelte
<script>
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();

  function sayHello() {
    dispatch('message', { text: 'Hello!' });
  }
</script>
```

```
app.svelte
<script>
  import MyComponent from './MyComponent.svelte';
</script>

<MyComponent on:message={(e) => console.log(e.detail.text)} />
```

Listing 13: Usage of the event dispatcher to send custom events.

Styling

Like most web frameworks, Svelte uses CSS to style content. To this end, it is possible to use `<style>` tags to define CSS similar to HTML (Listing 14). Svelte

automatically scopes defined styles. This means, the style will only affect markup that is in the same file as the style. This can prevent unexpected styling behavior in growing code bases.

```
<button>Click me!</button>

<style>
  button:hover {
    background: red;
  }
</style>
```

Listing 14: Example usage of `<style>` tag to add CSS.

Directives

In the previous section we introduced the `on:` keyword to react to events. This keyword is called a directive. Svelte provides multiple directives to control the behavior of components in some ways.

The `bind:` directive can be used to create two-way bindings between parent and child elements. In Svelte, data usually flows from parent to child with the child having no way of directly mutating the data that it is being passed. By passing a property using the `bind:` directive, Svelte enables data to flow the other way (Listing 15).

```
<script>
  let name = 'Peter';
</script>

<input bind:value={name} />
```

Listing 15: Two-way binding with the `bind:`-directive.

The `class:` directive is a utility that can be used to dynamically set CSS classes on an element. This directive takes an expression that when evaluation to true, will set the given class on the element. This directive also provides a shorthand in cases where a variable is used to set a class of the same name (Listing 16).

```
<script>
  let active = true;
</script>

<!-- all tree declarations are equivalent -->
<div class={active ? 'active' : ''} />
<div class:active={active} />
<div class:active />
```

Listing 16: Example usage of the `class:` directive.

Similar to the previous directive, the `style:` directive provides a utility for dynamically setting CSS-styles on an element. The directive takes a value that is used to set the style attribute, where `undefined` and `null` means that the attribute should not be set. In cases where a variable is used to set a style attribute of the same name, Svelte also provides a shorthand (Listing 17).

```
<script>
  let color = "red";
  let width = "20rem";
</script>

<!-- all tree declarations are equivalent -->
<div style="color:red; width:20rem" />
<div style:color="red" style:width="20rem" />
<div style:color style:width />
```

Listing 17: Example usage of the `style:` directive.

The `use:` directive is used to register an action on an HTML element. An action is a function that is called when the element is created. The function receives a reference to the element and can thus be used to augment behavior of an element. In Listing 18 the `use:` directive is used to register event listeners that log to the console when a mouse enters the element. While this functionality can be easily implemented using the `on:` directive, actions become especially useful when more complex behavior has to be configured in a reusable way. Section 2.4.1 will introduce an action that progressively enhances form elements.

```
<script>
  function listen(node) {
    node.addEventListener('mouseenter', () => console.log("mouse enter"))
    node.addEventListener('mouseleave', () => console.log("mouse leave"))
  }
</script>

<button use:listen>Click me!</button>
```

Listing 18: Example usage of actions to register an event listener.

TypeScript

TypeScript is a programming language that has gained a lot of popularity in recent years [6]. It is a superset of JavaScript that adds typing and type annotations to the language. Svelte has support for TypeScript out of the box. TypeScript can be enabled for a Svelte file by annotating the script block with a `lang` attribute (Listing 19).

```
<script lang="ts">
  let value: string = "World"

  function sum(a: number, b: number): number {
    return a + b;
  }
</script>
```

Listing 19: Svelte file that uses TypeScript.

It should be noted that using TypeScript in conjunction with Svelte is currently subject to a technical limitation: it is not possible to use TypeScript syntax outside the `<script>` tag [7]. One might for example want to type the parameters of an inline event handler (Listing 20). Fortunately, in most cases this can be fixed by refactoring the inline function to a function definition in the `<script>` tag.

```
<input on:change={(e: InputEvent) => {}}> /* Syntax Error */
```

Listing 20: TypeScript syntax in inline functions is currently not possible.

2.4 SvelteKit

SvelteKit is a framework that builds on top of Svelte to provide a toolkit for rapidly developing web applications [8]. It can be used out of the box to develop full stack applications, server rendered frontend applications, static web pages, and traditional single page applications without any server rendering. It provides multiple features, required for the development of modern web applications. Features include routing, support for different rendering patterns, build optimization, data preloading, and zero-configuration deployments. Furthermore, SvelteKit uses Vite[1] to enable a good development experience, with fast startup times and quick hot module replacement.

2.4.1 Features

In the following we give an overview of multiple SvelteKit concepts and features, which will be relevant for the rest of this study.

Routing

SvelteKit provides built-in support for routing. To this end, SvelteKit utilizes a file system based routing system. This means that the URLs that a user can access on the web application directly maps to the directory structure in the project's file system. A SvelteKit project contains a `routes` directory. This directory marks the root of the URL, e.g.: `http://example.com/`. Respectively, the directory structure `routes/about` would map to `http://example.com/about`. This system can also be used to create parameterized routes using characters that one would not normally expect within a file path. The directory structure `routes/blog/[slug]` would create a route, where `slug` serves as a wildcard (e.g.: `/blog/hello-world`, `/blog/123`). The path component matched by this wildcard can then be used in the load function (see Section 2.4.1). The actual content of a page is defined in a `+page.svelte` file (Listing 21).

[1] https://vitejs.dev/

routes/+page.svelte

```
<h1>My Website</h1>
<p>Welcome to my website...</p>
```

routes/about/+page.svelte

```
<h1>About</h1>
<p>about this site...</p>
<a href="/home">Home</a>
```

Listing 21: Example page-definitions in SvelteKit.

SvelteKit uses the HTML `<a>` tag to navigate between pages, rather than a framework specific `<Link>` component. This has the advantage of being progressively enhanceable. If JavaScript is not available on the client-side, the anchor tag's navigation event can be handled by the browser and the browser will do a regular full page reload to reach the navigation target. In the case of JS being available, SvelteKit's runtime router will intercept the navigation event and perform an in-app navigation to the target without causing a full page reload.

Most websites have some elements that should be displayed across multiple pages, such as a navigation bar or a footer. To this end, SvelteKit's router has the concept of layouts. A layout is placed alongside the pages inside the `routes` directory, inside a `+layout.svelte` file. This layout will automatically be applied to all route and all sub-routes of the directory, where the layout file is placed. The place where the actual page content should be rendered inside the layout is defined by a `<slot>` element (Listing 22).

routes/+layout.svelte

```
<nav>
  <a href="/home">home</a>
  <a href="/blog">blog</a>
  <a href="/about">about</a>
</nav>

<slot/>
```

Listing 22: Example for a layout that provides a navigation bar.

In some cases an application may have multiple routes, that should live on the same route as others, but should have their own shared layout. One could for example imagine an application where the user authentication pages should have a different

layout than the authenticated area of the app. To support this, SvelteKit provides a mechanism to group routes together without this route being reflected in the URL by wrapping the group name in parentheses (Listing 23).

```
routes/
  (restricted)/
    dashboard/
    details/
    +layout.svelte
  (public)/
    signin/
    register/
    +layout.svelte
```

Listing 23: Usage of layout groups to provide different layouts for public and restricted routes of an application.

In this example two new layout groups are introduced (`(restricted)` , `(public)`). Both groups can provide their own layout without having an influence on the URL. For example, the sign-in page is available on `/signin` (not `/(public)/signin`).

Loading Data

SvelteKit facilitates a standard way for fetching dynamic content that is to be rendered on a page. This is required to enable server side rendering and static site generation. To this end, every page can have a dedicated load function that is defined in a `+page.js` file. When a user requests a page, this load function is executed and the returned data is passed to the `+page.svelte` file as special property `data` (Listing 24).

In SSR mode, when a user first requests a page, the load function is executed on the server. Afterwards the Svelte-file is rendered once, before being sent to the user as populated HTML. In the browser, SvelteKit then hydrates the page. During hydration, SvelteKit runs the load function a second time on the client to create a consistent data model.

After the app has finished hydrating, all following navigation events will be handled by SvelteKit's runtime router. When the user now navigates to a different page the

routes/+page.js

```
export async function load() {
  const todos = (await fetch('/api/todos')).json();

  return { todos };
}
```

routes/+page.svelte

```
<script>
  export let data;
</script>

{#each data.todos as todo}
  <div>{todo.name}</div>
{/each}
```

Listing 24: Example page that fetches to-dos from an API and shows them.

load function now is invoked on the client-side directly, therefore saving an unnecessary roundtrip to the server. Because these load functions run on both server and the client, they are called universal load functions.

Some data fetching can only take place on the server side, for example database access. Therefore, SvelteKit also provides an alternative form of load function that is only executed on the server. These server load function are placed in `+page.server.js` files (Listing 25).

routes/+page.server.js

```
import db from '$lib/server/db';

export async function load() {
  const todos = await db.getTodos();

  return { todos };
}
```

Listing 25: Usage of server load functions to fetch to-dos from a database.

In the same way as universal load functions, server load functions are executed on the server when a user initially requests a page. But, because they can only run on the server, when a user navigates inside the app, instead of running the load function client-side, the client sends a request to the server for new data. The server then executes the load function and sends the resulting data back to the frontend

as JSON.

This behavior also highlights another difference between universal and server load functions. Because server load functions need to be able to send JSON to the client, all data they return must be serializable. This restriction does not apply to universal load functions. Therefore, they can return arbitrary data such as class instances (Listing 26).

routes/+page.js

```
class Todo {
  constructor(name) {
    this.name = name;
  }
}

export async function load() {
  const todos =  [ new Todo(`finish master's thesis`) ]

  return { todos };
}
```

Listing 26: Universal load function that returns non-serializable data.

Form Actions

The previous section discussed how SvelteKit applications send dynamic data to the client. But, many applications also need a way to send data to the server. To this end, SvelteKit provides form actions. `+page.server.js` files can export actions which make it possible to post data to the server using HTML's `<form>` elements (Listing 27).

The name of the action, the form should post to, is specified as a query parameter. This makes it possible to have multiple actions per route. Because the implementation uses only default HTML forms, it is again possible to progressively enhance this feature. In fact, in the shown implementation, submitting the form would cause a full page reload because the browser is sending the post request to the server. To progressively enhance this form, one can add the `enhance` action (Listing 28):

This action intercepts form submit events and instead posts them to the server as an asynchronous request. In case the server does not return an error, the client

routes/+page.server.js

```
export const actions = {
  postTodo: async () => {
    // post Todo
  }
}
```

routes/+page.svelte

```
<form method="POST" action="?/postTodo">
  <label>
    Todo name:
    <input name="todoName">
  </label>
</form>
```

Listing 27: Example server action to post a new to-do.

routes/+page.svelte

```
<script>
  import { enhance } from '$app/forms';
</script>

<form method="POST" action="?/postTodo" use:enhance>
```

Listing 28: Progressively enhanced form that will handle submits without a full page reload.

will then trigger the load function again to synchronize any data changes that could have happened as a result of the form action. This way the UI can forego a full page reload and therefore provide a better user experience.

Rendering

SvelteKit provides the functionality to define the rendering strategy for each route independently. By exporting the constants `ssr`, `csr` and `prerender` in a page or a layout, the rendering behavior can be precisely changed (Listing 29).

- `prerender` determines if the page should be rendered during the build process. If this set to true, SvelteKit will serve the page as static files. While this will improve performance it also means that the page cannot have form actions because SvelteKit will not provide handlers for a pre-rendered route.

```
routes/+page.js
// default settings:
export const prerender = false;
export const ssr = true;
export const csr = true;
```

Listing 29: Variables that can be exported to configure the rendering behavior of a page in SvelteKit.

Furthermore, the pages load function is ran during build-time. Therefore, any external dependencies (e.g.: REST API) need to be available at this point in time.

- `ssr` determines if the page should be rendered on the server site when the user first requests the page. This does not mean the page cannot specify a server load function. Instead, the client will simply request the data from the server load function as soon as the runtime is initialized on the client side. This option can sometimes be useful, when a page requires certain features that require access to browser exclusive API's to render its content. But, most of the time it is possible to defer access to these browser-specific API's by running the code after the function has been mounted.
- `csr` configures if SvelteKit should run any JavaScript on the client side for this page. This way applications that do not require JavaScript in the browser, can completely disable it. But this has certain side effects. For example, given an application has a layout that contains a navigation bar which needs JavaScript to work. If the user first loads the website on a page that has CSR disabled, the navigation bar will not work. On the other hand, if the user loads the website on a page that has CSR enabled and then navigates to a page with CSR set to false, the navigation bar will work.

2.5 UI5

UI5 is a JavaScript based web frontend framework. It was developed by SAP, a German enterprise software company. The framework was first open-sourced in 2013 under the title OpenUI5 [9]. It is focused on development of enterprise-ready

applications. UI5's main selling points are a wide range of integrated features, such as data-bindings, OData API[2] support, and localization. Furthermore, UI5 provides many ready-made UI-Components with support for accessibility, keyboard navigation and right-to-left languages [10].

UI5's UI components are tightly integrated with Fiori[3], a set of UI design guidelines that is optimized for the development of enterprise applications.

2.6 Methodology

As outlined in Section 1.1 it is unclear if SvelteKit's claimed benefits translate to the development of real-world business applications. To this end, this study tries to provide evidence that helps in classifying SvelteKit's true real-world strengths and weaknesses. Furthermore, we tried to identify potential shortcomings in SvelteKit's design and feature set. We aim to create a knowledge base of SvelteKit's advantages and disadvantages which can be used for future technology choices.

To achieve these goals we used SvelteKit to implement an application that serves a real-world business use case. To ensure the selected use case is representative, we chose an application that is already used in production. Furthermore, we were then able to compare the created SvelteKit implementations to the existing implementation.

We chose to implement three different variants of the application using SvelteKit. Firstly, a full stack application that implements frontend and backend in the same code base. Secondly, a frontend implementation that uses server side rendering but is only responsible for the UI, whereas business logic is handled by an existing backend. Finally, a single page application which only handled the UI and produces a static HTML bundle. We expect these application types to be very common in real world use cases.

[2] https://www.odata.org/
[3] https://experience.sap.com/fiori-design/

2.6.1 Evaluation

When comparing different languages and language frameworks, one can take a practically infinite amount of measurements. Examining these carefully can be very important for fulfilling non-functional system requirements such as performance. However, as we can clearly see by comparing the popularity of C and Python, there is more to the choice of language than simple numbers. We provide some basic metrics as a broad overview and encourage anybody looking for more broad or specific data to conduct further experiments specific to their requirements.

An important factor that cannot be measured easily is developer experience. There is some existing methodology for quantifying this elusive property, but it requires conducting and carefully evaluating large-scale developer surveys, which is beyond the scope of this research [11]. And even if the numbers do point a certain way, developer experience may be influenced significantly by factors such as prior experience with similar tools, or the domain of the software to be created. For example, requirements to interact with existing systems may make using a framework much less enjoyable. We therefore focus on highlighting what we believe to be the key differences in the approaches of the discussed frameworks, so the reader can make an educated guess as to which technology would probably fit their use case best.

3 Related Work

In this chapter we introduce related research that discusses Svelte or SvelteKit. Because Svelte only recently gained fame, not much scientific work has been published about the technology yet. Nonetheless, two works compare Svelte to contemporary JavaScript frameworks. Furthermore, one survey investigates trends in the JavaScript ecosystem and thus also discusses Svelte and SvelteKit.

3.1 Angular and Svelte Frameworks: A Comparative Analysis

In their work, Tripon et al. compare Angular and Svelte [12]. To this end, they implemented an example application in both technologies and compared time to set up the project as well as performance. Their results showed that the Svelte implementation generally performed better than the Angular implementation. Furthermore, they showed that the project setup was faster for Svelte than for Angular. But, they noted that Svelte did not configure a testing environment out of the box. This has since then been remedied, the current setup process for SvelteKit provides the option to configure unit test and integration test environments. Overall they concluded that Svelte is a good fit for small to medium-sized projects, while Angular is best suited for large projects.

3.2 DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte

In their work, Levlin conducted a performance analysis comparing React[1], Angular[2], Vue[3], and Svelte [13]. These performance benchmarks primarily focused on DOM manipulations and compared the frameworks speed when inserting, editing, and deleting DOM nodes. Levlin's results show that Svelte is among the fastest of the compared frameworks when manipulating only few nodes. On the other hand, Svelte struggles when manipulating large amounts of elements. Levlin determined React as the overall winner of their study because it performed well in the benchmark section, provides large amounts or resources, and was the most popular in the State of JS survey (Section 3.3). Notably, Levlin perceived Svelte as very easy to use and noted that it is primarily held back by its comparatively small user base.

3.3 State of JavaScript

State of JavaScript is a yearly Survey run by S. Greif and E. Burel [14]. The survey intends to identify trends in the web development, therefore helping developers when making technology choices. It asks questions about JavaScript language features, frontend-frameworks, meta-frameworks and build tools. It is an openly accessible survey that allows everybody to answer. Survey participants were mostly from past surveys and social media traffic. Furthermore, respondents where not filtered in any way. Thus, the survey cannot guarantee to be representative for the entire developer community. Instead, it shows results for a specific subset of developers.

The results for 2022 show Svelte as the fourth most used frontend framework with 21.1%, behind React (81.8%), Angular (48.8%), and Vue (46.2%). In terms of retention (participants that would use the framework again), Svelte placed second (89.7%), behind Solid (90.9%). SvelteKit also ranked fourth in terms of usage

[1] `https://react.dev/`
[2] `https://angular.io/`
[3] `https://vuejs.org/`

(11.9%) behind Next.js[4] (48.6%), Gatsby[5] (23%), and Nuxt[6] (18.1%). SvelteKit too placed second in for retention (92.5%), behind Astro[7] (92.8%). While these results are not representative for the software development community as a whole, they still show that SvelteKit is an up and coming technology that is worth to be investigated further.

[4] https://nextjs.org/
[5] https://www.gatsbyjs.com/
[6] https://nuxt.com/
[7] https://astro.build/

4 Implementation

In this chapter we go over the details of the software implementations we created during our study. To this end, we introduce the use-case and context of the model project. Afterwards, we summarize the architecture details of the original implementation. Finally, we introduce the three implementation variants we created and explain our design decisions.

4.1 Chauffeur Service

As a basis for our case study we used an application called DSW-FD (Dispositionssoftware-Fahrdienst). DSW-FD is used to schedule chauffeur rides for a German federal agency. The application provides a rich user interface with views for managing chauffeur jobs as well as chauffeur drivers. Clients which need a chauffeur ride can call a separate hotline where a handler creates a new chauffeur job. This job is then sent to DSW-FD where it appears in the overview of jobs and can then be processed by a dispatcher.

Drivers need to be manually assigned to a job by the dispatcher. To this end, the application provides suggestions for drivers which would be free when the job is scheduled and are closest to the job's departure point. Furthermore, the app provides functionality to group together different jobs so that they may be handled by a single driver, determine the return destination for the driver after they finished the job, as well as marking a job as being handled by a pool of drivers.

The application also provides features to directly interact with drivers, such as broadcasting messages to all drivers, reminding a driver to take their mandatory break, and directly calling a driver.

4.2 Current Implementation

The current Implementation is realized using a classic frontend-backend architecture. The backend is implemented using the Java web framework Spring[1]. It provides a SOAP web server on which new jobs are submitted from another service. Data is persisted in a MSSQL database. The backend also provides Multiple REST APIs and an OData API for communication with the frontend. Furthermore, the backend has to interact with Firebase to trigger push notifications on smartphone devices.

The system provides two frontends, an android application, which is used by the chauffeur drivers to receive jobs and communication from headquarters, and a web client which is used by handlers to manage jobs and drivers. The web client is implemented using OpenUI5[2]. OpenUI5 is a frontend framework published by SAP, it is intended for development of enterprise applications which follow SAP's Fiori design guidelines[3]. All APIs of the backend require authentication. To this end the frontend has to authenticate with a Keycloak[4] instance using OpenID Connect (OIDC). An overview of the system architecture is provided in Figure 4.1.

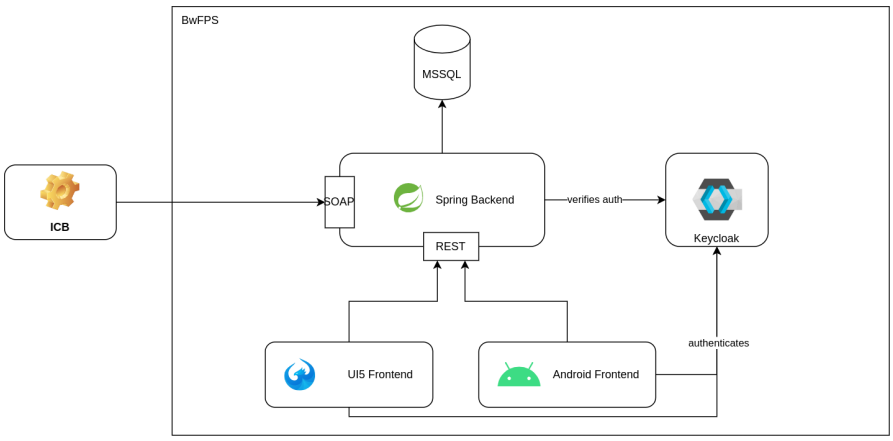
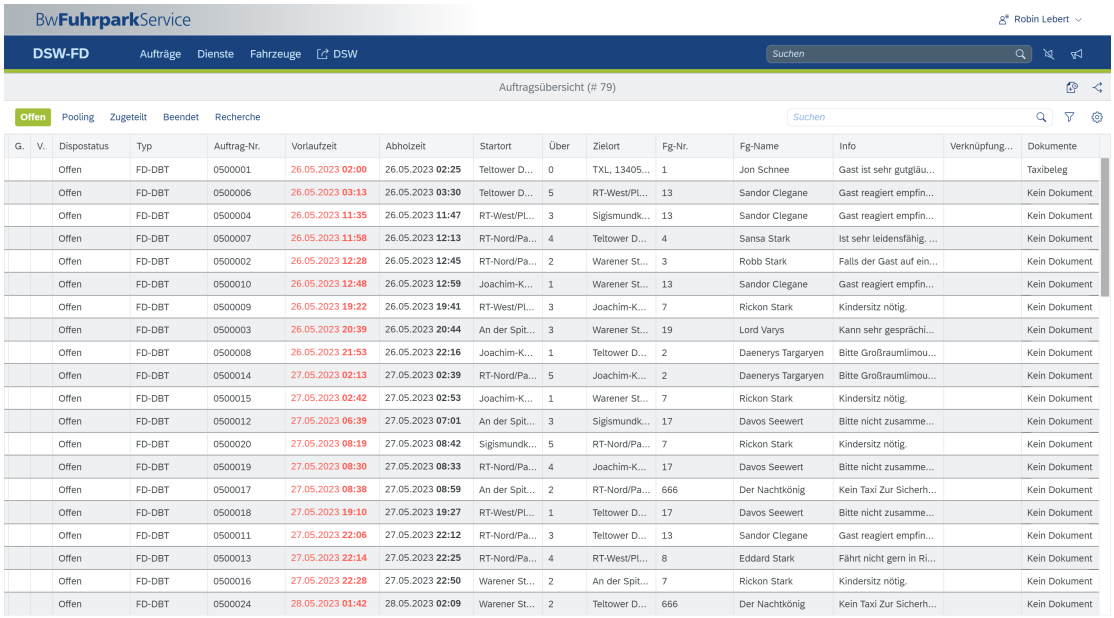


Figure 4.1: Architecture overview of the original implementation

[1] https://spring.io/
[2] https://openui5.org/
[3] https://experience.sap.com/fiori-design/
[4] https://www.keycloak.org/



G.	V.	Dispostatus	Typ	Auftrag-Nr.	Vorlaufzeit	Abholzeit	Startort	Über	Zielort	Fg-Nr.	Fg-Name	Info	Verknüpfung...	Dokumente
		Offen	FD-DBT	0500001	26.05.2023 **02:00**	26.05.2023 **02:25**	Teltower D...	0	TXL, 13405...	1	Jon Schnee	Gast ist sehr gutgläu...		Taxibeleg
		Offen	FD-DBT	0500006	26.05.2023 **03:13**	26.05.2023 **03:30**	Teltower D...	5	RT-West/Pl...	13	Sandor Clegane	Gast reagiert empfin...		Kein Dokument
		Offen	FD-DBT	0500004	26.05.2023 **11:35**	26.05.2023 **11:47**	RT-West/Pl...	3	Sigismundk...	13	Sandor Clegane	Gast reagiert empfin...		Kein Dokument
		Offen	FD-DBT	0500007	26.05.2023 **11:58**	26.05.2023 **12:13**	RT-Nord/Pa...	4	Teltower D...	4	Sansa Stark	Ist sehr leidensfähig. ...		Kein Dokument
		Offen	FD-DBT	0500002	26.05.2023 **12:28**	26.05.2023 **12:45**	RT-Nord/Pa...	2	Warener St...	3	Robb Stark	Falls der Gast auf ein...		Kein Dokument
		Offen	FD-DBT	0500010	26.05.2023 **12:48**	26.05.2023 **12:59**	Joachim-K...	1	Warener St...	13	Sandor Clegane	Gast reagiert empfin...		Kein Dokument
		Offen	FD-DBT	0500009	26.05.2023 **19:22**	26.05.2023 **19:41**	RT-West/Pl...	3	Joachim-K...	7	Rickon Stark	Kindersitz nötig.		Kein Dokument
		Offen	FD-DBT	0500003	26.05.2023 **20:39**	26.05.2023 **20:44**	An der Spit...	3	Warener St...	19	Lord Varys	Kann sehr gesprächi...		Kein Dokument
		Offen	FD-DBT	0500008	26.05.2023 **21:53**	26.05.2023 **22:16**	Joachim-K...	1	Teltower D...	2	Daenerys Targaryen	Bitte Großraumlimou...		Kein Dokument
		Offen	FD-DBT	0500014	27.05.2023 **02:13**	27.05.2023 **02:39**	RT-Nord/Pa...	5	Joachim-K...	2	Daenerys Targaryen	Bitte Großraumlimou...		Kein Dokument
		Offen	FD-DBT	0500015	27.05.2023 **02:42**	27.05.2023 **02:53**	Joachim-K...	1	Warener St...	7	Rickon Stark	Kindersitz nötig.		Kein Dokument
		Offen	FD-DBT	0500012	27.05.2023 **06:39**	27.05.2023 **07:01**	An der Spit...	3	Sigismundk...	17	Davos Seewert	Bitte nicht zusamme...		Kein Dokument
		Offen	FD-DBT	0500020	27.05.2023 **08:19**	27.05.2023 **08:42**	Sigismundk...	5	RT-Nord/Pa...	7	Rickon Stark	Kindersitz nötig.		Kein Dokument
		Offen	FD-DBT	0500019	27.05.2023 **08:30**	27.05.2023 **08:33**	RT-Nord/Pa...	4	Joachim-K...	17	Davos Seewert	Bitte nicht zusamme...		Kein Dokument
		Offen	FD-DBT	0500017	27.05.2023 **08:38**	27.05.2023 **08:59**	An der Spit...	2	RT-Nord/Pa...	666	Der Nachtkönig	Kein Taxi Zur Sicherh...		Kein Dokument
		Offen	FD-DBT	0500018	27.05.2023 **19:10**	27.05.2023 **19:27**	RT-West/Pl...	1	Teltower D...	17	Davos Seewert	Bitte nicht zusamme...		Kein Dokument
		Offen	FD-DBT	0500011	27.05.2023 **22:06**	27.05.2023 **22:12**	RT-Nord/Pa...	3	Teltower D...	13	Sandor Clegane	Gast reagiert empfin...		Kein Dokument
		Offen	FD-DBT	0500013	27.05.2023 **22:14**	27.05.2023 **22:25**	RT-Nord/Pa...	4	RT-West/Pl...	8	Eddard Stark	Fährt nicht gern in Ri...		Kein Dokument
		Offen	FD-DBT	0500016	27.05.2023 **22:28**	27.05.2023 **22:50**	Warener St...	2	An der Spit...	7	Rickon Stark	Kindersitz nötig.		Kein Dokument
		Offen	FD-DBT	0500024	28.05.2023 **01:42**	28.05.2023 **02:09**	Warener St...	2	Teltower D...	666	Der Nachtkönig	Kein Taxi Zur Sicherh...		Kein Dokument

Figure 4.2: Overview of chauffeur jobs in current implementation filled with dummy data.

4.3 SvelteKit Implementation

Our study focused on the core of DSW-FD's system landscape, comprised of the Spring backend and the OpenUI5 web frontend. We experimented with two different implementations. One where SvelteKit replaces both the backend and frontend, and one where SvelteKit only replaces the frontend. Both our implementations had to communicate with the Keycloak service to authenticate. The full stack implementation also had to interact with the MSSQL database. As to focus on the most relevant parts, APIs for the android application, Firebase communication, and the SOAP API for ICB were not implemented. We also decided to use TypeScript instead of regular JavaScript. This provided a better developer experience with little extra overhead. Because SvelteKit can infer types for most of its internal functionality, extra types had to be only defined for external APIs.

We focused our implementation on a subset of features required to manage chauffeur jobs. To this end we implemented three pages: the overview page that shows all currently relevant chauffeur jobs (Figure 4.2), the detail page for chauffeur jobs (Figure 4.3), and a view that is used to create new (internal) chauffeur jobs.

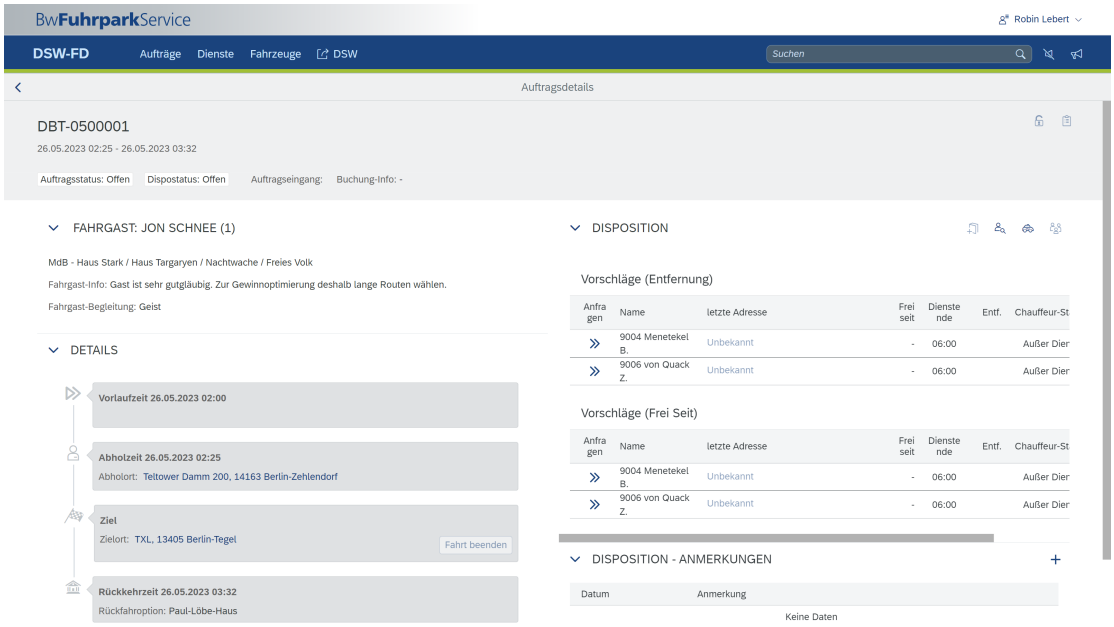


Figure 4.3: Chauffeur job details view in current implementation for a dummy job.

4.3.1 Full Stack Implementation

Our first approach was the full stack implementation. This implementation has to replace the UI5 frontend as well as the Spring backend (Figure 4.4). To this end, business logic, database communication and API's all had to be implemented in SvelteKit. We used Prisma[5] for object relational mapping to communicate with the database, because Prisma provides functionality to generate its data model from an existing database using introspection. This allowed us to save time on defining models.

This implementation makes use of SvelteKit's server load functions (Section 2.4.1). The load function for each page calls a separate service function which queries required data from the database. For posting data to the server we used SvelteKit's recommended mechanism, form actions (Section 2.4.1). Form actions send `FormData` objects[6] to the server. These objects hold the raw request data as key-value pairs. To validate the correctness of this data we used Zod[7]. Zod is a schema validation library that can be used to define constraints that a data structure has

[5] https://www.prisma.io
[6] https://developer.mozilla.org/en-US/docs/Web/API/FormData
[7] https://zod.dev/

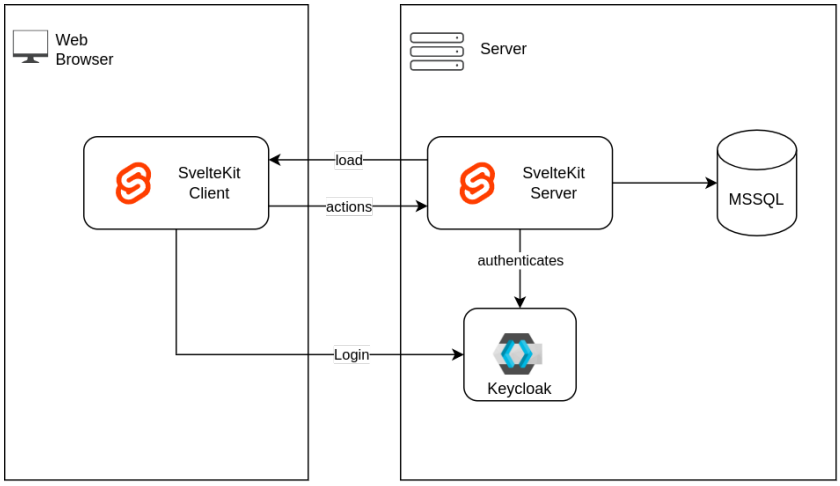


Figure 4.4: Architecture overview of the implementation using SvelteKit as a full stack framework

to satisfy. Furthermore, with the extension `zod-form-data`[8], Zod can be used to parse `FormData` objects into a given data structure. This overall improved the ergonomics of using SvelteKit's form actions.

4.3.2 Frontend Implementation

We also decided to explore an approach where SvelteKit is only used as a frontend. This approach provides more flexibility because backend technology can be chosen individually. Communication to the backend is primarily handled using web APIs. In our use case we decided to reuse the existing Java backend. The backend's REST API could be queried from SvelteKit. As REST APIs are something which can be called from frontend and backend, this approach can make use of SvelteKit's universal load functions (Section 2.4.1). This means that the SvelteKit server is only used during SSR. After the browser has loaded the page, universal load functions are executed client-side. The client then directs its requests directly to the backend instead of sending a request to the SvelteKit server which then requests the data from the backend. Therefore, saving one unnecessary hop. If loss of SSR is acceptable, this approach would also make it possible to run the application as an SPA, forgoing a dedicated SvelteKit server (Figure 4.5). This can be useful, as it

[8] `https://www.npmjs.com/package/zod-form-data`

allows for the SvelteKit application to be served as static files from a web directory or similar.

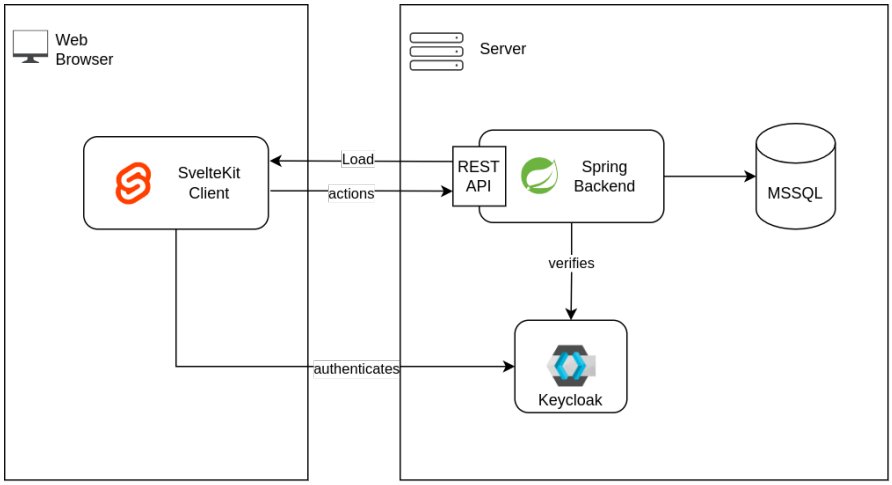


Figure 4.5: Architecture overview of the implementation using SvelteKit as an SPA framework without a dedicated SvelteKit server.

4.3.3 SPA Implementation

But this approach has several shortcomings explained in Section 5.3.3. Therefore, we further experimented with an implementation that again uses server load functions and server actions. This implementation is very similar to the full stack implementation discussed earlier. But instead of implementing business logic and database access in the SvelteKit server side code, the server side is primarily used as middleware that sends REST requests to the Spring backend Figure 4.6.

This approach has multiple advantages. It uses much of the built-in functionality of SvelteKit, applications can be progressively enhanced, the client requires fewer dependencies, because functionality such as parsing and validating form data can be handled server side. Furthermore, authentication is simplified, as only the backend needs to communicate with the API. And finally, the outlined problems with fetch in universal load functions is solved, because only the backend uses fetch.

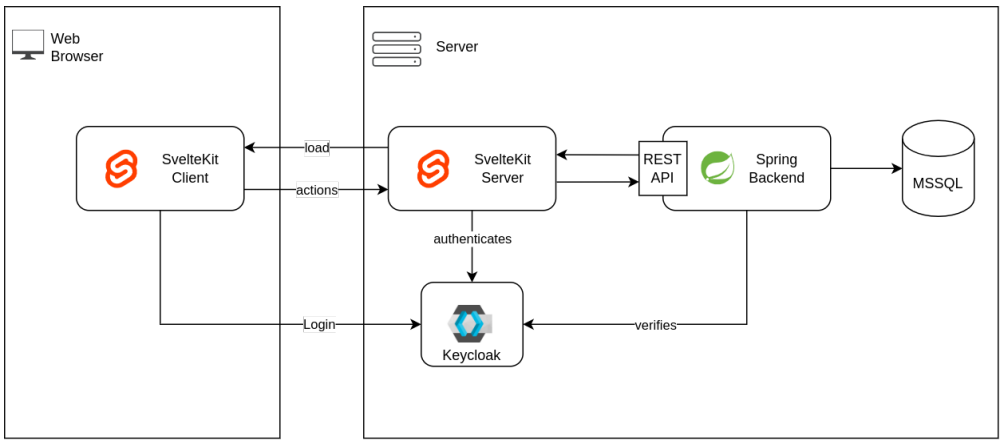


Figure 4.6: Architecture overview of the implementation using SvelteKit only for the frontend where the SvelteKit server serves as a middleware to handle communication with the Spring backend.

4.3.4 UI Considerations

One of the project client's requirements for the original implementation was that the frontend follows the SAP Fiori design guidelines. While not strictly necessary for our study, we decided to adhere to this requirement in our SvelteKit implementations. We hoped to gain insights into how SvelteKit behaves when interacting with UI libraries. The current implementation uses UI5 which has SAP Fiori components built into the framework itself. As it is not possible to use these components outside the framework, we had to use alternatives.

We first tried to use UI5 Web Components[9]. These premade components promise to be a feature rich and framework-agnostic implementation of the Fiori Design Guidelines. In practice, we noticed issues with this approach which will be discussed in Section 5.3.2. Therefore, we decided to use SAP Fundamental Styles[10], a library that provides CSS styles to create Fiori components. We then built a custom UI Component library around these styles. This had the added benefit of providing insights into how Svelte performs for creating UI components.

[9] https://sap.github.io/ui5-webcomponents/
[10] https://sap.github.io/fundamental-styles/

5 Evaluation

In this chapter we go over our evaluation results. We discuss the outcome of our performance benchmarks. Afterwards, we compare SvelteKit and UI5 on the basis of multiple different code examples to find out how they perform in terms of developer experience. Finally, we discuss further notes and findings about Svelte and SvelteKit we uncovered during our study.

5.1 Performance

To gain insights into SvelteKit's performance, we ran benchmarks on the existing UI5 implementation as well as all newly created SvelteKit implementations. Levlin already tested Svelte's performance in DOM manipulation tasks [13], thus we were primarily interested how SvelteKit would fare when loading pages as well as navigating between pages. To this end, we intended to measure the time to first byte (TTFB), first contentful paint (FCP), largest contentful paint (LCP), and time until a new view has rendered after navigation. Following the definitions of web.dev [15], TTFB describes the time it takes until the first byte of data reaches the browser, after first loading a website. First contentful paint describes the time it takes until the browser first renders part of the application to the screen, while largest contentful paint describes the time it takes until the largest image or text block visible within the viewport is rendered. Ordinarily, Lighthouse, a tool for measuring website performance integrated in Chromium, would have been used to measure TTFB, FCP, and LCP. But, we realized that Lighthouse's automated measurement of LCP did not provide a realistic measurement for how long the UI5 implementation took to render its content. Therefore, we resorted to using Chromium's performance tool to create performance recordings. We then took measurements of LCP manually

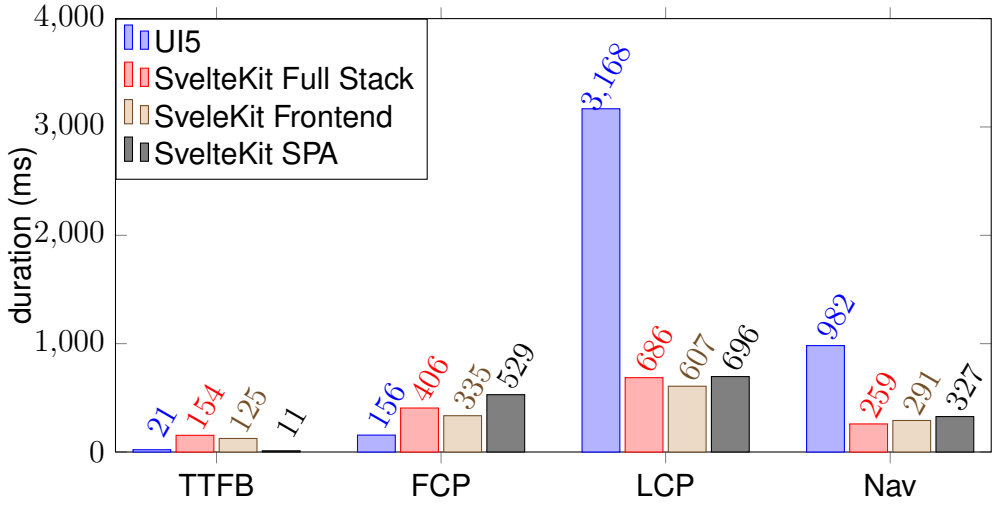


Figure 5.1: Benchmark Results for TTFB, FCP, LCP and Nav

from the recording by checking when all content has been rendered to the screen. The detailed benchmarking process is documented in Chapter 6.

All measurements were conducted on a Lenovo ThinkPad T480 with an Intel Core i7-8550U and 16 GB RAM running Windows 11. The measurements where taken using the developer tools of Google Chrome 116.0. We benchmarked the existing UI5 implementation, the SvelteKit full stack implementation (SK-FS), and the SvelteKit implementation, using SvelteKit primarily as a frontend framework, but redirecting requests through the SvelteKit server (SK-FE). Furthermore, we were interested how SvelteKit would fare when deploying it as static files in form of a classic SPA (SK-SPA).

The benchmark results shown in Figure 5.1 mostly confirm our expectations that SvelteKit would perform better in the LCP and navigation metrics. The results of SK-SPA are particularly interesting. As expected by forgoing SSR, it shows the fastest TTFB of all three SvelteKit implementations. But it takes slightly longer than the other two implementations to render its content. This confirms the assumption that SSR improves FCP outlined in Section 2.2. But despite its fast TTFB, the SPA implementation still has the slowest FCP of all implementations. This is a result of how SvelteKit loads its data. The UI5 implementation shows a loading spinner immediately after its FCP, before fetching data from the backend. The SK-SPA implementation on the other hand first awaits the results of all load functions

relevant to the current page before rendering anything to the screen. While this approach makes sense for a server rendered application, if the application is being rendered in the browser exclusively, this can mean a prolonged blank screen before data is shown. Nonetheless, in our implementation the load times proved to be fast enough to barely matter.

The Benchmarks also show that all SvelteKit implementations perform faster when navigating, compared to the UI5 implementation. Furthermore, SvelteKit provides out of the box support for preloading all required code and data for a page as soon as the user hovers over a link to that page. This can make page loads appear instantaneous to a user in some cases. For our benchmarks we disabled this feature to make comparisons more meaningful.

Overall, our performance analysis shows that SvelteKit can significantly improve initial page load times compared to UI5. This metric is most relevant for conversion rate [16], a metric for public facing-pages, such as e-commerce platforms. Nonetheless, fast page load times can contribute to the overall user experience for all types of applications.

5.2 Development Comparison

In this chapter we compare SvelteKit and UI5 in terms of ergonomics and developer experience. To this end, we show implementations of common use-cases for both frameworks.

5.2.1 Page Definitions

UI5 follows the model-view-controller (MVC) design pattern. A component consists of two files. The view file is usually written in XML and describes the UI layout. The behavior is defined using a controller file usually written in JavaScript. In Svelte on the other hand, layout and behavior is written in a single Svelte file (Section 2.3.1).

In UI5, the layout is written by nesting UI components in an XML file. It is not possible to use HTML directly, instead all HTML functionality has to be somehow handled by a UI5 component. In contrast, Svelte augments HTML with custom components

and special syntax. This approach makes it possible to fall back to HTML in cases where Svelte has not support for the given functionality. UI5 on the other hand, has to explicitly support every HTML feature through its components. Furthermore, knowledge resources about HTML are of limited use for UI5 development.

UI5 started development in 2008, thus the framework made design decisions for features such as imports long before they were standardized. As a result, import statements in UI5 feel outdated from a modern perspective (Listing 30) and do not integrate nicely with libraries that use the standard ESM syntax[1]. On the other hand, imports in Svelte follow the ESM syntax and thus integrates nicely with most modern libraries.

```
sap.ui.define([
  "sap/ui/core/Control",
  "sap/m/RatingIndicator",
  "sap/m/Label",
  "sap/m/Button"
], (Control, RatingIndicator, Label, Button) => {
  // ...
});
```

Listing 30: Example of UI5's import syntax.

5.2.2 Components

Creation of custom components is an essential feature in many modern UI Frameworks, It enables composition and reuse of isolated UI pieces such as buttons and inputs. As noted in Section 4.3.4, we initially tried to use UI5 web components as a component library in Svelte. But, after realizing multiple shortcomings with this approach, we pivoted to writing our own component library in Svelte using a predefined CSS style library. This provided multiple insights into Svelte's applicability for creating custom UI components.

In the following we show an example component that provides a button which triggers an alert when clicked. The alert shows a custom text that is passed to the control. Listing 31 shows a possible implementation in Svelte for such a component. The implementation is 7 lines of code long. Listing 32, shows a correspond-

[1] https://blogs.sap.com/2017/04/30/how-to-include-third-party-libraries-modules-in-sapui5/

ing implementation in UI5. With 19 lines of code, this implementation is immediately longer. Furthermore, because the code is split across two files, it is harder to understand its behavior in a single read.

AlertButton.svelte

```
<script>
  export let text;

  function onShowAlert() {
    alert(text);
  }
</script>

<button on:click={onShowAlert}>Show Alert</button>
```

Listing 31: Alert button implementation in Svelte.

AlertButton.control.xml

```
<core:FragmentDefinition
    xmlns="sap.m"
    xmlns:core="sap.ui.core"
    xmlns:layout="sap.ui.layout">
  <Button text="Show Alert" press="onShowAlert" />
</core:FragmentDefinition>
```

AlertButton.js

```
sap.ui.define(
  ['sap/ui/core/XMLComposite'],
  (XMLComposite) => {
  "use strict";

  return XMLComposite.extend("ui5.example.control.AlertButton", {
    metadata: {
      properties: { text: {type : "string" } }
    },

    onShowAlert() {
      alert(this.getText());
    },
  });
});
```

Listing 32: Alert button implementation in UI5.

On the other hand the UI5 implementation is stronger, because it uses a UI5 compo-

nent per default. The UI5 component is styled following the Fiori Design guideline, follows accessibility guidelines and ensures correct keyboard navigation behavior. In this regard UI5 can still save time, even tough the initial implementation code is more verbose. Nonetheless, the shown example is very simple. In complex use cases where UI5 does not provide prebuilt functionality, Svelte will likely pull ahead in terms of usability.

5.2.3 Routing

SvelteKit decided to integrate a filesystem-based routing system where files and directories in the file system have special meaning (Section 2.4.1). UI5 on the other hand uses a declarative routing approach. Routes are declared and linked to a view in a manifest file (Listing 33).

SvelteKit's approach comes with some inherent advantages and disadvantages. Probably the biggest advantages is that it is immediately clear what the purpose of a file is. Especially for simple cases this routing system works with no overhead and makes it possible to work very fast when adding new routes. But projects with a large amount of routes will inevitably have many files with the same name. In our experience this could make it harder to use IDE search functions to quickly navigate between many files because searching most of the time returned more than one `+page.svelte` or `+page.js` file. With UI5's approach, the route definitions are always found in the same place. While this means that the purpose of a file is not always immediately deducible, it is still easy enough to look up where a file is used. UI5's approach also provides more flexibility in structuring the project.

Another effect of SvelteKit's approach is that solutions to some problems can cause a lot of file changes in a version control system. During our implementation efforts, we found the need to wrap multiple routes in a layout group. This meant, all files hat to be moved to a new subdirectory. This caused a lot of noise in the version control system for a change that was simple in nature. In UI5 the same problem could have been solved by simple changing the router configuration.

It has also to be noted that this filesystem-based approach to routing makes it effectively impossible to split a singular SvelteKit application into multiple smaller projects. Because routing is strictly bound to the filesytem, it is only possible to

configure routing in the root project.

manifest.json

```
{
  "sap.ui5": {
    "routing": {
      "routes": [
        {
          "pattern": "",
          "name": "home",
          "target": "home"
        },
        {
          "pattern": "products/{productId}",
          "name": "productDetails",
          "target": "productDetails"
        }
      ],
      "targets": {
        "home": {
          "viewName": "Home",
        },
        "productDetails": {
          "viewName": "product/detail/ProducDetails"
        }
      }
    }
  }
}
```

Listing 33: Example routing configuration for a UI5 application with a Home view and a detail view for products.

5.3 Findings

In this section we discuss further aspects of SvelteKit we discovered during our study which were not easily comparable to UI5. These finding shall server as further help when deciding if SvelteKit is a good fit for a given use case.

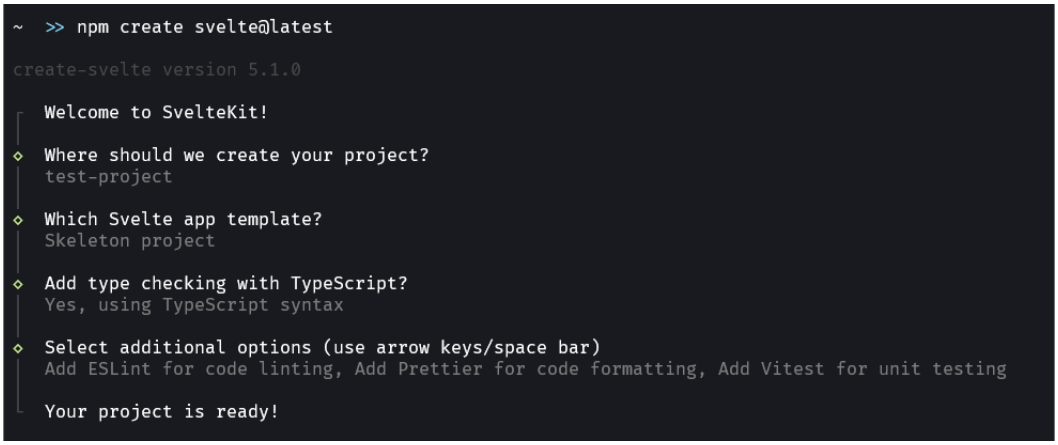


Figure 5.2: Tool for creating a new SvelteKit project

5.3.1 Project Setup

SvelteKit provides a command line utility to create new projects (Figure 5.2). The utility can be run with npm and provides a guided Wizard to select configuration options. This utility provides options to configure TypeScript for type checking, ESLint[2] for code linting, Prettier[3] for code formatting, as well as Vitest[4] and Playwright[5] for testing.

5.3.2 Component Libraries in Svelte

As described in Section 4.3.4, the implementation was required to follow the SAP Fiori Design Guidelines. To this end we tried using both UI5 web components and a plain CSS classes approach using SAP Fundamental Styles. In either case we decided to create a library of Svelte components that wraps the actual UI utility. This decision was made to reduce the amount of boilerplate code and repetition required to use some UI elements. Furthermore, this decision made it easier to migrate from UI5 web components to fundamental styles, because only the component library had to be changed.

[2] https://eslint.org/
[3] https://prettier.io/
[4] https://vitest.dev/
[5] https://playwright.dev/

Custom components can reduce the amount of code duplication by encapsulating UI elements into a reusable block. Therefore, components are an important tool in modern frontend design. Nonetheless, we noticed some shortcomings when working with custom components in Svelte. Notably, directives (Section 2.3.1) do not work on custom components. One might for example expose `class` and `style`-attributes on a component to make it possible to pass extra styling to it (Listing 34).

MyButton.svelte

```
<script>
  let clazz;
  export { clazz as class };
  export let style;
</script>

<button class={clazz} {style}><slot/></button>
```

App.svelte

```
<MyButton class="mt-2" style="background: red">Click me!</MyButton>
```

Listing 34: Svelte Component that provides a class and style attribute.

Because Svelte has no notion that these attributes correspond to `class` and `style`-attributes of an HTML-element, it is not possible to use directives to set these values. This means that the consumer of a component library is missing useful features provided by Svelte. Passing style classes especially caused further problems. As noted in Section 2.3.1, Svelte scopes style definitions to the component they are declared in. This means that it is not possible to pass styles down to a child component without explicitly declaring a style as global.

App.svelte

```
<MyButton class="highlight">Click me!</MyButton>

<style>
  .highlight:hover {
    filter: brightness(150%);
  }
</style>
```

Listing 35: The style will not apply to the custom component because it is scoped.

In Listing 35 a style is defined to highlight the button on hover. But because Svelte

scopes the rule to the component it is defined in, the style will not work. Svelte provides the possibility to mark style as global, but this means that every element in the application could potentially be affected by it. This can cause unintended side effects and is therefore not recommended. Another way to handle this issue would be to use libraries that are designed around global styles. Tailwind CSS[6] for example defines all its styling utilities as global CSS classes and therefore is not affected by this problem (Listing 36).

```
App.svelte
<MyButton class="hover:brightness-150">Click me!</MyButton>
```

Listing 36: A fixed version of Listing 35 that uses Tailwind CSS.

5.3.3 Universal vs. Server Load Functions

As noted in Section 4.3.3, we encountered multiple problems with the frontend only implementation that utilizes universal load functions. The primary problem is that SvelteKit provides no support for sending data to the backend. While data loading is streamlined through universal load functions, server actions, as used in the full stack implementation, can only be used server side. This means that submitting data has to be handled manually. Furthermore, submitting data will only work when JavaScript is enabled. This is likely the main reason why SvelteKit provides no support for this use case, because it prioritizes progressive enhancement.

Furthermore, using fetch in universal load functions requires usage of a special fetch function supplied by SvelteKit. This special fetch function makes sure that fetching data behaves the ame way on the client and server. Additionally, when a page with a universal load function is accessed, the load function is first run on the server during SSR, and the result is sent to the client. During hydration on the client side, the load function is then run again. This means, that ordinarily all fetch calls executed in the load function would run two times, once on the server and once on the client. To prevent this, SvelteKit's implementation inlines fetch responses on the server side, and then uses the inlined response on the client during hydration. This means, that it is advisable to use SvelteKit's fetch implementation. But, as

[6] https://tailwindcss.com/

this special fetch function is only exposed as an argument in load functions, one has to always pass this fetch function on any delegates, which require network communication. Furthermore, it is not possible to use custom HTTP clients, such as Axios[7].

5.3.4 Centralized Load Function

SvelteKit's architecture enforces that all asynchronous data has to be acquired inside a page's load function for it to be available during server side rendering. This provides a clean and easy to understand flow for simple implementations. But it can increase complexity in cases where a single isolated component has to be reused in multiple different routes.

One could for example imagine a simple component which displays a Twitter feed, which has to be used in multiple places. Listing 37 outlines the directory structure for an example where both route a and b need to use a shared component that fetches and renders a Twitter feed. The way to implement this in SvelteKit would be to fetch the data in the load function of both routes and pass it to the twitter component as a prop (Listing 38).

```
routes/
  a/
    +page.js
    +page.svelte
  b/
    +page.js
    +page.svelte
  TwitterComponent.svelte
```

Listing 37: Directory hierarchy that has a reusable component.

[7] https://axios-http.com/

```
a/+page.js b/+page.js
export async function load({ fetch }) {
  const twitterFeed = (await fetch('/api/twitterfeed')).json();

  return {
    twitterFeed,
    // other page data...
  }
}
```

```
a/+page.svelte b/+page.svelte
<script>
  import TwitterComponent from '../TwitterComponent.svelte';

  export let data;
</script>

<TwitterComponent feed={data.twitterFeed} />
<!-- other page markup... -->
```

```
TwitterComponent.svelte
<script>
  export let feed
</script>

{#each feed as tweet}
  <!-- ... -->
{/each}
```

Listing 38: Example Implementation of a reusable component in SvelteKit

This pattern needs to be repeated for each route that wants to use the twitter component. This not only results in unnecessary code duplication, but can also cause problems later in a project's lifecycle. If the twitter component is to be removed later on, it is imaginable that one of the fetch calls is left in by accident, because the data fetching is detached from where the data is actually used, thus causing unnecessary traffic.

Other approaches circumvent this problem by coupling data fetching more closely to its usage. For example, a similar implementation using React could fetch the required data inside the component (Listing 39).

TwitterComponent.jsx

```jsx
export function TwitterComponent() {
  const [twitterFeed, setTwitterFeed] = useState([]);
  useEffect(() => {
    fetch('/api/twitterfeed')
      .then(res => res.json())
      .then(data => setTwitterFeed(data))
  }, []);

  return <>
    {twitterFeed.map(tweet => (
      // ...
    ))}
  </>
}
```

a/page.jsx b/page.jsx

```jsx
import TwitterComponent from '../TwitterComponent';

export async function Page() {
  return <>
    <TwitterComponent />
    { /* other page markup... */ }
  </>
}
```

Listing 39: Example of a reusable component using React server components.

With this approach, the Twitter component becomes truly isolated from the page, because it can fetch its own data. On the other hand, because SvelteKit's approach enforces such a strict architecture, it is immediately clear where data fetching happens. This can have benefits when trying to understand existing code bases.

5.3.5 Reactivity Pitfalls

While Svelte's reactivity is overall very intuitive, it nonetheless has some pitfalls.

Svelte's reactive statements (Section 2.3.1) work by analyzing at compile time which variables are referenced inside a reactive statement. These variables are tracked as dependencies and every time one of these dependencies changes, Svelte reevaluates the reactive statement. But in cases where the dependency is used indirectly, for example through a function call, the dependency cannot be seen by Svelte's

compiler. Therefore, updates to the dependency will not trigger reevaluation of the reactive-statement.

```
<script>
  let count = 1;

  $: doubled = calcDoubled();

  function calcDoubled() {
    return count * 2;
  }
</script>

<button on:click={() => (count++)}>+</button>
<div>{count} * 2 = {doubled}</div>
```

Listing 40: `doubled` will not be recalculated when `count` changes.

Listing 40 illustrates this behavior. In this example, the function `calcDoubled` calculates the value of `doubled` in correspondence to `count`. But, because `count` is not directly referenced in the reactive statement, it is not tracked as a dependency. Therefore, `doubled` will not be updated when `count` changes. To fix this example, `count` needs to be passed as an explicit parameter to this function. In this way the Svelte compiler can detect `count` as a dependency of `doubled` and will create the correct update statements (Listing 41). Section 5.3.8 will discuss a future update to Svelte that would fix this problem.

```
<script>
  let count = 1;

  $: doubled = calcDoubled(count);

  function calcDoubled(c) {
    return c * 2;
  }
</script>
```

Listing 41: Fixed version of Listing 40.

5.3.6 Ecosystem Maturity

The original implementation uses Keycloak with OIDC as authorization mechanism. The SvelteKit frontend and SPA implementations had to therefore implement authentication with the Keycloak service. To log in with Keycloak, the user is first redirected to the Keycloak login page, where they have to then enter their credentials. After successful login, the user is redirected back to the application with an access token. This access token can then be used to authenticate with the Spring backend. In the SvelteKit frontend implementation the SvelteKit server is sending the actual requests to the Spring backend whereas in the SPA implementation, backend requests are sent client-side. This difference made it impossible to use the same authentication library for both implementations.

To integrate authentication into the SPA implementation we used `oidc-client-ts` [8], a stable and battle tested library. Thanks to Svelte's adherence to web standards, the library was easily hooked up with SvelteKit, without any major problems. On the other hand, authentication in the SvelteKit frontend implementation requires stronger integration with SvelteKit itself because the backend needs to be able to use the users access token to communicate with the Spring backend while also handling authentication between SvelteKit client and server. To this end, we used Auth.js[9] a general purpose authentication solution with support for many authentication providers. As of this writing, SvelteKit support for Auth.js is still experimental. This became apparent, because Auth.js did not implement features such as automatic access token refresh.

This highlights another important point about SvelteKit. The framework tries to adhere to web standards where possible. Because of this, it is often possible to use general purpose JavaScript libraries without any special adjustments. On the other hand, in cases where tighter integration with SvelteKit is required SvelteKit's relative novelty becomes apparent in the absence of mature and well-developed libraries for certain use cases.

[8] `https://www.npmjs.com/package/oidc-client-ts`
[9] `https://authjs.dev/`

5.3.7 Deployment

SvelteKit provides first-class adapters that automate creating production deployments for various hosting providers such as Vercel[10], Netlify[11], Cloudflare Pages[12], and Cloudflare Workers[13]. Furthermore, adapters exist to create standalone Node.js server, or generate a static bundle that can be served from a simple file-base web server. Beyond this, a wide range of community plugins exist which further extend the range of platforms SvelteKit can be deployed to. This means that SvelteKit generally supports great flexibility for deployment with comparatively little effort.

But this wide range of supported platforms has some shortcomings. SvelteKit does mostly not implement features that require platform specific functionality. One such feature would be a hook that is executed when the application shuts down. This can sometimes be required for example to close open connections. While this can be easily worked around in a Node.js environment, for example by using `process.on('sigint', ...)`, other features are not added this easily.

SvelteKit wants to be a serverless framework [17]. With many serverless platforms not having support for websockets yet [18], websockets is a feature that is not yet supported in SvelteKit out of the box. While it is possible to add websocket support to the Node.js platform, this has limitations. By default, the Node.js adapter creates an entry point that starts a web server to host all handlers required by SvelteKit. It is possible to forgo this entry point and instead write a custom server that uses the handlers directly. In this way it is possible to create a web server that uses websockets. But, this approach only works for the production build. In development another workaround is required that works with hot-module replacement.

5.3.8 Future of Svelte

In September 2023, the Svelte team gave a first preview of the next major API planned for version 5 of Svelte [19]. The headline feature of this new iteration is called runes. Runes are a set of compiler instructions that replace the system of

[10] https://vercel.com/
[11] https://www.netlify.com/
[12] https://pages.cloudflare.com/
[13] https://workers.cloudflare.com/

reactivity currently used in Svelte (Section 2.3.1). With runes, Reactivity becomes more explicit. Instead of every top level variable in a svelte file being reactive by default, state that should be reactive has to be marked with the `$state` rune (Listing 42).

```
<script>
  let count = $state(0);
</script>
```

Listing 42: The `$state` rune will be used in Svelte 5 to mark a variable as reactive.

While initially being more verbose, this has multiple advantages. Firstly, this syntax is more explicit, clearly marking which variable is reactive and which is not. Secondly, in the current reactivity system of Svelte only top-level variables are reactive. With runes, it will be possible to mark variables as reactive, anywhere in the code. Further, it will even be possible to place state runes in JavaScript files. This will make it easier to move out state into separate files. Whereas before, large rewrites were necessary to move state logic out of a Svelte file, with this new syntax it will be possible to use the same syntax in Svelte files and in JS files.

Furthermore, Svelte 5 also introduces the `$derived` and `$effect` runes which replace the `$:` syntax (Listing 43). In the current version of Svelte, `$:` is used to handle two different features, reactive binding of variables and defining side effects.

```
<script>
  let count = $state(0);

  const doubled = $derived(count * 2);

  $effect(() => {
    console.log(`new value of count: ${count}`);
  })
</script>
```

Listing 43: Example usage of the `$derived` and `$effect` runes.

The new rune system will also solve the problems outlined in Section 5.3.5. Currently, reactive dependencies are tracked at compile-time. This makes it impossible

to track indirect dependencies. Runes on the other hand, will track their dependencies at run-time, therefore making it possible to also track indirect dependencies. This will make Svelte's system for reactivity overall easier to reason about.

5.3.9 Stability

Historically, the Svelte team did not shy away from innovating the framework. Both Svelte 2 and Svelte 3 introduced major breaking changes [20, 21]. Furthermore, with runes, Svelte is already receiving the next major breaking change. The old system for reactivity is planned for removal with Svelte 7. In the meantime, Svelte will support both syntaxes in the same code base on a per component basis. This gives projects time to migrate to runes. Nonetheless, especially in large code bases this can cause costly migrations. For past breaking changes the Svelte team provided tools to automate much of the migration work [20]. They aim to do the same for runes, hopefully minimizing the effort required to migrate.

Version 1.0 of SvelteKit was released only recently in December 2022 [22]. It is not possible to say how stable SvelteKit's APIs will be going forward. If Svelte's history is any indication, SvelteKit will probably also receive major API overhauls, should the need arise. Historically, Svelte has proven to be fast moving, therefore projects that use SvelteKit should be ready to put in regular migration work, to prevent the code base from becoming outdated.

6 Discussion and Future Work

In this work we investigated SvelteKit's applicability for the development of enterprise applications. To this end, we chose a business application used productively in a real-world scenario. We reimplemented this application in SvelteKit using a full stack, single-page application, and a frontend only approach. Furthermore, we compared the existing implementation, written in UI5 and Java, with our SvelteKit implementations in terms of performance and development effort. Finally, we provided insights into SvelteKit which can help in making future technology choices.

Our results show that all of our SvelteKit implementations performed significantly better in terms of page load and navigation speed. Especially when used with server side rendering, SvelteKit was able to achieve the fastest page load time. Notable outlier was the single page application variant that, while providing the fastest time to first byte, required the longest time out of the SvelteKit implementations to render any content to the screen. This is a result of SvelteKit's data fetching mechanism which is not suited for deferred rendering of content.

In terms of development effort our comparative analysis provides multiple examples where SvelteKit is expected to be easier to use. This is a result of Svelte's concise syntax and feature set which aims to ease declaration of user interfaces. Furthermore, Svelte's colocation between UI and controlling logic provides a clearer and easier system to reason about. Finally, Svelte's language tools provided better support for working with the language. Types are automatically inferred across most of SvelteKit's API's, making it easy to work with growing code bases. This is contrasted by UI5's vast set of features. Especially UI5's large component library provides capabilities for many often required features such as analytical tables.

SvelteKit only recently reached its first stable release and is still being actively developed. This means that some key features such as native websocket and localization support are not provided yet. Furthermore, Svelte's development team has in the

past not shied away from overhauling major parts of the language (Section 5.3.9). SvelteKit's library ecosystem also has not reached maturity. In our study we were not able to find a library that provided full out-of-the-box support for authentication through OIDC.

In our opinion, SvelteKit makes it easy to develop custom UI components. This makes SvelteKit especially suited for use-cases where a custom design system is desired, or pre-built components cannot satisfy all feature requirements. Furthermore, SvelteKit's flexible rendering model makes it suitable for small projects where it is feasible to colocate frontend and backend code, as well as larger projects with separated backend. When considering SvelteKit for a new project it furthermore has to be evaluated if SvelteKit's ecosystem provides mature solutions for the projects requirements.

Future work could further investigate SvelteKit's developer experience in the form of user surveys which was out of scope for this work. Furthermore, it is not clear how SvelteKit compares to contemporary meta frameworks, such as Next.js, Astro, and Remix. To this end, future work could compare SvelteKit to these frameworks.

Appendix

Benchmark Setup

1. Navigate application to the chauffeur job overview page.
2. Start a new performance measurement in Chrome.
3. Reload the page.
4. Wait until page has settled.
5. Click on a predefined job in the overview.
6. Wait until page has settled.
7. Stop performance measurement.
8. Write down measurements reported by the web vitals plugin.
9. Search for first frames where content was visible after navigation in the performance view.
10. Search for pointer down and pointer up events.

Bibliography

[1] *What is Enterprise Software? - Everything you need to know - AWS*. URL: https://aws.amazon.com/what-is/enterprise-software/ (visited on 09/27/2023).

[2] Vangie Beal. *What is An Enterprise Application?* May 2010. URL: https://www.webopedia.com/definitions/enterprise-application/ (visited on 09/27/2023).

[3] OfferZen Origins. *Svelte Origins: A JavaScript Documentary*. June 2022. URL: https://www.youtube.com/watch?v=kMlkCYL9qo0 (visited on 08/15/2023).

[4] Engineer Bainomugisha et al. "A survey on reactive programming." In: *ACM Computing Surveys* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: https://dl.acm.org/doi/10.1145/2501654.2501666 (visited on 10/12/2023).

[5] Tom Occhino and Jordan Walke. *JS Apps at Facebook*. Aug. 2013. URL: https://www.youtube.com/watch?v=GW0rj4sNH2w (visited on 10/12/2023).

[6] *Stack Overflow Developer Survey 2023*. en. URL: https://survey.stackoverflow.co/2023/ (visited on 10/31/2023).

[7] *[Proposal] Run js expressions in markup template through Svelte script preprocessor code · Issue #4701 · sveltejs/svelte*. URL: https://github.com/sveltejs/svelte/issues/4701 (visited on 10/12/2023).

[8] *SvelteKit • Web development, streamlined*. en. URL: https://kit.svelte.dev/ (visited on 09/30/2023).

[9] Andread Kurz. *A Brief History of OpenUI5 (and SAPUI5) | SAP Blogs*. Nov. 2020. URL: https://blogs.sap.com/2020/11/04/a-brief-history-of-openui5-and-sapui5/ (visited on 10/24/2023).

[10] Andreas Kunz. *What is OpenUI5 / SAPUI5 ? | SAP Blogs*. Dec. 2013. URL: https://blogs.sap.com/2013/12/11/what-is-openui5-sapui5/ (visited on 10/24/2023).

[11] Jenny Morales et al. "Programmer eXperience: A Systematic Literature Review." In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 71079–71094. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2920124.

[12] Teodor-Dorin Tripon, Gianina Adela Gabor, and Elisa Valentina Moisi. "Angular and Svelte Frameworks: a Comparative Analysis." In: *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. June 2021, pp. 1–4. DOI: 10.1109/EMES52337.2021.9484119.

[13] Mattias Levlin. *DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte*. 2020. URL: https://www.doria.fi/handle/10024/177433.

[14] Sacha Greif and Eric Burel. *State of JavaScript 2022*. en. 2022. URL: https://2022.stateofjs.com/en-US/ (visited on 04/11/2023).

[15] *web.dev*. URL: https://web.dev/ (visited on 10/09/2023).

[16] *Load time to conversion statistics*. URL: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/load-time-to-conversion-statistics/ (visited on 10/23/2023).

[17] *Expose a way to inject middleware in the server pipeline for adapt-node · Issue #334 · sveltejs/kit*. URL: https://github.com/sveltejs/kit/issues/334 (visited on 09/28/2023).

[18] *Do Vercel Serverless Functions support WebSocket connections?* URL: https://vercel.com/guides/do-vercel-serverless-functions-support-websocket-connections (visited on 09/28/2023).

[19] The Svelte team. *Introducing runes*. Sept. 2023. URL: https://svelte.dev/blog/runes (visited on 09/29/2023).

[20] Rich Harris. *Svelte v2 is out!* Apr. 2018. URL: https://svelte.dev/blog/version-2 (visited on 11/01/2023).

[21] Rich Harris. *Svelte 3: Rethinking reactivity*. Apr. 2019. URL: https://svelte.dev/blog/svelte-3-rethinking-reactivity (visited on 03/07/2023).

[22] The Svelte team. *Announcing SvelteKit 1.0*. Dec. 2022. URL: https://svelte.dev/blog/announcing-sveltekit-1.0 (visited on 03/07/2023).

Name: Hannah Lappe Matrikelnummer: 922114

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 02.11.23 Lappe

Hannah Lappe