HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Svelte

Projektarbeit an der
Hochschule Ravensburg-Weingarten

Studiengang Angewandte Informatik

Autor: Batdelger Davaajav
Matrikelnummer 27438

Version vom: 28. Februar 2020

Betreuer: Prof. Dr. Marius Hofmeister

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AST – Abstract Syntax Tree

NPM – Node Package Manager

REST - Representational State Transfer

SPA – Single Page Application

1 Einführung

Diese Projektarbeit beschäftigt sich mit dem neuen Framework Svelte (https://svelte.dev). Zunächst wird Svelte kurz vorgestellt, die Funktionsweise und die Performance mit anderen Frameworks verglichen und auf die Geschichte von Svelte eingegangen.

1.1 Was ist Svelte

Svelte ist ein Framework, um das Erstellen von grafischen Benutzeroberflächen für Webanwendungen zu erleichtern. Der Programmcode besteht zunächst aus den im Frontend verbreiteten Sprachen HTML, CSS und JavaScript. Diese werden von Svelte durch eine zusätzliche Syntax erweitert. Damit lässt sich der Programmcode einer grafischen Benutzeroberfläche in Komponenten gliedern. Dies erleichtert die Programmierung vor allem bei Single Page Applications, bei denen beim Seitenaufruf einmalig die komplette Benutzeroberfläche geladen wird und weitere Inhalte durch spätere Serveraufrufe ergänzt werden.

Ein weiteres wichtiges Konzept von Svelte gegenüber gewöhnlichem JavaScript („VanillaJS“), ist die Reaktivität des DOMs. Das bedeutet, dass bei Änderungen des Zustands einer Komponente das DOM automatisch aktualisiert wird. Ohne Reaktivität müssten diese Aktualisierungen sonst vom Programmierer zusätzlich eingefügt werden.

Auch die Verwendung von Kontrollstrukturen direkt im HTML-Code um die DOM-Aktualisierung zu steuern vereinfacht die Arbeit des Programmierers.

Weiterhin bietet Svelte eine ganze Reihe Möglichkeiten, um unkompliziert Animationen zu HTML-Elementen hinzuzufügen.

Svelte bringt nicht nur Vorteile beim Programmieren, sondern auch eine höhere Performance des Codes. Dies soll in den nächsten Kapiteln genauer beschrieben werden.

1.2 Funktionsweise von Svelte

Frontend-Frameworks bringen zwar dem Programmierer viele Vereinfachungen beim Erstellen des Quelltextes, jedoch müssen dabei an anderer Stelle diese Vereinfachungen wieder auf die vom Browser verständlichen Sprachen HTML, CSS und JavaScript abgebildet werden. Bei den meisten bekannten Frameworks passiert dies im Browser selbst während der Nutzung der Anwendung. Das Framework wird dabei üblicherweise in einer zusätzlichen Bibliothek an den Browser ausgeliefert.

Um Zustandsänderungen im JavaScript-Code mit dem DOM zu synchronisieren, behelfen sich einige Frameworks mit einem virtuellen DOM. Das echte DOM wird dabei auf JavaScript-Objekte abgebildet und nur diese werden vom Programmierer manipuliert. Das virtuelle DOM merkt sich dabei, welche Elemente beim nächsten Update im echten DOM aktualisiert werden müssen. Es wird dann nur der Teil aktualisiert, der im virtuellen DOM auch geändert worden ist.

Svelte geht dabei einen anderen Weg. Anstatt während der Laufzeit über ein virtuelles DOM herauszufinden, welche echten DOM-Elemente manipuliert werden müssen, untersucht dies bei Svelte ein Compiler. Der Compiler übersetzt die Syntax von Svelte in reinen JavaScript-Code. Für notwendige DOM-Aktualisierungen wird dann vom Compiler entsprechender Programmcode ergänzt. Bis auf ein paar wenige Zusatzfunktionen muss keine extra Bibliothek vom Browser ausgeliefert werden. Es entsteht weder beim Starten der Anwendung noch während der Laufzeit ein zusätzlicher Aufwand. (1)

Die folgende Grafik verdeutlicht dies.

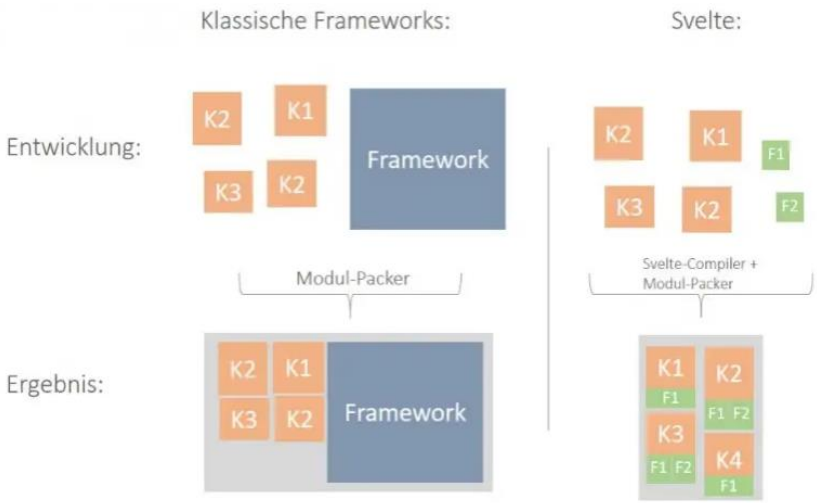

Abbildung 1: Funktionsweise von Svelte

(2)

1.3 Verbreitung

Ein wichtiges Kriterium bei der Auswahl eines Frameworks ist auch seine Verbreitung. Denn je mehr ein Framework von anderen Entwicklern verwendet wird, desto mehr Hilfestellung findet sich auch im Internet. Svelte ist ein eher neues Framework und im Vergleich zu anderen noch nicht oft im Einsatz.

Die folgende Grafik zeigt die Anzahl der GitHub-Downloads seit dem Jahr 2015 von Angular, React, Vue und Svelte. (3)

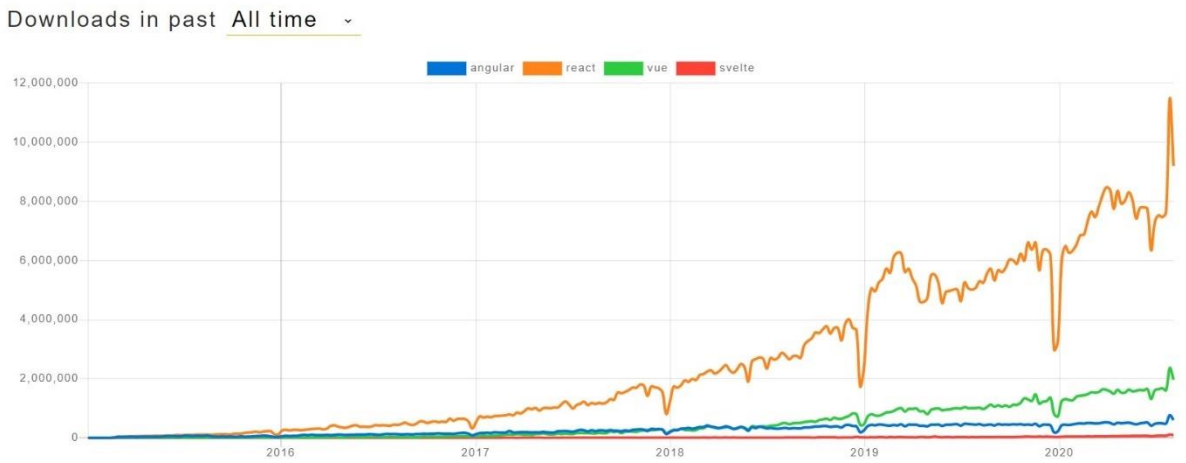

Abbildung 2: Anzahl der GitHub-Downloads seit dem Jahr 2015

Von den bekannten Frameworks wird React scheinbar am meisten verwendet. Vue hat Angular überholt. Svelte ist noch nicht sehr verbreitet.

Betrachtet man die Entwicklung von Svelte für sich allein, erkennt man in untenstehender Abbildung wie sich die Verbreitung von Svelte in den letzten ein bis zwei Jahren entwickelt hat. In den Kategorien „Top 10k Seiten“, „Top 100k Seiten“ und „Top 1m Seiten“ des Internets hat sich der Einsatz seit Februar 2020, also innerhalb eines Jahres, etwa vervierfacht.

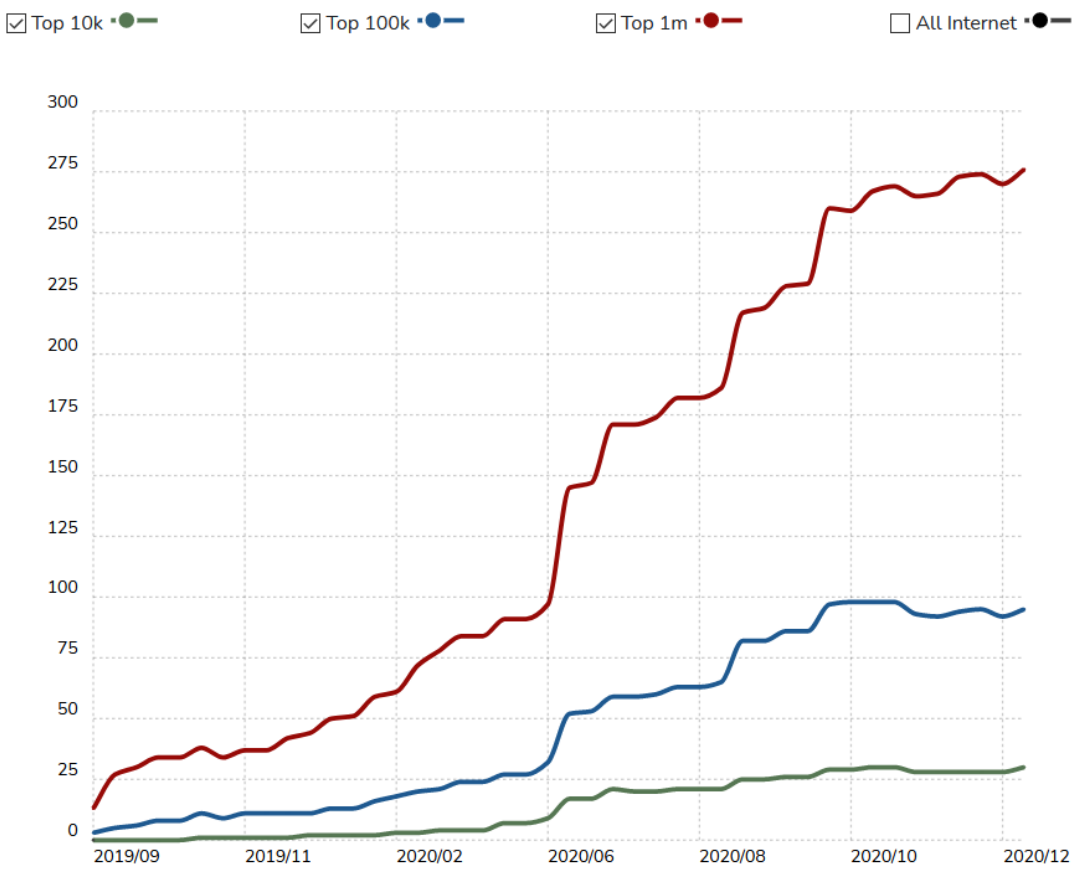

Abbildung 3: Einsatz von Svelte unter den Top 10k, Top 100k und Top 1m Seiten.

Die geringe Verbreitung wird bei Svelte oft als Nachteil genannt. Dieser Nachteil ist jedoch kein technisches Problem und wird geringer werden, wenn mehr Programmierer Svelte verwenden.

1.4 Performance

Die Performance eines Frameworks ist auch ein wichtiges Argument bei der Auswahl. Hier soll Svelte nach drei Kategorien mit anderen Frameworks verglichen werden:

- Geschwindigkeit bei der Aktualisierung des DOMs
- Geschwindigkeit, bis die App im Browser geladen ist
- Speicherbedarf

Ryan Carniato hat im Dezember 2020 auf der Blogseite „JavaScript in plain English“ die Performance von 20 JavaScript-Frameworks verglichen. Im Folgenden wird Svelte den drei sehr bekannten Frameworks „Angular“, „React“ und „Vue3“ gegenübergestellt. Außerdem dient pures JavaScript („VanillaJS“) als Referenz. Dabei wurden die gemessenen Werte immer prozentual mit „VanillaJS“ verglichen und in jeder Kategorie der Mittelwert gebildet. (4)

Beim Vergleich der Kategorie „Geschwindigkeit der DOM-Aktualisierung“ wurden neun verschiedene Operationen auf einer Tabelle durchgeführt. Dies waren z.B. Operationen wie das Einfügen von 10000 Zeilen. Die Mittelwerte der Abweichungen zu „VanillaJS“ sind in folgender Tabelle dargestellt.

VanillaJS	Svelte	Vue3	React	Angular
1	1,28	1,51	1,96	2,05

Tabelle 1: Geschwindigkeit der DOM-Aktualisierung

Das bedeutet, dass Svelte im Vergleich zu „VanillaJS“ im Durchschnitt zu 28% langsamer, aber schneller als die Vergleichskandidaten ist.

Auch wurde die Zeit verglichen, die es dauert, bis eine Test-App im Browser geladen ist. Hier spielten drei Faktoren eine Rolle. Die Größe des an den Browser ausgelieferten Code, die Zeit zum Parsen der Skripte und der Zeitpunkt nach dem sich die Netzwerklast und die CPU wieder im Ruhezustand befindet. Wieder sind die Abweichungen im Vergleich zu „VanillaJS“ dargestellt.

VanillaJS	Svelte	Vue3	React	Angular
1	1,02	1,15	1,37	2,80

Tabelle 2: Zeit zum Laden der App im Browser

Die letzte Kategorie ist der Speicherbedarf. Es wurden vier Operationen auf einer Tabelle durchgeführt sowie der Speicherbedarf direkt nach dem Laden der Test-App untersucht. Die Abweichungen zu „VanillaJS“ finden sich in folgender Tabelle.

VanillaJS	Svelte	Vue3	React	Angular
1	1,34	1,61	1,84	2,07

Tabelle 3: Speicherbedarf

Der Vergleich gibt nur einen groben Eindruck, dass die Performance von Svelte im Vergleich zu anderen bekannten Frameworks sehr gut ist.

1.5 Lesbarkeit der Syntax

Ein Framework mit hoher Performance ist nicht viel wert, wenn es dem Programmierer schwer fällt Code zu schreiben. Auch die Lesbarkeit ist wichtig, da Code deutlich häufiger gelesen als geschrieben wird. Mit Svelte ist es dabei möglich mit erstaunlich wenig Codezeilen reaktive Komponenten zu erzeugen.

Im Folgenden wird die Lesbarkeit der Syntax von Svelte mit Vue, React und Webkomponenten an einem Beispiel verglichen. Als Beispiel dient die Addition zweier Zahlen aus zwei Textfeldern. Die Ausgabe soll dabei direkt auf die Eingabe reagieren, d.h. das Ergebnis soll ohne weitere Aktion des Benutzers sofort angezeigt werden. (5)

Abbildung 4: Beispielkomponente zum Vergleich der Lesbarkeit

Wird das Beispiel mit Webkomponenten realisiert, so müssen die „input“-Listener für die Textfelder und die nach einer Eingabe nötige Aktualisierung des Ergebnisses von Hand programmiert werden. Dafür werden ohne Einrückungen und Leerzeilen etwa 567 Zeichen benötigt.

```
class Add extends HTMLElement {
  constructor() {
  super();
    this.innerHTML = `
      <input class="a" type="number" value="1">
      <input class="b" type="number" value="2">
      <p></p>
    `;
    this.querySelector(".a").oninput = () => this.update();
    this.querySelector(".b").oninput = () => this.update();
    this.update();
  }

  update() {
    let a = this.querySelector(".a").value;
    let b = this.querySelector(".b").value;
    let result = parseInt(a) + parseInt(b);
    this.querySelector("p").textContent = a + " + " + b + " = " + result;
```

```
  }
}
customElements.define("add-test", Add);
```

Etwas weniger, nur 442 Zeichen, werden mit React benötigt. Die Handler-Funktionen müssen jedoch programmiert werden.

```
import React, { useState } from 'react';

export default () => {
  const [a, setA] = useState(1);
  const [b, setB] = useState(2);

  function handleChangeA(event) {
    setA(+event.target.value);
  }

  function handleChangeB(event) {
    setB(+event.target.value);
  }

  return (
    <div>
      <input type="number" value={a} onChange={handleChangeA}/>
      <input type="number" value={b} onChange={handleChangeB}/>

      <p>{a} + {b} = {a + b}</p>
    </div>
  );
};
```

Noch weniger benötigt Vue mit nur 263 Zeichen. Die Handler-Funktionen müssen dank Binding nicht programmiert werden.

```
<template>
  <div>
    <input type="number" v-model.number="a">
    <input type="number" v-model.number="b">

    <p>{{a}} + {{b}} = {{a + b}}</p>
  </div>
</template>

<script>
  export default {
    data: function() {
```

```
      return {
        a: 1,
        b: 2
      };
    }
  };
</script>
```

Schließlich schafft es Svelte mit nur 145 Zeichen. Zustandsänderungen werden durch Zuweisungen automatisch erkannt. Durch das Binding entfallen die Handler-Funktionen.

```
<script>
   let a = 1;
   let b = 2;
</script>

<input type="number" bind:value={a}>
<input type="number" bind:value={b}>

<p>{a} + {b} = {a + b}</p>
```

Ein kürzerer Quellcode führt auch zu weniger Bugs

Fairerweise muss man aber sagen, dass das Beispiel so gewählt wurde, so dass Svelte seine Stärken voll ausspielen kann.

1.6 Versionen

Die aktuelle Version von Svelte ist die Version 3 vom April 2019. Diese Version bringt viele Verbesserungen bei der Syntax im Vergleich zu den beiden Versionen 1 und 2, die eher das Ziel hatten, das Compiler-Konzept von Svelte zu testen. (6)

2 Infrastruktur

Dieses Kapitel soll einen Überblick über die Infrastruktur von Svelte geben. Der typische Workflow von der Installation von Svelte bis zur Veröffentlichung der Webseite für Benutzer soll dargestellt werden.

2.1 Installation

Svelte ist integriert in das Ökosystem von node.js und lässt sich somit über den Paketmanager npm installieren. Auf der offiziellen Webseite wird ein Starterprojekt angeboten. Dies lässt sich einfach als zip-Datei herunterladen und entpacken oder mit Hilfe von degit mit folgender Befehlszeile in ein beliebiges lokales Projektverzeichnis klonen:

```
degit sveltejs/template my-svelte-project
```

Anschließend lassen sich mit dem Befehl `npm install` über ein Terminal alle Abhängigkeiten installieren. Die Abhängigkeiten stehen dabei in der Datei „package.json“.

2.2 Ordnerstruktur

Nach der Installation des Starter-Projekts enthält das Projektverzeichnis vier Unterordner: „node_modules“, „src“, „scripts“ und „public“.

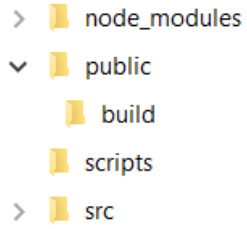

Abbildung 5: Ordnerstruktur eines Svelte-Projekts

Unter „node_modules“ finden sich alle heruntergeladenen abhängigen Programmpakete. Unter anderem Svelte selbst, der Modul-Bundler „rollup“ sowie der Entwicklungsserver „sirv“ sind hier enthalten. Daneben gibt es noch eine Reihe weiterer Pakete, die von Svelte benutzt werden.

Im Ordner „src“ befinden sich alle vom Entwickler geschriebenen Svelte-Komponenten, sowie weiterer Quellcode, der sich nicht einer Komponente zuordnen lässt. Eine Svelte-Komponente befindet sich dabei immer in genau einer Datei mit dem Namen der Komponente und der Erweiterung „.svelte“. Auch können zur weiteren Ordnung der Komponenten Unterverzeichnisse erstellt werden.

Das Verzeichnis „scripts“ bietet Platz für externe JavaScript-Bibliotheken, die der Entwickler zusätzlich verwendet.

Und schließlich werden im Ordner „public“ alle Dateien abgelegt, die vom Browser öffentlich zugänglich sind. Hier befindet sich die „index.html“-Datei, die von jedem Browser zuerst geladen wird. Auch können hier z.B. Bilder und weitere CSS-Dateien wie „global.css“ oder andere Ressourcen liegen, die für die Funktionalität der Webseite wichtig sind. Außerdem existiert ein Unterordner „build“, in dem der vom Svelte-Compiler generierte und von „rollup“ gebündelte Code in einer Datei „bundle.js“ erstellt wird. Die „index.html“ lädt dann beim Öffnen der App das Skript „build/bundle.js“.

2.3 Entwicklungsprozess

Zum Entwickeln wird neben npm zusätzlich ein Texteditor benötigt. Es empfiehlt sich aber einen speziellen Code-Editor wie z.B. Visual Studio Code zu verwenden. Für diesen existiert auch ein Plugin, das Syntax-Highlighting für Svelte anbietet. Zum Testen und Debuggen des Programmcodes kann der Entwickler einen modernen Browser wie z.B. Chrome oder Firefox verwenden.

Um die Anwendung zu erzeugen und auf einem Server verfügbar zu machen, sind die folgenden Schritte nötig:

Quellcode -> Compiler -> Bundle -> Start/Reload Entwicklungsserver

Der Svelte-Quellcode wird zuerst kompiliert und dann vom Modul-Bundler „rollup“ zu den Dateien „bundle.js“ und „bundle.css“ zusammengefasst. Die Dateien werden im Verzeichnis „public/build“ erzeugt. Durch Erstellen dieses Paketes muss der Browser nur eine einzige JavaScript-Datei und eine einzige CSS-Datei herunterladen. Dies beschleunigt das Starten der Anwendung. Dabei gibt es zwei Varianten: In der Entwicklungsversion ist der kompilierte Code weiterhin für Menschen lesbar. In der Produktionsversion ist der kompilierte Code auf ein Minimum reduziert, um das Starten der Webseite noch weiter zu beschleunigen. Auch erschwert der minimierte Code das Reengineering des Quelltextes.

Ist das Bundle erstellt, wird anschließend der Entwicklungsserver „sirv“ gestartet. Nun kann die Anwendung mit einem Browser genutzt werden.

Der gesamte Vorgang lässt sich entweder mit dem Befehl `npm run dev` für die Entwicklungsversion oder mit dem Befehl `npm run build` für die Produktionsversion starten.

Svelte verwendet standardmäßig als Modul-Bundler „rollup“ und als Entwicklungsserver „sirv“. Es kann aber auch ein anderer Bundler wie z.B. „webpack“ verwendet werden.

In der Konfigurationsdatei „package.json“ werden im Abschnitt „scripts“ die verwendeten Tools konfiguriert.

```
"scripts": {
    "build": "rollup -c",
    "dev": "rollup -c -w",
    "start": "sirv public"
}
```

„rollup“ und „sirv“ bieten außerdem sogenanntes Hot-Code-Reloading. Dies hat den Vorteil, dass bei Änderungen im Quellcode das generierte Bundle sofort auf den neuesten Stand gebracht wird.

Soll die entwickelte Anwendung veröffentlich werden, kann der Inhalt des „public“-Verzeichnis von einem beliebigen Webserver angeboten werden.

2.4 Debugging

Der Svelte-Quellcode kann dank der vom Compiler erstellten Sourcemap direkt im Browser analysiert werden. Eine Sourcemap ist eine Datei die den Sourcecode, den der Entwickler geschrieben hat, also unter anderem die Svelte-Komponenten rekonstruieren kann. Die Datei hat den Namen „bundle.js.map“ und befindet sich im gleichen Verzeichnis wie die Datei „bundle.js“.

Im Browser können dann nicht nur im JavaScript-Code, sondern auch bei im HTML-Code enthaltenen Anweisungen Breakpoints gesetzt werden.

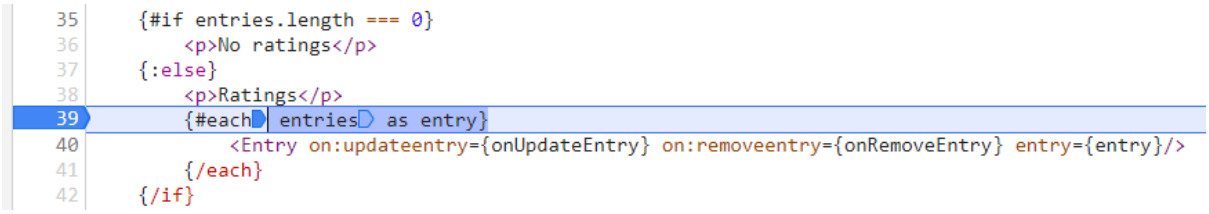

Abbildung 6: Debugging von Svelte-Code in Chrome

Hat man einen Breakpoint gesetzt, kann wie gewohnt mit dem Debugging fortgefahren werden.

3 Syntax

In diesem Kapitel soll nun ein grober Überblick über die Syntax von Svelte gegeben werden (7). Die vollständige Syntax ist deutlich umfangreicher und kann auf der offiziellen Webseite von Svelte im Browser online ausprobiert werden. Dort finden sich auch zahlreiche Tutorials, um Svelte zu lernen.

3.1 Aufbau des Quellcodes

Eine grundlegende Idee von Svelte ist die Einteilung des Quellcodes in GUI-Komponenten. Dies erleichtert die Programmierung für Single Page Applications. Jede GUI-Komponente hat ihren Quellcode dabei in einer eigenen Datei mit der Erweiterung „.svelte“. Der Quellcode in einer „.svelte“-Datei besteht aus drei Abschnitten: JavaScript, HTML und CSS. Dadurch ist die Komponentenlogik (JavaScript), die Struktur (HTML) und die Gestaltung der Komponente weiterhin getrennt. Dabei muss nicht jeder der drei Abschnitte zwingend vorhanden sein. Auch die Reihenfolge der Abschnitte spielt keine Rolle, da diese durch Tags eindeutig unterschieden werden können. Der JavaScript-Abschnitt wird eingeleitet durch ein <script>-Tag und der CSS-Abschnitt beginnt nach einem <style>-Tag. Der Rest ist HTML.

```
<script>
    // JavaScript-Code mit Svelte-Erweiterungen
</script>

<!-- Beliebige HTML-Elemente mit Svelte-Erweiterungen -->

<style>
    /* CSS-Code mit Svelte Erweiterungen */
</style>
```

In allen drei Abschnitten kann die gewöhnliche Sprachsyntax benutzt werden. Außerdem können die syntaktischen Erweiterungen, die Svelte bereitstellt, verwendet werden. Diese werden in den folgenden Unterkapiteln erläutert.

Ein weiterer Vorteil von Komponenten ist die Abkapselung des Quellcodes zu anderen Komponenten. Besonders bei CSS hat es den Vorteil, dass alle Styles nur innerhalb der Komponente wirken und somit die Gestaltung einer anderen Komponente nicht beeinflusst wird. Trotzdem ist es möglich CSS-Regeln zu erstellen, die global wirken. Diese werden in der Datei „global.css“ im Unterverzeichnis „public“ definiert.

3.2 Dynamisches HTML

Svelte erlaubt die Verwendung von JavaScript-Variablen im HTML-Code. Das DOM wird dabei mit den Werten der JavaScript-Variablen aktualisiert. Die Aktualisierung findet immer dann

statt, wenn der Variable ein neuer Wert zugewiesen wird. JavaScript-Variablen werden dabei im HTML-Code mit geschweiften Klammern markiert.

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

<h1>Hello {name}!</h1>
```

Dies Aktualisierung funktioniert jedoch nur über eine direkte Zuweisung mit dem „=“-Operator. Nur die Änderung der Elemente eines Arrays oder Objekts hat keine Auswirkungen.

Auch eine wiederholte Zuweisung eines Wertes innerhalb einer Schleife führt zu einer mehrfachen Aktualisierung des DOMs.

3.3 Kindkomponenten

Komponenten können in Svelte ineinander verschachtelt werden. Die äußere Komponente nennt man dann Vaterkomponente und die innere Komponente Kindkomponente. Dadurch lassen sich aus vorhandenen kleinen Komponenten wieder größere erstellen. Auch ist es übersichtlicher, eine große Komponente in mehrere Kindkomponenten zu zerlegen, um den Code besser zu strukturieren. Z.B. enthält ein Dialogfenster für eine Benutzeranmeldung normalerweise ein Textfeld zur Eingabe des Benutzernamens und ein Textfeld zur Eingabe des Passworts. Für die Passworteingabe könnte eine eigene Kindkomponente erzeugt werden, da in diese, weitere Funktionalitäten, wie z.B. das Anzeigen eines Icons zur Änderung der Sichtbarkeit des Passworts integriert werden können. Damit kann die Passwortkomponente auch in anderen Vaterkomponenten wie z.B. in Formularen zur Benutzerregistrierung verwendet werden.

Ziel der Kindkomponente ist es, unabhängig von der Vaterkomponente zu sein. Die Kindkomponente hat höchstens die Möglichkeit Nachrichten über Ereignisse an den Vater zu senden. Durch diese Unabhängigkeit muss in der Kindkomponente auch keine besondere Syntax verwendet werden.

Die Vaterkomponente kann jedoch eine Kindkomponente z.B. mit dem Namen `Child` im JavaScript-Abschnitt mit folgender Syntax einbinden:

```
<script>
   import Child from './Child.svelte';
</script>
```

Im HTML-Abschnitt kann die Kindkomponente dann verwendet werden:

```
<Child/>
```

3.4 Reaktivität

Reaktivität bedeutet in Svelte den Zustand der Anwendung mit dem DOM und andersherum zu synchronisieren. In Kapitel 3.2. hat man schon gesehen, dass Zuweisungen von Werten das DOM aktualisiert. Fügt man nun einem Element einen Listener hinzu, können auch Ereignisse von HTML-Elementen in JavaScript erfasst werden. In der angegebenen Handler-Funktion kann wieder durch eine Zuweisung das DOM aktualisiert werden.

```
<script>
   let count = 0;

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

<button on:click={handleClick}>
    {count} mal geklickt.
</button>
```

3.5 Props

Mit Hilfe von sogenannten Props kann eine Vaterkomponente einer Kindkomponente bestimmte Startwerte mit auf den Weg geben. Die Props haben einen Namen und einen Wert. In der Kindkomponente muss dann eine gleichnamige Variable mit dem Zusatz `export` definiert werden. Diese wird bei der Erzeugung der Kindkomponente automatisch mit dem zugehörigen Wert der „props“ initialisiert.

Vaterkomponente:

```
<script>
   import Child from './Child.svelte';
</script>
<Child weekday={'Friday'}/>
```

Kindkomponente (Child.svelte):

```
<script>
   export let weekday;
</script>
<p>Heute ist {weekday}</p>
```

3.6 Kontrollstrukturen

Die Steuerung des Programmflusses durch Kontrollstrukturen ist fester Bestandteil von JavaScript. Mit Svelte lässt sich auch der Aufbau des DOMs mit Bedingungen und Schleifen steuern. Das bedeutet, dass direkt im HTML-Code Kontrollstrukturen benutzt werden können. Zur Verfügung stehen Verzweigungen, Schleifen und auch die Zustände von Promises.

3.6.1 Verzweigungen

Mit Verzweigungen lassen sich in Abhängigkeit von JavaScript-Variablen unterschiedliche Elemente aus dem HTML-Code anzeigen. Durch eine Zuweisung im JavaScript wird das DOM aktualisiert. Mit geschweiften Klammern wird die Verzweigung im HTML-Code markiert. Möglich sind auch Verzweigungen mit alternativen Wegen (else if/else).

```
<script>
   let x = 7;
</script>

{#if x > 10}
   <p>{x} is greater than 10</p>
{:else if 5 > x}
   <p>{x} is less than 5</p>
{:else}
   <p>{x} is between 5 and 10</p>
{/if}
```

3.6.2 Schleifen

Schleifen bieten die Möglichkeit, im HTML-Code definierte Elemente wiederholt anzuzeigen. Im folgenden Beispiel wird ein in JavaScript definiertes Array mit Wochentagen mit der speziellen `each-as` Syntax durchlaufen. In jedem Schleifendurchlauf ist hier der aktuelle Array-Platz in der Variable `weekday` verfügbar. Damit können die Eigenschaften `id` und `name` ausgegeben werden.

```
<script>
   let weekdays = [
       {id: 1, name: "Monday"},
       {id: 2, name: "Tuesday"},
       {id: 3, name: "Wednesday"},
       {id: 4, name: "Thursday"},
       {id: 5, name: "Friday"},
       {id: 6, name: "Saturday"},
       {id: 7, name: "Sunday"}
   ];
</script>

{#each weekdays as weekday}
   <b>{weekday.id}. {weekday.name}</b><br>
{/each}
```

Der Code erzeugt folgende Ausgabe:

1. **Monday**
2. **Tuesday**
3. **Wednesday**
4. **Thursday**
5. **Friday**
6. **Saturday**
7. **Sunday**

3.6.3 Promises

Promises sind Zustände von asynchronen Funktionen. Folgende Zustände sind möglich:

- Funktion läuft noch
- Funktion erfolgreich beendet
- Funktion mit Fehler beendet

Svelte bietet die Möglichkeit in Abhängigkeit des Zustandes der Promise unterschiedliche Abschnitte des DOMs anzuzeigen.

Im folgenden Beispiel wird eine externe Resource `random-number` angefragt, um eine Zufallszahl zu generieren. Während die Anfrage läuft, wird im `await`-Block im HTML-Code eine Wartemeldung angezeigt. Bei Erfolg wird die Zahl in einem Paragraf angezeigt. Tritt stattdessen ein Fehler auf, wird eine rote Fehlermeldung angezeigt.

```
<script>
   let promise = getRandomNumber();

   async function getRandomNumber() {
       let response = await fetch(`tutorial/random-number`);
       let text = await response.text();
       if(!response.ok)
           throw new Error(text);
       return text;
   }
</script>

{#await promise}
   <p>warte…</p>
{:then number}
   <p>Die Zahl ist {number}</p>
{:catch error}
   <p style="color: red">{error.message}</p>
{/await}
```

3.7 Events

Mit Hilfe von Events kann eine Komponente einer anderen etwas mitteilen, ohne diese kennen zu müssen. „Nicht kennen“ heisst, dass keine Referenz zur anderen Komponente existiert. Durch diese Kommunikation werden die Komponenten unabhängiger voneinander. Dies ist für das Testen und die Wiederverwendbarkeit der Komponenten von Vorteil. In Svelte existieren zusätzlich zu den bereits in HTML und JavaScript vorhandenen DOM-Events auch Events von Kindkomponenten zur Vaterkomponente, sowie Events, die bei der Erzeugung und Zerstörung von Komponenten auftreten.

3.7.1 DOM-Events

Svelte erlaubt die Registrierung einer Handler-Funktion für DOM-Events direkt als Attribut im HTML-Tag mit der `on:event` Syntax. Im folgenden Beispiel wird `mousemove`, also eine Mausbewegung über das `div`-Element, registriert.

```
<script>
   let m = {x: 0, y: 0};

   function handleMousemove(event) {
      m.x = event.clientX;
      m.y = event.clientY;
   }
</script>

<div on:mousemove={handleMousemove}>
   Die Mausposition ist {m.x} x {m.y}
</div>
```

Durch die Zuweisungen in der Handler-Funktion wird das DOM automatisch aktualisiert und somit immer die aktuelle Mausposition angezeigt.

3.7.2 Component-Events

Komponenten können in Svelte wieder andere Komponenten verwenden. Dadurch entsteht eine Vater-Kind-Beziehung zwischen den Komponenten. Mit Hilfe von „Props“ (siehe Kapitel 3.5.) kann die Vaterkomponente der Kindkomponente Startwerte mitgeben. Da vom Kind zum Vater keine Referenz existiert, muss das Kind mit dem Vater über Events kommunizieren. Da diese Events zwischen zwei Komponenten ausgetauscht werden, nennt man sie „Component-Events“.

Der Programmierer muss dafür die Events in der Kindkomponente selbst erzeugen. Durch das Importieren und den Aufruf des `createEventDispatcher` wird eine `dispatch`-Funktion erzeugt. Diese kann verwendet werden, um ein Event an andere Komponenten zu senden. Man muss dafür als Parameter einen Namen und den Inhalt der Nachricht angeben. Im folgenden Beispiel wird in der Klick-Handlerfunktion des Buttons ein Event mit dem Namen

„message“ und der Variablen „text“ mit dem Wert „Hallo!“ erzeugt. Dadurch wird das Event bei Klick auf den Button an die Zuhörer gesendet.

Kindkomponente:

```
<script>
   import {createEventDispatcher} from 'svelte';
   let dispatch = createEventDispatcher();

   function handleClick() {
      dispatch('message', {
         text: 'Hallo!'
      });
   }
</script>

<button on:click={handleClick}>
   Klick, um Hallo zu sagen!
</button>
```

Die Vaterkomponente kann dem Event der Kindkomponente zuhören, indem sie auf die gleiche Art, wie bei den DOM-Events, mit der `on:event`-Syntax den Listener registriert und eine eigene Handlerfunktion für das „Component-Event“ angibt. Diese heißt hier `handleMessage` und gibt den Inhalt vom empfangenen Event in einer „Alert-Box“ aus.

Vaterkomponente

```
<script>
   import Child from './Child.svelte';

   function handleMessage(event) {
      alert(event.detail.text);
   }
</script>

<Child on:message={handleMessage}/>
```

3.7.3 Lifecycle-Events

„Lifecycle-Events“ sind Events, die während der Lebensdauer einer Komponente von Svelte selbst erzeugt werden. Es existieren in Svelte vier „Lifecycle-Events“.

- onMount
  Wird erzeugt, nachdem das DOM einer Komponente erstmalig geladen wurde. Zu diesem Zeitpunkt könnte man z.B. Inhalte vom Server laden.

- onDestroy
  Wird erzeugt, wenn eine Komponente aus dem DOM entfernt wird. Dies kann für Aufräumarbeiten verwendet werden, um Speicher und Reosurcen wieder freizugeben. Z.B. kann ein registrierter Listener sich wieder abmelden.
- beforeUpdate
  Wird direkt vor einer DOM-Aktualisierung erzeugt.
- afterUpdate
  Wird direkt nach einer DOM-Aktualisierung erzeugt.

Die folgende Grafik veranschaulicht die Abfolge der „Lifecycle-Events“.

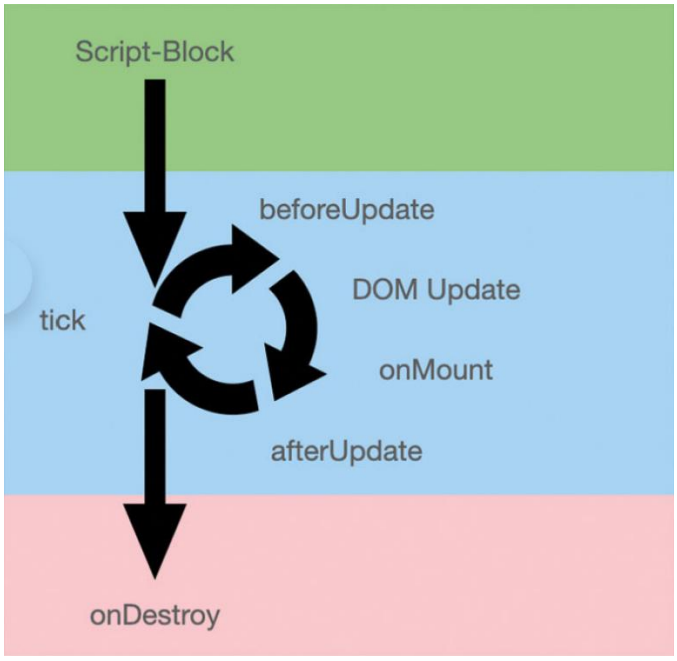

Abbildung 7: Lifecycle einer Svelte-Komponente

(8)

Das folgende Beispiel zeigt die Verwendung von „onDestroy“.

```
<script>
   import { onDestroy } from 'svelte';

   let seconds = 0;
   let timer = setInterval(() => seconds = seconds + 1, 1000);

   onDestroy(() => clearInterval(timer));
</script>
```

3.8 Binding

Oft ist es notwendig den Datenfluss zwischen Vater- und Kindkomponente oder zwischen JavaScript-Variablen und HTML-Elementen in beide Richtungen zu synchronisieren. Eine Vaterkomponente kann z.B. mit Hilfe der vorgestellten „props“ (Kapitel 3.5.) der Kindkomponente Daten übergeben, da diese der Vaterkomponente bekannt ist. Daten aus JavaScript-Variablen können an HTML-Elemente, z.B. als Attribute oder Eigenschaften übertragen werden. Um jedoch umgekehrt Datenänderungen aus HTML-Elementen in JavaScript zu erfassen oder Änderungen in einer Kindkomponente an die Vaterkomponente weiterzuleiten, sind normalerweise Events nötig.

Mit Hilfe der Binding-Syntax können beide Seiten sich gegenseitig synchronisieren, ohne Events programmieren zu müssen.

Im folgenden Beispiel wird die Variable `weekday` an die `value`-Eigenschaft des `input`-Elements gebunden. Ändert sich die `value` -Eigenschaft durch eine Benutzereingabe, ändert sich auch gleichzeitig die zugehörige Variable `weekday`. Durch die Variablenänderung wird intern eine Zuweisung durchgeführt. Diese sorgt wieder dafür, dass das DOM aktualisiert wird.

```
<script>
   let weekday = 'Montag';
</script>

<input bind:value={weekday}>
<p>Heute ist {weekday}!</p>
```

Das Binding funktioniert bei Svelte nicht nur für input-Elemente, sondern auch für eine Reihe weiterer Elemente und wie vorher genannt, auch für Variablen in Kindkomponenten, die mit Variablen in Vaterkomponenten synchronisiert werden sollen. Auch dazu sind keine „Component-Events“ notwendig.

3.9 Slots

Der Aufbau des DOMs kann innerhalb einer Kindkomponente mit Kontrollstrukturen gesteuert werden. Diese wurden in Kapitel 3.6. vorgestellt. Eine Vaterkomponente hat von außen normalerweise keinen Einfluss auf das DOM der Kindkomponente. Mit Hilfe von sogenannten Slots lässt sich jedoch im DOM der Kindkomponente ein Container erstellen, den der Vater mit eigenen Elementen füllen kann. Dies macht die Kindkomponente sehr flexibel.

Im folgenden Beispiel stellt die Kindkomponente zwei Slots mit dem Namen „name“ und „address“ bereit. Die Vaterkomponente füllt die Slots mit `span`-Elementen.

Kindkomponente “ContactCard.svelte”:

```
<article class="contact-card">
   <slot name="name"></slot>
   <slot name="address"></slot>
</article>
```

Vaterkomponente:

```
<ContactCard>
   <span slot="name">P. Sherman</span>
   <span slot="address">42 Wallaby Way<br>Sydney</span>
</ContactCard>
```

4 Der Svelte-Compiler

In diesem Kapitel soll die Funktionsweise des Svelte-Compilers grob an einem Beispiel dargestellt werden. Der Vorgang gliedert sich für jede Komponente in mehrere Schritte.

1. Erstellung eines abstrakten Syntaxbaums
2. Analyse des Baums, um Abhängigkeiten zu erfassen
3. Erzeugung von Javascript-Code
4. Ergänzung der CSS-Selektoren

Der Prozess kann in dieser Arbeit nicht bis ins Detail dargestellt werden. Es soll nur ein grober Überblick über den Ablauf erfolgen. (9), (10)

Der Compiler erzeugt eine Datei mit JavaScript-Code und eine Datei mit CSS-Code.

Als Beispiel dient eine kleine Svelte-Komponente mit einem Button, die durch Klick auf den Button den Titel des Buttons ändert.

```
<script>
   let count = 0;

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

<button on:click={handleClick}>
   Clicked {count} times
</button>

<style>
   button {
      color: red;
   }
</style>
```

4.1 Erstellung eines abstrakten Syntaxbaums

Im ersten Schritt erstellt der Compiler aus dem Quellcode der Komponente einen abstrakten Syntaxbaum. Dadurch kann der Compiler den Quellcode später leichter analysieren.

Ein abstrakter Syntaxbaum ist eine Darstellung der Struktur des Programmcodes in einem Baum. Zeichen wie z.B. Klammern oder Semikolons werden nicht in den Syntaxbaum übernommen, da diese nur zur Trennung von Abschnitten im Programmcode benötigt werden.

Ein Syntaxbaum zur Beispielkomponente könnte folgendermaßen aussehen.

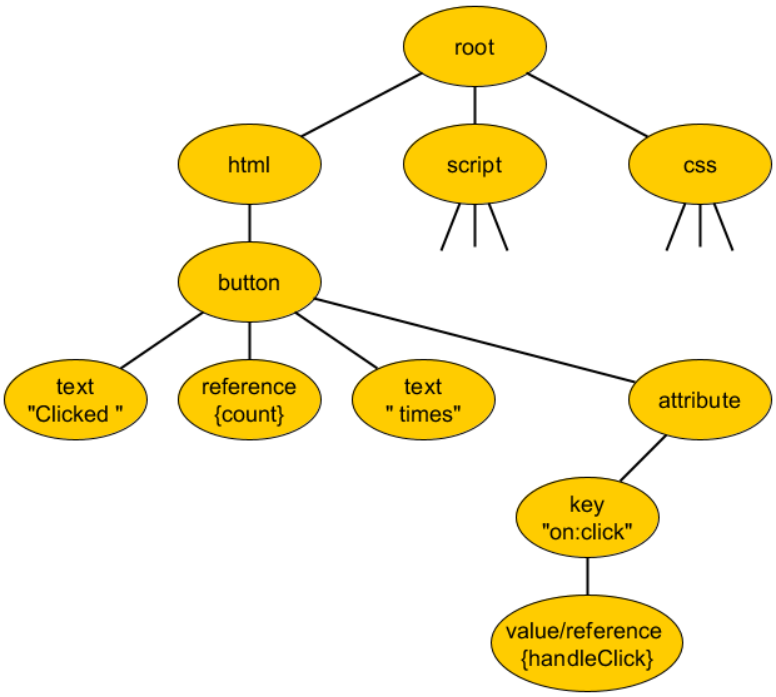

Abbildung 8: Abstrakter Syntaxbaum der Beispielkomponente

4.2 Analyse des Syntaxbaums

Mit Hilfe des Syntaxbaums kann der Compiler nun alle Referenzen zwischen dem HTML-Code und dem JavaScript-Code ermitteln. In unserem Beispiel gibt es im HTML-Code die zwei Referenzen `{count}` und `{handleClick}`. Diese werden in einer Liste mit dem Namen `ctx` gespeichert. Zu jeder Referenz werden außerdem zusätzliche Information über ihre Art gespeichert. Somit kann unterschieden werden, ob es sich bei der Referenz z.B. um eine Listener-Registrierung oder um eine Schleife handelt.

`ctx[0]`: Enthält `{count}`

`ctx[1]`: Enthält `{handleClick}`

4.3 Erzeugung des JavaScript-Code

Mit Hilfe dieser Liste lässt sich nun der kompilierte JavaScript-Code erzeugen.

Zunächst wird der kompilierte JavaScript-Code für den HTML-Teil erzeugt. Dazu werden für jede Komponente folgende drei Funktionen erzeugt:

- `create`: Diese Funktion erstellt die DOM-Elemente.

```
function create() {
    button = document.createElement("button");
    text1 = document.createTextNode("Clicked ");
    text2 = document.createTextNode(ctx[0]);
    text3 = document.createTextNode(" times");
}
```

Die Referenz `ctx[0]` wird als Inhalt für `text2` angegeben.

- `mount`: Diese Funktion fügt die Elemente ins DOM ein und registriert Listener.

```
function mount() {
    component.appendChild(button);
    button.appendChild(text1);
    button.appendChild(text2);
    button.appendChild(text3);

    button.addEventListener("clicked", ctx[1]);
}
```

Die Referenz `ctx[1]` wird als Handler-Funktion registriert.

Zur `mount`-Funktion wird auch eine entsprechende `unmount`-Funktion generiert, die die Komponente wieder aus dem DOM entfernt.

- `update`: Wird aufgerufen, wenn im JavaScript-Teil eine Zuweisung erfolgt. Dadurch wird das DOM aktualisiert.

```
function update() {
    text2.textContent = ctx[0];
}
```

Der Inhalt von `text2` wird mit der Referenz `ctx[0]` aktualisiert.

Der Svelte-Compiler erzeugt noch weitere Funktionen, diese können aber in dieser Arbeit nicht alle erläutert werden.

Anschließend wird der ursprüngliche JavaScript-Code übernommen und nach jeder Zuweisung, bei der das DOM aktualisiert werden muss, die `update`-Funktion aufgerufen.

```
let count = 0;

function handleClick() {
    count += 1;
    component.update();
}
```

4.4 Erzeugung des CSS-Code

Da der vom Entwickler geschriebene CSS-Code für jede Komponente lokal definiert ist, der Browser aber den gesamten CSS-Code global einbindet, können die Selektoren den entsprechenden DOM-Elementen nicht mehr eindeutig zugeordnet werden. Die Selektoren werden daher mit einer eindeutig unterscheidbaren Klasse erweitert.

```
<style>
    button.svelte-bf94eq {
        color:red
    }
</style>
```

5 Umsetzung einer Beispielanwendung

Ein Ziel dieses Projekts war es unter anderem mit Svelte eine Beispielanwendung umzusetzen, um die Vorteile von Svelte zu demonstrieren. Auch kann so Svelte leichter mit ähnlichen Frameworks verglichen werden. Im Folgenden soll die Realisierung beschrieben werden.

5.1 Die Rating-App

Bei der Beispielanwendung handelt es sich um eine App um Professoren zu bewerten. Die App ist als Single-Page-Application mit mehreren Svelte-Komponenten konstruiert. Die Anwendung soll es ermöglichen Professoren auf einer Skala von 0 bis 5 Sternen zu bewerten und anschließend in einer Liste anzuzeigen. Die Liste soll editierbar sein, d.h. der Name und die Bewertung sollen nachträglich geändert werden können. Außerdem soll ein Eintrag aus der Liste wieder entfernt werden können.

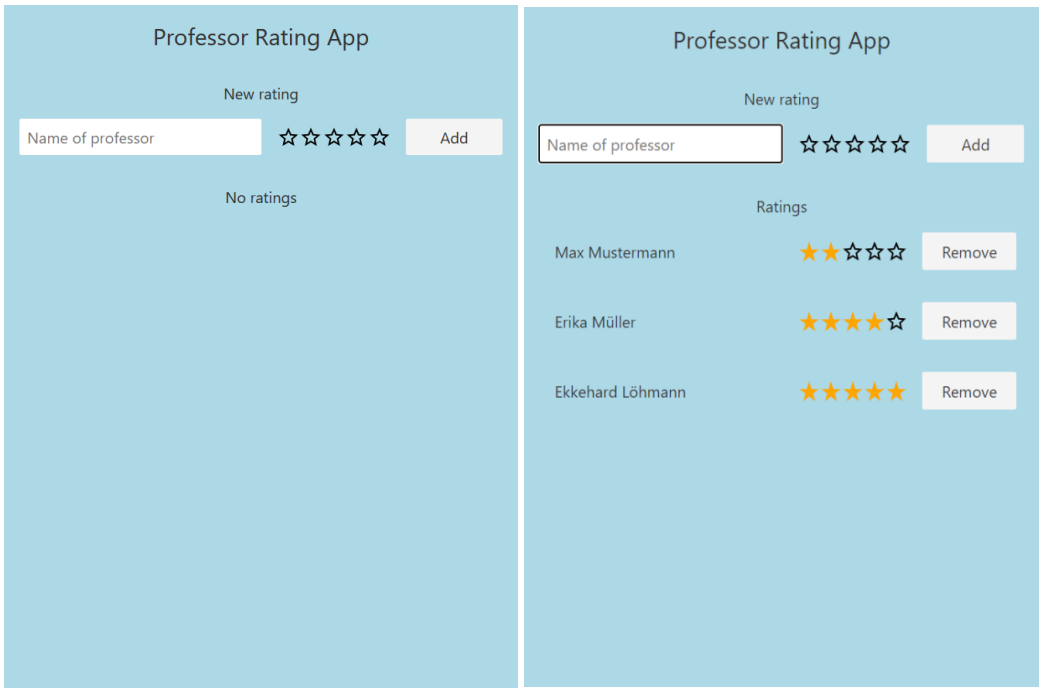

Abbildung 9: Screenshot der Rating-App

5.2 Komponenten

Die App besteht aus vier unterschiedlichen Komponenten. Die Rating-Komponente zeigt fünf Sterne nebeneinander an, auf die der Benutzer klicken kann. Dadurch entsteht eine Ratingzahl. Die AddEntry-Komponente erfasst den Namen und das Rating für einen neuen Eintrag in der Bewertungsliste. Die Entry-Komponente zeigt eine bestimmte Zeile in der Bewertungsliste an. Eine Zeile enthält den Namen des Professors, die vergebenen Sterne und einen „Entfernen“-Button, um den Eintrag aus der Liste zu entfernen. Außerdem enthält die

Entry-Komponente eine Logik, um zwischen der Anzeige und dem Editieren des Namens zu wechseln. Die App-Komponente ist die oberste Komponente und enthält die anderen Koponenten.

Die vier Komponenten stehen in folgender Hierarchie zueinander. Komponenten weiter oben in der Hierarchie erzeugen ihre Kindkomponenten.

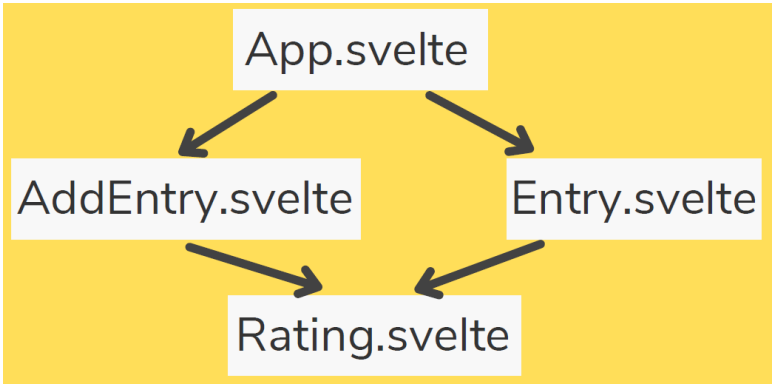

Abbildung 10: Komponentenhierarchie der Rating-App

Die folgenden Unterkapitel beschreiben die Komponenten detaillierter und erläutern die für Svelte besonderen Abschnitte.

5.2.1 Rating-Komponente

Die Rating-Komponente enthält die fünf Sterne als SVG-Grafiken. Es gibt volle und leere Sterne. Die Anzahl der vollen Sterne stellen genau das vom Benutzer gegebene Rating dar. Die Anzahl der übrigen leeren Sterne berechnet sich aus 5 – Rating. Um die Sterne im DOM darzustellen bieten sich zwei Schleifen an. Dank Svelte lassen sich die Schleifen direkt im HTML-Code integrieren.

```
{#each {length: rating} as _, i}
      <svg class="star-icon" data-id="{i+1}" on:click={setRating}>
         <use href="#icon-star-full"/>
      </svg>
   {/each}

   {#each {length: 5-rating} as _, i}
      <svg class="star-icon" data-id="{i+1+parseInt(rating)}"
           on:click={setRating}>
         <use href="#icon-star-empty"/>
      </svg>
   {/each}
```

Die `setRating`-Methode wird bei Klick auf einen Stern aufgerufen und ermittelt die `id` des geklickten Sterns, um sie der Variable `rating` in JavaScript zuzuweisen. Außerdem wird das so ermittelte Rating an die Vaterkomponente gesendet.

```
let setRating = (event) => {
    rating = event.currentTarget.getAttribute("data-id");
    //sendet an Vaterkomponente
    dispatch('rating', rating);
}
```

5.2.2 AddEntry-Komponente

Die AddEntry-Komponente enthält ein Textfeld zur Eingabe des Professors, die im vorherigen Kapitel beschriebene Rating-Komponente und den Button, um die Bewertung hinzuzufügen.

```
<input type="text" placeholder="Professor name" bind:value={entry.name}/>
<Rating on:rating={onRating} rating={entry.rating}/>
<button on:click={addEntry}>Add</button>
```

Das Textfeld und die Rating-Komponente wird mit einem Objekt `entry` mit den Eigenschaften `name` und `rating` synchronisiert. Beim Textfeld geschieht dies über ein „Binding“ und bei der Rating-Komponente über „Props“ und „Events“.

Der Event-Handler reagiert auf das „rating“-Event der Kindkomponente und setzt die `rating`-Eigenschaft des `entry`-Objekts.

```
let onRating = (event) => {
    entry.rating = event.detail;
};
```

Bei Klick auf den „Add“-Button wird der folgende Klick-Handler ausgeführt.

```
let addEntry = () => {
    let newEntry = {name: entry.name, rating: entry.rating}

    //sendet an Vaterkomponente
    dispatch('addentry', newEntry);
    entry.name = "";
    entry.rating = 0;
};
```

Dieser erstellt eine Kopie des `entry`-Objekts und versendet die Kopie über ein Event „addentry“ an die Vaterkomponente „App“. Darauf wird das `entry`-Objekt zur Eingabe einer weiteren Bewertung zurückgesetzt. Durch die Zuweisungen wird das DOM entsprechend aktualisiert.

5.2.3 App-Komponente

Die App-Komponente ist die oberste Komponente in der Hierarchie. Sie nutzt ein Array `entries` zur Speicherung aller Bewertungen. Es beschreibt den aktuellen Zustand der App.

Außerdem enthält sie als Kindkomponenten die AddEntry-Komponente zum Erstellen einer neuen Bewertung, sowie eine Liste mit Entry-Komponenten zur Anzeige der bereits vorhandenen Bewertungen.

Eine Verzweigung ermittelt im HTML-Code, ob das `entries`-Array leer ist und ein entsprechender Hinweis angezeigt werden soll.

```
<p>New rating</p>
<AddEntry on:addentry={onAddEntry}/>

{#if entries.length === 0}
   <p>No ratings</p>
{:else}
   <p>Ratings</p>
   {#each entries as entry}
      <Entry on:removeentry={onRemoveEntry} entry={entry}/>
   {/each}
{/if}
```

Die Komponente hört auf Events der beiden Kindkomponenten. Der bei einem Hinzufügen einer Bewertung aufgerufene Event-Handler erweitert das Array `entries` um die neue Bewertung.

```
let onAddEntry = (event) => {
   entries = [...entries, event.detail];
};
```

Der für das Entfernen zuständige Event-Handler filtert die Bewertung aus dem Array heraus.

```
let onRemoveEntry = (event) => {
   entries = entries.filter(entry => entry !== event.detail);
};
```

Hier erkennt man wieder die Vorteile von Svelte. Durch Zuweisungen wird das DOM automatisch aktualisiert. Allerdings muss darauf geachtet werden, dass wirklich eine Zuweisung stattfindet. Würde man z.B. ein Element mit der `Array.push` Methode zu einem Array hinzufügen, müsste man das erweiterte Array wieder sich selbst zuweisen, damit Svelte das DOM aktualisiert.

5.2.4 Entry-Komponente

Die Entry-Komponente realisiert einen Eintrag in der Bewertungsliste. Sie besteht ähnlich wie die NewEntry-Komponente aus einem Bereich für den Professornamen, der Rating-Komponente mit den fünf Sternen zur Änderung der Bewertung und einem Button, um die Bewertung aus der Liste zu entfernen. Der Bereich für den Professornamen enthält allerdings eine zusätzliche Logik, um bei Klick auf den Namen ein Textfeld zum Editieren anzuzeigen. Bei erneutem Klick außerhalb des Textfeldes („blur“), soll das Textfeld wieder deaktiviert werden. Hierzu wird eine Variable `edit` verwendet, die je nach Zustand zwischen `true` und `false` wechselt. Außerdem bekommt die Komponente von ihrem Vater ein `entry`-Objekt übergeben („props“), das den Namen und die Bewertung des Eintrags enthält.

Im HTML-Code wird eine Verzweigung verwendet, um zu entscheiden, ob der Text als `input`-Element editierbar ist oder als `div`-Element nur ausgegeben wird.

```
{#if edit}
   <input type="text" placeholder="Name of professor"
      bind:this={inputElement} bind:value={entry.name}
      on:blur={toggleEdit}/>
{:else}
   <div on:click={toggleEdit}>{entry.name}</div>
{/if}

<Rating on:rating={onRating} rating={entry.rating}/>
<button on:click={removeEntry}>Remove</button>
```

Für beide Elemente wird der Event-Handler `toggleEdit` aufgerufen. Dieser ändert die Editierbarkeit und fokussiert das Textfeld, nachdem es aktiviert wird. Dabei muss die Fokussierung solange hinausgezögert werden, bis das DOM aktualisiert ist und das Textfeld auch angezeigt wird. Diese Verzögerung geschieht durch Aufruf der `tick`-Methode, die eine Promise zurückgibt. Ist das DOM aktualisiert, wird die Promise aufgelöst und die `focus`-Methode wird aufgerufen.

```
let toggleEdit = async () => {
   edit = !edit;

   if(edit) {
      await tick();
      inputElement.focus();
   }
};
```

Das `input`-Element erhält außerdem ein Binding an die Eigenschaft `entry.name`, um sich mit dem `entry`-Objekt zu synchronisieren.

Die Rating-Komponente wird als Kindkomponente erzeugt und bekommt die Eigenschaft `entry.rating` übergeben. Außerdem hört sie, wie die NewEntry-Komponente auch, auf „rating“-Events, um die Bewertung zu aktualisieren.

Der „Remove“-Button besitzt einen Klick-Handler, der das Entfernen an die Vaterkomponente als Ereignis meldet.

```
let removeEntry = () => {
   //sendet an Vaterkomponente
   dispatch('removeentry', entry);
};
```

5.3 Styles

Ein Vorteil von Svelte ist die Kapselung der CSS-Styles innerhalb der Komponenten. D.h. die Wirkung der Styles ist nur auf die jeweilige Komponente begrenzt. Damit stören die Styles nicht die Gestaltung von anderen Komponenten. CSS-Klassennamen können somit doppelt vergeben werden.

Die Gestaltung der Beispiel-App ist einfach gehalten. Die App-Komponente besitzt ein Flex-Layout und zeigt ihre Inhalte untereinander an. Die AddEntry- und die Entry-Komponente besitzen ein Grid-Layout um drei Spalten erzeugen. Die innere Rating-Komponente besitzt ein Flex-Layout, um die fünf Sterne nebeneinander anzuzeigen. Die gesamte App hat eine feste Breite.

5.4 Kommunikation zwischen Frontend und Backend

Die bisher beschriebene Beispielanwendung wurde durch eine zweite Version ergänzt, um die Liste der Bewertungen auf einem entfernten Server in einer Datenbank zu speichern. Zum Testen wurde das von Prof. Dr. Marius Hofmeister auf seiner Webseite „web-forward.de“ unter dem Beitrag „Tutorial: Dive into Vue+Node+REST“ veröffentlichte Backend verwendet. Dieses stellt dem Client eine REST-API zur Verfügung und speichert die Bewertungsliste in einer JSON-Datei. (11)

Die REST-API hat vier Endpoints:

Endpoint	**Beschreibung**
GET /profs	Liste der Bewertungen holen
POST /prof	Neue Bewertung in die Liste einfügen
PUT /prof/:id	Bewertung mit bestimmter id verändern
DELETE /prof/:id	Bewertung mit bestimmter id löschen

Tabelle 4: Beschreibung der Endpoints

Im Frontend hinzugekommen ist eine neue Klasse „RatingApi“ in der Datei „api.js“. Die Klasse bietet vier Hilfsmethoden, um die API-Operationen per Ajax durchzuführen.

GET /profs wurde z.B. folgendermaßen implementiert:

```
getEntries() {
    return fetch(this.url + "/profs")
    .then(response => response.json());
}
```

Die Hilfsmethoden werden in der Hauptkomponente App.svelte aufgerufen, da diese den Zugriff auf den gesamten Zustand der App besitzt. Zustände, die in den Kindkomponenten vorhanden sind, werden entweder je nach Richtung über Props, Events oder Bindings synchronisiert.

Um z.B. eine Bewertung hinzuzufügen, wird in der App-Komponente nicht mehr direkt die Liste mit den Bewertungen geändert, sondern die Hilfsmethode zum Hinzufügen der Bewertung auf dem Server aufgerufen.

```
let onAddEntry = (event) => {
    api.addEntry(event.detail).then(() => update());
};
```

Anschließend wird die Liste vom Server neu geladen.

```
let update = () => {
    api.getEntries().then((serverentries) => entries = serverentries);
};
```

Durch die Zuweisung von `serverentries` zu `entries` wird das DOM aktualisiert und die neue Liste angezeigt.

5.5 Installation

Die Rating-App lässt sich einfach installieren durch Herunterladen eines der beiden Repositories. (12), (13)

Anschließend müssen folgende Befehle im Terminal ausgeführt werden:

```
npm install
```

```
npm run dev
```

Nun ist es möglich über die Adresse localhost:5000 mit dem Browser darauf zuzugreifen

6 Fazit

Insgesamt hat mir die Arbeit mit Svelte sehr viel Spaß gemacht. Dies möchte ich betonen, da ich vor Beginn der Arbeit noch nie etwas von Svelte gehört habe. Ich habe vorher Frontends mit Webkomponenten programmiert und habe bis dahin noch kein Framework ausprobiert. Die Syntax von Svelte hat mich vor allem wegen ihrer Übersichtlichkeit überzeugt und dass man im Vergleich zu Webkomponenten deutlich weniger Quellcode schreiben muss. Dies liegt vor allem an dem Databinding. Auch der neue Weg, den Quellcode zu kompilieren und somit darauf zu verzichten eine umfangreiche zusätzliche Bibliothek an den Browser auszuliefern macht für mich Sinn. Daraus entsteht auch die gute Performance von Svelte. Leider habe ich keine Erfahrung mit anderen Frontend-Frameworks und konnte deshalb Svelte nicht mit z.B. Vue vergleichen. Kleinere Projekte würde ich sofort mit Svelte programmieren. Für ein großes Projekt würde ich mir noch weitere Frameworks anschauen. Auch möchte ich betonen, dass ich von Svelte nur die wichtigsten Möglichkeiten genauer vorgestellt habe. Weitere interessante Features sind z.B. Stores (Behälter zum Speichern von Zuständen, die von mehreren Komponenten benutzt werden) und die einfache Definition von Animationen.

Die Einarbeitung ist mir sehr leichtgefallen, vor allem da es auf der Webseite von Svelte ein umfangreiches Tutorial und viele Beispiele gibt, die man sofort online ausprobieren kann, ohne erst ein Projekt erstellen zu müssen. Nur bei wenigen Fragen konnte mir die offizielle Seite nicht weiterhelfen, dafür gab es aber schon Antworten bei der bekannten Frageseite „stackoverflow“.

Die Entwickler von Svelte arbeiten auch noch an interessanten Erweiterungen und weiteren Projekten.

Zukünftige Versionen sollen z.B. eine Unterstützung für TypeScript enthalten.

Außerdem arbeiten die Entwickler von Svelte auch an „Sapper“, einem serverseitigen Framework für node.js. Mit diesem ist es möglich, Svelte-Komponenten mit node.js zu kompilieren und in HTML-Code umzuwandeln. Dieser generierte HTML-Code kann dann direkt an den Browser ausgeliefert werden. Ein Vorteil dabei ist die bessere Optimierung für Suchmaschinen, da der zum Client gesendete HTML-Code direkt von einem Bot analysiert werden kann. (14)

Ein weiteres interessantes Projekt neben Svelte ist „Svelte Native“ mit dem sich mit der Svelte-Syntax direkt Android- oder iOS-Apps erstellen lassen. (15)

7 Quellenverzeichnis

1. Svelte. *https://svelte.dev/blog/virtual-dom-is-pure-overhead.* [Online]

2. heise.de. *https://www.heise.de/hintergrund/Wieso-Svelte-Ein-Einstieg-in-das-JavaScript-Framework-4686037.html.* [Online]

3. icickle Technologies. *https://www.icicletech.com/blog/angular-react-vue-svelte-comparison.* [Online]

4. JavaScript in Plain English. *https://javascript.plainenglish.io/javascript-frameworks-performance-comparison-2020-cd881ac21fce.* [Online]

5. Svelte. *https://svelte.dev/blog/write-less-code.* [Online]

6. Svelte. *https://svelte.dev/blog/svelte-3-rethinking-reactivity.* [Online]

7. Svelte. *https://svelte.dev/tutorial/.* [Online]

8. Entwickler Magazin - Svelte unter der Haube. *https://kiosk-entwickler.de.* [Online]

9. dev.to. *https://dev.to/joshnuss/svelte-compiler-under-the-hood-4j20.* [Online]

10. Tan Li Hau. *https://lihautan.com/the-svelte-compiler-handbook/.* [Online]

11. Web-Forward. *https://web-forward.de/2020/06/tutorial-dive-into-vue-node-rest/.* [Online]

12. Github. *https://github.com/Bagi5512/svelte-rating-app.* [Online]

13. Github. *https://github.com/Bagi5512/svelte-rating-app-with-backend.* [Online]

14. Sapper. *https://sapper.svelte.dev/.* [Online]

15. Svelte Native. *https://svelte-native.technology/.* [Online]