WEBENTWICKLUNG

# Nur das Nötigste

Mit Svelte hält ein komplett neues Konzept Einzug in die Frontend-Entwicklung: Compiler Driven Development.

Beim Thema Webentwicklung im Frontend hat sich in den letzten Jahren nicht besonders viel getan. Während vor allem zwischen 2012 und 2015 Frameworks, Bibliotheken und Werkzeuge wie Unterwäsche ausgetauscht wurden, ist seit 2017 eine gewisse Stagnation festzustellen. Das ist positiv, denn niemand möchte stärker mit Framework-Migrationen als der eigentlichen Webentwicklung beschäftigt sein.

Während die Stagnation die Tür für Nischenprojekte weit öffnete, gibt es mit Svelte nun eine interessante Alternative zu den etablierten Frameworks. Svelte [1] versteht sich – im Gegensatz zu den etablierten Frameworks – als „Compiler Driven Framework". Das bedeutet, dass man mit Svelte eine Webapplikation erstellen kann, die, sobald sie fertiggestellt ist, nicht mehr von Svelte abhängig ist. Svelte ist daher einzig und allein ein Compiler, der den erforderlichen Code ausgibt, um die Applikation lauffähig zu machen.

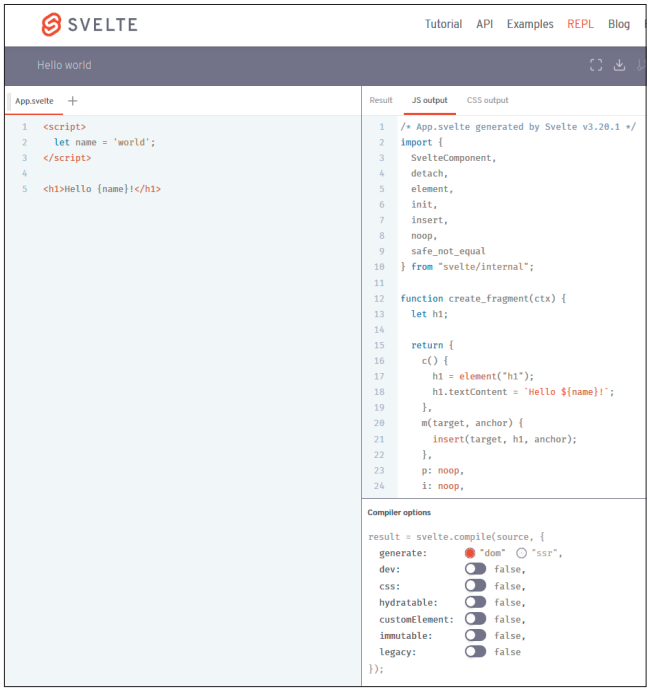
## Geschichte

Svelte ist nicht erst gestern erschienen. Wie die meisten erfolgreichen JavaScript-Projekte bewegte sich die Entwicklung in den ersten beiden Jahren eher unter dem Radar. Die erste Version wurde Ende 2016 von Rich Harris veröffentlicht. Als Mitarbeiter der „New York Times" ist er in der JavaScript-Community nicht unbekannt. So lieferte er unter anderem Beiträge in Projekten wie Rollup und Ractive.js. Seine Erfahrungen mit JavaScript-Compilern durch das Rollup-Projekt waren für die Entwicklung von Svelte durchaus relevant.

Die Idee von Svelte ist ganz einfach erklärt. Am Anfang steht die Frage: Wie viel Code benötigt eine Hallo-Welt-Applikation bei Verwendung eines Frontend-Frameworks? Für Applikationen, die Frameworks wie React, Angular oder Vue einsetzen, spielt die Größe des Problems kaum eine Rolle – es wird im Regelfall derselbe Ballast zum eigentlichen Code hinzugefügt. Durch Methoden wie „Tree Shaking" können zwar durchaus entsprechende Optimierungen stattfinden, das Ergebnis wird dadurch aber nur geringfügig beeinflusst. Besonders bei großen Frameworks wie Angular kann das schnell problematisch werden. Svelte geht hier einen anderen Weg und fügt nur den Code ein, der wirklich gebraucht wird.

## Compiler Driven

Im Prinzip steht hinter Svelte die Überlegung, dass es zwei Wege gibt, eine Abstraktion zu erstellen: Runtime und Compiletime. Sicherlich ist die Unterscheidung nicht immer ganz klar. Dieser Fall bezieht sich auf ein Beispiel, das Listing 1 zeigt. Während der Originalcode mit dem Schlüsselwort *await* ausgezeichnet ist, können Sie eine Abstraktion schreiben beziehungsweise eine Abstraktion von einer Library verwenden, um dieselbe Funktionalität auch ohne *await* zu erhalten.




**Die Svelte-Spielwiese** [3] bietet mehrere Optionen (Bild 1)

Eine andere Alternative ist es, einen intelligenten Compiler zu verwenden. Dieses System ist in der Lage, die Befehle so zu verändern, dass diese in einer gewünschten Zielarchitektur ausgeführt werden können. Der Unterschied zur Runtime-Abstraktion besteht darin, dass sich der Compiler um die Abstraktion kümmert.

Im Fall von Svelte bedeutet dies: Anstatt ein Framework einzusetzen, wird ein Compiler verwendet. Der Vorteil: Der Code muss nicht zur Laufzeit durch die verschiedenen Abstraktionsschichten laufen, sondern der Compiler kann die notwendigen Instruktionen bereits vorab einfügen.

## Los geht's

Um Svelte nutzen zu können ist also die Unterstützung durch den Svelte-Compiler erforderlich. Der Compiler übersetzt (transpiliert) **.svelte*-Dateien in JavaScript. Um den Svelte-Compiler zusätzlich mit normalen JavaScript-Modulen nutzen zu können, sollte auch noch ein Bundler installiert werden. Für die Beispiele in diesem Artikel kommt der Bund- ▶

**Listing 1: Run- und Compiletime-Abstraktionen**

```
// Originalcode
await doOne();
await doTwo();
await doThree();

// Notwendige Änderungen -> Run-Time Abstraktion
function sequence(callbacks) {
  const next = callbacks.shift();
  if (next) {
    return Promise.resolve(next()).then(
      () => sequence(callbacks));
  }
}
sequence([doOne, doTwo, doThree]);

// Ohne Änderung -> Compiletime Abstraktion
doOne().then(doTwo).then(doThree);
```

ler Parcel [2] zum Einsatz. Ein neues Svelte-Projekt mit Parcel als Bundler kann durch die folgenden beiden Kommandozeilenbefehle aufgesetzt werden:

```
mkdir my-svelte-project && cd my-svelte-project
npm init -y && npm i parcel-bundler
  parcel-plugin-svelte svelte --save-dev
```

Die Befehle legen einen neuen Ordner an (*my-svelte-project*), der mit der Datei *package.json* ausgestattet wird, und auch die Abhängigkeiten von Parcel beziehungsweise Svelte sind damit schon installiert. Der Code für eine einfache Hallo-Welt-Applikation mit Svelte sieht so aus (*App.svelte*):

```
<script>
  let name = 'world';
</script>
<h1>Hello {name}!</h1>
```

Eine Svelte-Datei besteht aus drei Blöcken: dem *script*-Block, der den JavaScript-Code enthält, dem *style*-Block, der das Styling der Komponente festlegt, und dem übrigen Code, der als Komponentenblock für die DOM-Repräsentation interpretiert wird. Jeder Block ist dabei optional, sogar eine leere Datei ist eine gültige Svelte-Datei.

Um die Verwandlung des obigen Hallo-Welt-Codes zu betrachten, können Sie entweder den Svelte-Compiler direkt verwenden oder die im Internet verfügbare Read-Evaluate-Print-Loop (REPL) nutzen [3]. Dabei gibt es verschiedene Optionen, wie Bild 1 zeigt.

Das Beispiel einer Hallo-Welt-Applikation ist sehr klein und schlägt nahezu jedes verfügbare Framework in puncto Größe. Trotzdem ist der in Listing 2 gezeigte Code immer noch relativ lang.

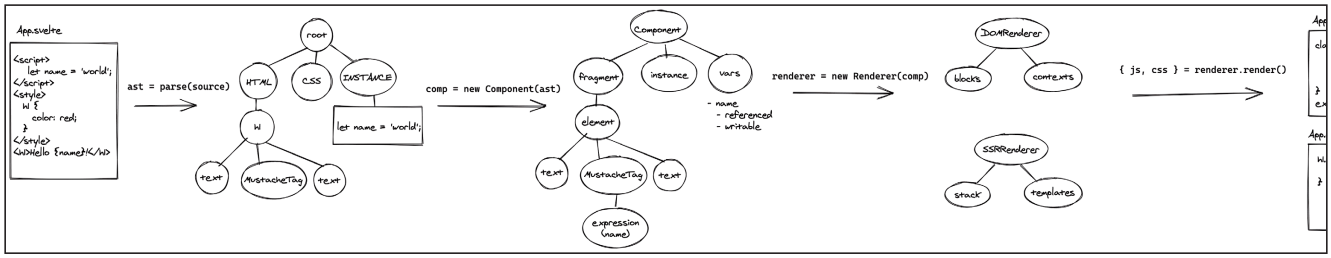
Sehen Sie sich den Code genauer an, fällt sofort die Verwendung des Moduls *svelte/internal* auf. Wird hier letztlich also doch mit einem Framework gearbeitet? Sicherlich ist es sinnvoll, die einzelnen Hilfsfunktionen direkt von einem gemeinsamen Modul zu verwenden. Auf diese Art und Weise wird jeder Helfer nur einmal importiert.

Der im Beispiel gezeigte Mechanismus über eine Hilfsbibliothek ist nicht neu. Bei .NET werden viele Helfer auch erst durch Base Class Libraries (BCL) bereitgestellt. Ein Beispiel für die Verwendung der BCL ist die Integration der Unterstützung von *async/await*. Selbst bei TypeScript kann man jeden

**Listing 2: Hallo Welt (App.js)**

```
/* App.svelte generated by Svelte v3.20.1 */
import {
  SvelteComponent, detach,
  element, init, insert,
  noop, safe_not_equal
} from "svelte/internal";

function create_fragment(ctx) {
  let h1;
  return {
    c() {
      h1 = element("h1");
      h1.textContent = `Hello ${name}!`;
    },
    m(target, anchor) {
      insert(target, h1, anchor);
    },
    p: noop,
    i: noop,
    o: noop,
    d(detaching) {
      if (detaching) detach(h1);
    }
  };
}
let name = "world";
class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, null, create_fragment,
      safe_not_equal, {});
  }
}
export default App;
```




**Die vier Phasen** des Svelte-Compilers (Bild 2)

Helfer über eine Referenz zum Paket *tslib* einbinden. Die Beispielkomponente ist alleine noch nicht lauffähig. Zuvor müssen Sie die exportierte Klasse noch instanzieren und an ein bereits bestehendes DOM-Element anbinden. Dies ist komplett ohne eine Runtime möglich – dem Compiler-Driven-Ansatz sei Dank.

Die nachfolgenden Zeilen zeigen die Anbindung der erstellten App-Komponente im *body*-Element. Im Code wird die Datei *App.svelte* direkt referenziert. Ein Bundler mit Anbindung an den Svelte-Compiler kann diese Referenz direkt über die transpilierte *App.js* auflösen:

```
// Anbindung von Svelte (index.js) //
import App from './App.svelte';
const app = new App({
  target: document.body,
  props: {
    name: 'world'
  }
});
```

Durch die implizite Komponentendeklaration eignet sich Svelte besonders für Cross-Framework-Anwendungen. Beispielsweise ist eine Svelte-Applikation in Microfrontend-Anwendungen im Regelfall direkt und ohne aufwendige Integration einsatzbereit.

Zur komfortablen Entwicklung der erstellten Applikation empfiehlt sich eine Anpassung der Datei *package.json*. Wie in Listing 3 gezeigt, sollten entsprechende Skripte zum effizienten Debugging (*start*) beziehungsweise einem produktionsreifen Bauvorgang (*build*) eingefügt werden.

In der Tat werden hier keinerlei Laufzeitabhängigkeiten aufgeführt – die Abhängigkeit von den genannten Paketen besteht somit nur während der Entwicklung. Zu diesem Zeitpunkt ist erste Beispielapplikation lauffähig und kann benutzt werden. Dafür müssen Sie nur die Datei *dist/index.js* in einer bestehenden HTML-Datei referenzieren.

## Der Svelte-Compiler

Um Svelte im Detail zu verstehen, muss man früher oder später einen genauen Blick auf den Svelte-Compiler werfen. Zum Glück gibt es neben dem frei verfügbaren Quellcode auch noch bereits aufbereitetes Informationsmaterial [4]. Insgesamt besteht der Svelte-Compiler aus vier Phasen – zwei Phasen mehr als bei einem klassischen Compiler. Die vier Kompilierungsstufen von Svelte sind:

- Identifizieren des abstrakten Syntaxbaums (AST) mithilfe der drei oben beschriebenen Blöcke.
- Erstellen des Komponentenmodells unter Berücksichtigung des AST.
- Anlegen des entsprechenden Renderers (client- oder serverseitig).
- Rendering mit Rückgabe von JavaScript, CSS und weiteren Dateien (Sourcemaps)

Dabei ist die Ausgabe für moderne Browser optimiert. Folgerichtig ist eine Unterstützung des alten Internet Explorer 11 (IE 11) nur über zusätzliche Tools erreichbar. Bild 2 illustriert die Phasen nochmals.

Zusätzlich zu den vier Phasen können noch weitere Stufen wie beispielsweise Tree Shaking oder weitere Optimierungen hinzugefügt werden. Diese zusätzlichen Stufen wür- ▶

**Listing 3: Skripte zur Entwicklung mit Svelte über Parcel (package.json)**

```
{
  "name": "my-svelte-project",
  "version": "1.0.0",
  "description": "A simple Svelte example.",
  "main": "dist/index.js",
  "scripts": {
    "start": "parcel src/index.js",
    "build": "parcel build src/index.js"
  },
  "keywords": ["svelte", "demo", "tutorial"],
  "author": "Florian Rappl",
  "license": "ISC",
  "devDependencies": {
    "parcel-bundler": "^1.12.4",
    "parcel-plugin-svelte": "^4.0.6",
    "svelte": "^3.20.1"
  }
}
```

den allerdings nicht vom Compiler kommen, sondern eher von einem Bundler wie Parcel.

Die Prozessierung unter anderem von CSS führt dazu, dass Svelte durchaus weiß, welche CSS-Styles tatsächlich verwendet werden. Dadurch können nicht genutzte CSS-Regeln effizient ausgelassen werden. Dieser Vorteil besteht auch für die geschriebenen HTML-Deklarationen.

Svelte verwendet zur Umsetzung der Interaktivität kein virtuelles DOM (kurz VDOM). Dies wäre ohne Runtime-Unterstützung gar nicht beziehungsweise nur sehr schwer möglich. Stattdessen werden die impliziten Zustandsänderungen in konsistente imperative Anweisungen übersetzt. Dadurch ist keinerlei VDOM-Abgleich nötig. Um die Zustandsvariablen effizient in HTML widerzuspiegeln, werden Mustache-Anweisungen in den HTML-Block eingefügt. Der Svelte-Compiler kann diese dann entsprechend behandeln und durch JavaScript-Code lauffähig machen.

In Listing 1 wurde bereits eine Mustache-Anweisung eingesetzt: Durch die Verwendung von geschweiften Klammern wird ein einfacher JavaScript-Ausdruck eingefügt. Daneben gibt es noch Spezialanweisungen. So wird über *{#if}* ein konditionaler Block eingefügt. Für Schleifen existiert *{#each}*, und über *{#await}* kann man sehr einfach das Ergebnis einer asynchronen Auswertung darstellen. Hier ein Beispiel für den Einsatz konditionaler Blöcke:

**Listing 4: Lebenszyklus einer Svelte-Komponente**

```
<script>
  import { onMount, beforeUpdate, afterUpdate,
    onDestroy } from 'svelte';
  let name = 'world';

  setTimeout(() => {
    name = 'user';
  }, 2000);

  console.log('initializing');

  onMount(() => {
    console.log('mounted');
  });

  beforeUpdate(() => {
    console.log('before update');
  });

  afterUpdate(() => {
    console.log('after update');
  });

  onDestroy(() => {
    console.log('destroyed');
  });
</script>

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

**Listing 5: Ein einfaches Dokument (index.html)**

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content=
      "width=device-width, initial-scale=1.0">
    <title>Svelte Twitter Timeline</title>
    <link rel="stylesheet" href="global.css">
  </head>
  <body>
    <div id="app"></div>
    <script src="index.js"></script>
  </body>
</html>
```

```
<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}
{/if}
```

Die Verwendung von *{#each}* ist sehr ähnlich. Insgesamt ist die Art und Weise der Deklaration möglicherweise nicht ganz so elegant beziehungsweise nicht ganz so konsistent wie in React, allerdings sehr viel geradliniger und leichtgewichtiger als in den meisten anderen Frameworks. Svelte macht hier vieles richtig.

Neben allerlei Umstellungen nutzt der Compiler, wie bereits gesehen, auch noch interne Hilfsfunktionen. Es gibt vier Kategorien von Helfern:

- DOM-Manipulatoren wie *append, insert* und *detach*.
- Zustandshelfer wie *schedule_update* und *flush*.
- Werkzeuge für den Lebenszyklus von Komponenten, zum Beispiel *onMount* und *beforeUpdate*.
- Helfer für Animationen und Styling, beispielsweise *create_animation*.

Alle Helfer werden in der ein oder anderen Situation zum Kompilieren von Svelte-Komponenten benötigt.

## Komponenteninteraktion

Zu einer vollständigen Diskussion über Svelte – oder jedes andere Frontend-Framework – gehört auch ein Blick in das Komponentenmodell. Schon erwähnt wurde, dass eine Svelte-Datei einer Komponente entspricht. Diese Datei beinhaltet drei Blöcke, um eine Komponente vollständig bezüglich Styling, Verhalten und DOM-Repräsentation darstellen zu können. Innerhalb des *script*-Blocks können auch Callbacks definiert werden, die beim Abarbeiten des Lebenszyklus der Komponente verwendet werden. Der Lebenszyklus einer Svelte-Komponente durchläuft folgende Stadien:

- Die Komponente wird initialisiert.
- Die Komponente wurde im DOM erstellt beziehungsweise verankert (*onMount*).
- Die Komponente wird aktualisiert (*beforeUpdate*).
- Die Komponente wurde aktualisiert (*afterUpdate*).
- Die Komponente wurde aus dem Dokument Object Model (DOM) entfernt (*onDestroy*).

Der Zugriff auf die einzelnen Bereiche ist durch eine Definition von Callbacks (Hooks) möglich. Die Hooks können Sie über Funktionen setzen, die Svelte zur Verfügung stellt. Eine Ausnahme bildet das Initialisieren der Komponente. Hier kann man direkt im *script*-Block arbeiten.

In Listing 4 wird das Eingangsbeispiel leicht modifiziert, um die verschiedenen Lebenszyklusmethoden in Aktion zu sehen. Dabei ist der *script*-Block als eine Art Konstruktor-Aufruf zu betrachten. Tatsächlich verändert der Svelte-Compiler diesen Block nahezu vollständig in eine *init*-Funktion, die während des Konstruktor-Ablaufs aufgerufen wird.

Im gezeigten Beispiel folgt die Ausgabe in der Konsole genau den Deklarationen im Code. Dabei wird die Funktion *setTimeout* genutzt, um das Setzen der Variable *name* zu verzögern. Interessanterweise erkennt Svelte diese Modifikation selbst und benötigt keine Runtime-Hilfe oder den Aufruf einer Spezialfunktion wie *setState*.

Bei manchen Modifikationen erkennt Svelte allerdings nicht automatisch, dass dies eine weitere Änderung nach sich zieht. Es muss eine Spezialsyntax genutzt werden, die in der Tat gültiges JavaScript ist: die Definition eines Labels. Als Label wird einfach ein Dollarzeichen (*$*) verwendet. Dadurch wird im vorherigen Code beispielsweise eine Zeile wie *$: document.title = name;* auch dann ausgeführt, wenn sich die Variable *name* ändert.

Um Parameter an Komponenten weiterreichen zu können, müssen Sie diese Variablen exportieren. Zur genaueren Betrachtung wird nachfolgend mit der Twitter-Timeline ein kleines Svelte-Projekt gestartet.

## Twitter-Timeline

Als ein etwas fortgeschrittenes Beispielprojekt soll eine einfache Replikation der Twitter-Timeline dienen. Dabei steht eine sinnvolle Unterteilung in einzelne Komponenten und deren Interaktion im Vordergrund. Ebenfalls wichtig ist eine gute Integration des Stylings und eine robuste Behandlung von HTTP-Anfragen. Als darunterliegende CSS-Bibliothek kommt Bootstrap [5] zum Einsatz.

**Listing 6: Hauptkomponente (App.svelte)**

```
<script>
  import Timeline from './Timeline.svelte';
  import { onMount } from 'svelte';

  let posts = undefined;

  onMount(() => {
    fetch(
     'https://jsonplaceholder.typicode.com/comments')
    .then(res => res.json())
    .then(retrievedPosts => {
      posts = retrievedPosts;
    })
    .catch(err => {
      posts = err;
    });
  });
</script>

<div class="container">
  <div class="row">
    {#if posts === undefined}
      <b>Lade Daten ...</b>
    {:else if posts instanceof Error}
      <span style="color: red">Beim Laden ist ein
        Fehler aufgetreten.</span>
    {:else}
      <Timeline posts={posts} />
    {/if}
  </div>
</div>
```

Simuliert werden soll damit der Bau einer echten Applikation. Setzen Sie dazu zunächst ein HTML-Dokument wie in Listing 5 auf. Das Dokument dient als Basis, welche die globalen CSS-Styles und das notwendige JavaScript-Bundle referenziert. Wie bereits im ersten Beispiel wird auch hier Parcel zur Umwandlung des Codes genutzt.

Im ersten Schritt wird die aktuelle Timeline ausgelesen. Als Datengrundlage dient ein Satz Pseudokommentare, die das JsonPlaceholder-API bereithält. Listing 6 zeigt den Inhalt der Datei *App.svelte*, die gleichzeitig die Hauptkomponente der Applikation ist.

Die Variable *posts* wird genutzt, um die API-Anfrage sinnvoll und strukturiert zu behandeln. Die eigentliche HTTP-Anfrage erfolgt im *onMount*-Callback. Der Vorteil ist eine mögliche Verwendung im serverseitigen Rendering.

Die Klassen *container* und *row* wurden bereits über Bootstrap definiert. Beim Initialisieren zeigt der Code eine kurze Meldung bezüglich des Ladevorgangs an. Die *Timeline*-Komponente ist in einer weiteren Datei *Timeline.svelte* definiert und wurde explizit importiert. Hier folgt Svelte genau den ▶


**Das Twitter-Timeline-Beispiel** in Aktion (Bild 3)

Vorgaben von React, um die skalierbare Entwicklung sehr einfach zu machen.

Listing 7 zeigt den Code für die *Timeline*-Komponente. Dort wird die Variable *posts* exportiert, um als Eigenschaft einsetzbar zu sein. Außerdem wird im Code CSS in einem <*style*>-Tag verwendet. Svelte liefert dafür automatisch generierte CSS-Klassen, die genau auf die entsprechenden Elemente passen.

Ein Vorteil von Svelte gegenüber beispielsweise Web Components besteht darin, dass wie auch bei React beziehungsweise JSX die Übergabe von Parametern an Komponenten durch Standard-JavaScript-Objekte möglich ist. Wie bei JSX funktioniert der Wechsel mit geschweiften Klammern auch in Attributen. Die *TimelineEntry*-Komponente ist eine abschließende Hülle für den eigentlichen Inhalt. Dieser wird in eineKomponentenamens *TimelineEntryContent* verpackt und ausgespielt. Dabei setzt Letztere alle Werte in den exportierten Feldern, während *TimelineEntry* einen allgemeinen *post* erwartet. Zusätzlich wird noch mehr auf Eigenschaften wie die Ausrichtung (*aligned*) eingegangen.

Analog zum Einstiegsprojekt können Sie auch diesen Code von Parcel bauen lassen. Anstatt auf die *index.js* Datei zu verweisen, sollten Sie in diesem Fall direkt die Datei *index.html* als Einstiegspunkt verwenden. Bild 3 zeigt die Timeline der fertigen Applikation.

Selbstverständlich können Sie zu diesem Zeitpunkt noch mehr Funktionalität hinzufügen oder weitere kleinere Spielereien integrieren. Der vollständige Quellcode ist auf GitHub hinterlegt [6].

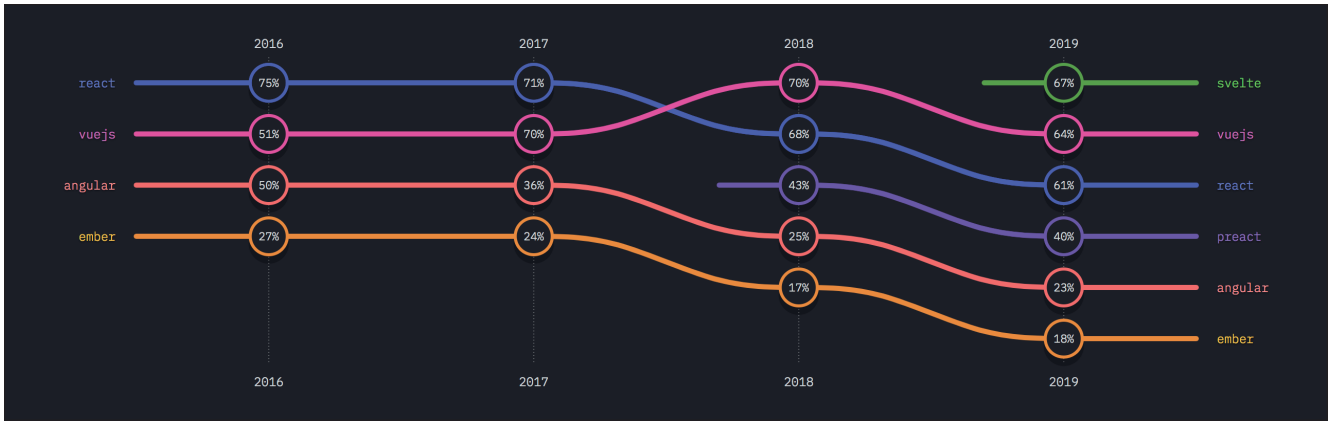
## Sapper

Um Svelte wirklich in aller Tiefe kennenzulernen, empfiehlt es sich, die Online-Tutorials durchzuarbeiten. Diese gehen in aller Breite und Tiefe interaktiv auf die einzelnen Facetten ein [7]. Eines der Details, die nahezu nie diskutiert werden, ist dabei das abschließende Bauen von Anwendungen für die echte Welt.

Für die Beispiele zu diesem Artikel haben Sie eine elegante Methode durch die Verwendung von Parcel gesehen. Im Tutorial wird jedoch exklusiv auf vorhandene Templates für Rollup beziehungsweise Webpack verwiesen.

Eine weitere Möglichkeit bietet die Verwendung eines ganzes Frameworks. Dafür gibt es mit Sapper eine spannende Alternative [8].

Sapper versteht sich als Framework zur Entwicklung von serverseitig getriebenen Webapplikationen auf Basis von




**Interesse an Frontend-Frameworks** laut der Umfrage „State of JS“ [9] (Bild 4)

Svelte. Kurz gesagt eine Alternative zu Next oder Nuxt für Entwickler mit React- beziehungsweise Vue-Hintergrund. Sapper kümmert sich um eine effiziente serverseitige Aufbereitung von Svelte-Komponenten, die anschließend besonders geschmeidig in clientseitige Interaktivität übergehen. Dieses „Progressive Enhancement" ist für besonders schnelle Single Page Applications (SPA) essenziell. Viele SPAs leiden unter den vergleichsweise langen Initialisierungsroutinen. Ein weiterer Aspekt, der durch Sapper verbessert werden soll, ist die Suchmaschinenoptimierung. Dies ist klassischerweise auch eine Schwachstelle von SPAs, die nahezu komplett in JavaScript geschrieben wurden.

In Sapper stellt jede Svelte-Komponente im Ordner *src/routes* eine eigene Seite dar. Dabei gibt der Dateiname klassischerweise die Route vor. Spezialtags wie *svelte:head* können dazu verwendet werden, Metainformationen oder andere globale DOM-Elemente zu setzen. Die folgenden Codezeilen zeigen die Deklaration der Route */about*:

```
<!-- src/routes/about.svelte -->
<svelte:head>
  <title>Über</title>
</svelte:head>

<h1>Über diese Seite</h1>
<p>Ein bisschen Text ...</p>
```

**Listing 7: Die Zeitleiste (Timeline.svelte)**

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

  export let posts = [];
</script>

<style>
  .timeline-centered {
    position: relative;
    margin: 30px 0;
  }

  /* mehr Styles */
</style>

<div class="timeline-centered">
  {#each posts as post, i}
    <TimelineEntry
      post={{
        date: post.created,
        message: post.body,
        typeId: post.type,
        user: post.userName,
      }}
      aligned={i % 2 ? 'left' : 'right'} />
  {/each}
</div>
```

Sapper hält sich bei der Umsetzung ziemlich genau an die Vorgaben von Next.js. Hier wird Convention-over-Configuration gelebt. Als Ergebnis erhält man ein geradliniges Framework, das vor allem Next.js-Entwicklern sehr vertraut vorkommen dürfte.

## Fazit

Mit Svelte wurde nach Jahren endlich wieder ein interessantes Framework für die Frontend-Webentwicklung veröffentlicht. Neben einem interessanten Ansatz bringt Svelte eine Menge nützlicher Werkzeuge mit. Daher verfügt das Framework bereits jetzt über ein spannendes Ökosystem. Das Interesse der JavaScript-Community ist – wie Bild 4 zeigt – ebenfalls sehr groß [9].

Der wohl größte Nachteil von Svelte war die fehlende Unterstützung für TypeScript. Zwar konnten beliebige TypeScript-Module konsumiert und durch einen Bundler transpiliert werden, der JavaScript-Code in Svelte-Komponenten war allerdings nicht inkludiert. Dieser Code lief primär über den Svelte-Compiler und musste daher von diesem verstanden werden. Das hat sich laut des Blogposts [10] auf der Svelte-Seite inzwischen geändert.

Der Einstieg wird durch die großartige Dokumentation inklusive einer dynamischen REPL recht einfach gestaltet. Mit Sapper gibt es auch bereits eine durchaus ausgereifte Plattform für die serverseitige Verwendung. Dadurch ist Svelte eine performante und leichtgewichtige Alternative zu den bekannten Frameworks, die in vielen Aspekten das Beste aus Angular, React und Vue vereint. ■


*[1] Svelte, https://svelte.dev*
*[2] Parcel, https://parceljs.org*
*[3] Svelte REPL, www.dotnetpro.de/SL2010Svelte1*
*[4] Svelte Compiler Handbook, www.dotnetpro.de/SL2010Svelte2*
*[5] Bootstrap, https://getbootstrap.com*
*[6] Die Beispielanwendung auf GitHub, www.dotnetpro.de/SL2010Svelte3*
*[7] Adding Data, www.dotnetpro.de/SL2010Svelte4*
*[8] Sapper, https://sapper.svelte.dev*
*[9] Umfrageergebnisse zu Frontend-Frameworks, www.dotnetpro.de/SL2010Svelte5*
*[10] Svelte <3 TypeScript, www.dotnetpro.de/SL2010Svelte6*



**Dr. Florian Rappl**
ist Solution Architect bei smapiot mit Einsatzgebiet digitale Transformation. Als Microsoft MVP für Entwicklungstools befasst er sich mit Themen wie C#/.NET, TypeScript sowie skalierbaren Backends und Frontends in der Cloud.
**@FlorianRappl**


**dnpCode** A2010Svelte