SVELTE – EIN ÖKOSYSTEM FÜR VANILLA-JAVASCRIPT (TEIL 1)

Framework und Compiler für hocheffizienten Code

Im Zentrum des Svelte-Frameworks steht ein Compiler, der framework-unabhängigen JavaScript-Code produziert.

Im Gegensatz zu den bekannten Frameworks wie Angular, React oder Vue.js verfolgt Svelte nicht den Ansatz eines Virtual DOM (Document Object Model). Vielmehr übersetzt der in Svelte integrierte Compiler HTML-Templates in reinen (Vanilla-)JavaScript-Code, der das DOM einer Web-App direkt manipuliert.

Beim Übersetzen greift der Compiler auf standardisierte Vorgaben zurück. Damit enthält eine Svelte-App keinen Programmcode von externen Bibliotheken, insbesondere keine framework-spezifischen Module. Was sich in der Regel positiv auf die Startupzeit und das Laufzeitverhalten der Web-App auswirkt. Zudem entfallen alle laufzeitintensiven Operationen von komplexen Algorithmen, um Änderungen im Virtual DOM mittels selektiver Updates auf den Anwendungsbaum zu übertragen.

Ein Programmierer in Svelte muss sich nicht mit dem Overhead des Virtual DOM befassen – er kann sich auf die Umsetzung der fachlichen Anforderungen beschränken. Ein Entwickler programmiert Komponenten mittels einer Erweiterung der JavaScript-Syntax etwa vergleichbar JSX (JavaScript XML) in React. Diese Arbeitsweise – Templating genannt – orientiert sich am Aufbau von HTML-Dateien. Zusätzliche JavaScript-Elemente für die Einbindung ins DOM bettet der Programmierer in geschweifte Klammern ein. Für die reaktive Programmierung kommen JavaScript-Statements zum Einsatz, die durch das Label *$:* am Beginn gekennzeichnet werden. Um Event-Handler für DOM-Events einzubetten, greift man auf das HTML-Attribut *on:* zurück.

Seit dem Aufkommen von JavaScript hat sich über die Jahre hinweg rund um die Programmiersprache ein sehr großes Ökosystem entwickelt. Neben der eigentlichen Sprache (für die inzwischen unter der Bezeichnung ECMAScript ein Standard vorliegt), gehört Node.js als Standardlaufzeitumgebung für JavaScript zum Fundament des Ökosystems.

ECMAScript beschreibt eine dynamisch typisierte, objektorientierte, aber klassenlose Programmiersprache. In JavaScript lässt sich je nach Bedarf objektorientiert, funktional oder prozedural programmieren. Dabei umfasst die Sprachspezifikation eine ereignisgesteuerte Architektur und den kompletten Funktionsumfang einer imperativen Programmiersprache.

Entwicklung und Ausführung von JavaScript-Code erfolgt plattformunabhängig über Node.js; kurz Node genannt. Diese Laufzeitumgebung steht als Open-Source-System zur Verfügung und unterstützt alle gängigen Betriebssysteme Linux, macOS und Windows.

Die gute Performance einer Anwendung stellen einige direkt in das Binärpaket kompilierten Node-Module sicher. Zusätzlich hat Google eine ressourcensparende Laufzeitumgebung V8 für JavaScript entwickelt. V8 findet sowohl in Webbrowsern als auch in Node.js selbst Verwendung. Dabei unterstützt V8 eine Just-in-time-Kompilierung für nativen Maschinencode und weitere ausgefeilte Optimierungstechniken.

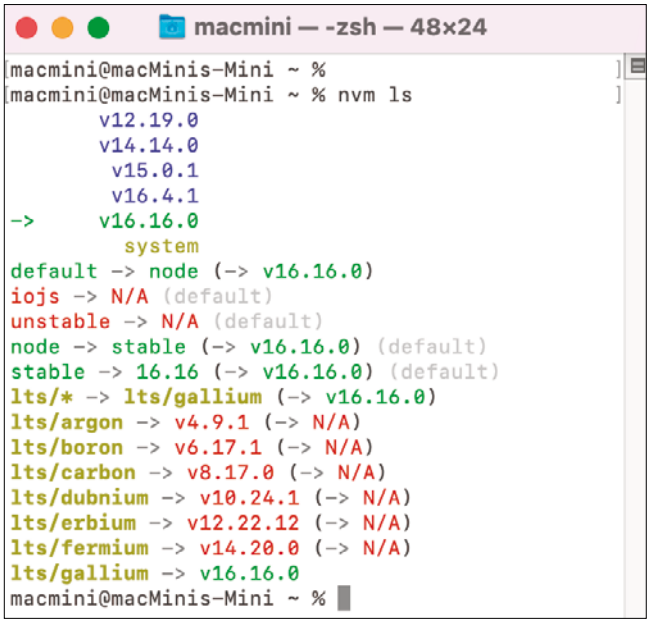
Durch ein zentrales Repository lassen sich mit dem Paketmanager npm weitere Pakete in die Entwicklung mit JavaScript einbinden. Über die Homepage von Node.js führt

Bild: https://svelte.dev/

man schnell eine Installation der aktuellen LTS (Long Term Support)-Version durch. Das zugehörige Setup-Programm von Node richtet neben der Laufzeitumgebung auch gleich die passende npm-Version ein. Um mit verschiedenen Versionen von Node.js auf der gleichen Hardware parallel zu arbeiten, empfiehlt sich der Einsatz des Node Version Manager (NVM). Mit NVM nutzt man leicht verschiedene Node-Umgebungen für Entwicklung, Test oder Produktion, die man zielgerichtet umschaltet. Eine Deinstallation einer anderen Node-Version ist vorher nicht erforderlich!

Der gängige NVM ist auf POSIX-kompatible Shells ausgerichtet. Deshalb gibt es für Windows eine spezielle Implementierung von Corey Butler: NVM for Windows. Der Befehl *nvm use <version-nr>* wechselt zwischen installierten, verschiedenen Node-Versionen. Sollte die gewünschte Version von Node auf dem System nicht vorhanden sein, nimmt *nvm install <version-nr>* schnell eine Installation vor.

Die NVM-Software besitzt ein integriertes Hilfesystem, das man über *nvm -h* erreicht. Eine Installation der aktuellen LTS (Long Term Support)-Version nimmt der Befehl *nvm install --lts* vor. Die aktuell durch NVM dem Betriebssystem aktivierte Version zeigt der Befehl *nvm current* an. Alle auf dem Arbeitsplatz installierten Node-Versionen gibt der Befehl *nvm ls* aus (Bild 1).



Mit der NVM-Software kann man parallel unterschiedliche Node-Versionen auf derselben Hardware in verschiedenen Terminal-sessions nutzen (Bild 1)

Mehrere Möglichkeiten für den Einstieg in ein Svelte-Projekt

Die Homepage von Svelte beschreibt zwei Möglichkeiten, ein Grundgerüst für ein Svelte-Projekt zu erhalten: Einsatz der Svelte REPL (Read-Eval-Print-Loop) oder von degit einem Projekt-Scaffolding-Tool. Die Svelte REPL entspricht einer rudimentären Online-IDE, die interaktiv Svelte-Befehle sofort ausführt. Änderungen im Quellcode zeigt Svelte REPL sofort an, vorausgesetzt in der Toolbar ist der Eintrag *Result* ausgewählt. Um den Quellcode für eine spätere Weiterentwicklung zu nutzen, speichert die Online-IDE diesen in GitHub ab.

Über das Icon *download zip file* lädt Svelte REPL den Quellcode als ZIP-Datei *svelte-app.zip* herunter. Die Online-IDE eignet sich nicht nur für den ersten Einstieg in Svelte, sondern auch für das Erkunden der Arbeitsweise des Svelte-Compilers.

Um ein Startup-Projekt für die Svelte-Entwicklung einzurichten, hat der Macher von Svelte Rich Harris ein Scaffolding-Tool *degit* (ausgesprochen: depth git) programmiert. Es handelt sich um ein npm-Package, das in Anlehnung an den Git-Befehl *git clone --depth 1 <repository> <verzeichnis>* eine lokale Kopie eines Git-Repositorys im Ordner *<verzeichnis>* anlegt. Zu Beginn installiert man mit *npm install -g degit* das Scaffolding-Tool oder man benutzt es direkt über npx (einem in npm integrierten CLI-Tool, um sofort npm-Pakete über deren CLI ohne Installation auszuführen). Das GitHub-Repository *github.com/sveltejs/template* enthält ein Template mit einem Grundgerüst (Scaffold) für eine Svelte-App.

Der Befehl *npx degit sveltejs/template <projektname>* lädt dieses Grundgerüst in ein neues Projekt mit dem Namen *<projektname>* herunter. Diese Vorgehensweise sollte man nur noch solange verwenden, wie sie die Svelte-Dokumentation beschreibt, da dieses Projekt-Template aktuell nicht mehr gewartet wird.

Um mit TypeScript zu programmieren, erzeugt der Node-Befehl *node scripts/setupTypeScript.js* daraus ein TypeScript-Projekt. Abschließend installiert man alle für die Svelte-Entwicklung benötigten npm-Pakete mittels *npm install*. Jetzt enthält der neue Projektordner alle notwendigen Bestandteile, um die Svelte-App auszuführen, zu testen und weiterzuentwickeln. Als Build-Tool kommt in diesem Fall *Rollup* und für lokales Testing das npm-Package *sirv(-cli)* zum Einsatz. Lokales Testing erfordert wie üblich einen Build der Web-App: *npm run build*.

Seit einiger Zeit empfiehlt auch Rich Harris (der Schöpfer von Svelte) für die Erzeugung eines Grundgerüst einer Svelte-App den Einsatz von Vite. Bei Vite handelt es sich ebenfalls um ein Scaffolding-Tool, das vom Vue.js-Entwickler Evan You stammt und sich für mehrere JavaScript-Frameworks eignet (Bild 2).

Mit dem Befehl *npm create vite@latest* startet man den Scaffolding-Prozess zur Einrichtung eines Entwicklungsprojekts. Nach Auswahl von Svelte und der gewünschten Programmiersprache JavaScript oder TypeScript erzeugt vite das zugehörige Grundgerüst im angegebenen Projektordner. Für Ausführung, Test oder Weiterentwicklung der Svelte-App ▶

Svelte – ein Ökosystem für Vanilla-JavaScript

1: Svelte – Compiler und Framework
2: Entwicklungstools perfektioniert für Svelte
3: User-Interface-Tools/Komponenten für Svelte
4: SvelteKit & Co. – Framework und Tools für Svelte-Apps
5: Svelte-Programmierung für Desktop & Mobile

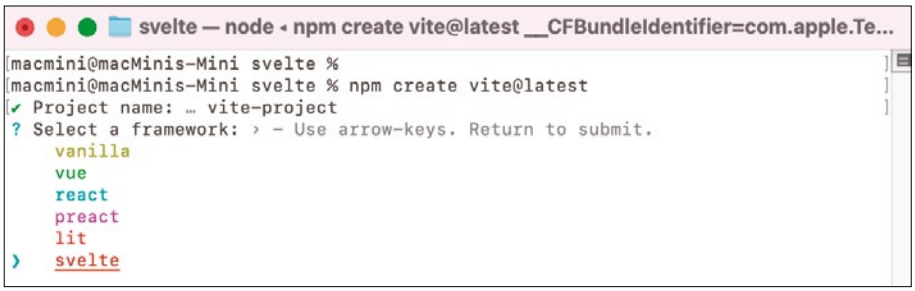
führt man mit *npm install* noch eine Installation der erforderlichen Bestandteile für das Projekt durch. Dieses Svelte-Projekt greift über vite auf esbuild für den Build in Entwicklung und Test der Anwendung zurück. Zur Auslieferung der Software nutzt vite für den Build-Prozess derzeit noch den Rollup-Bundler.

Projektstruktur und Workflow beim Arbeiten in einem Svelte-Projekt

Anstatt interaktiv mit Vite ein Svelte-Projekt zu erzeugen, erledigt dies sofort ohne Benutzerabfragen der Terminalbefehl *npm create vite@latest <projektname> -- --template svelte*. Eine besondere Rolle nimmt die *package.json*-Datei im Hauptverzeichnis des Projekts ein. Diese Datei enthält alle für das Arbeiten mit dem Projekt benötigten Metainformationen.

Dazu gehört der Name des Moduls/der Anwendung, der Name des Autors, eine kurze Beschreibung zum Funktionsumfang der Svelte-App, die Versionsnummer und alle Abhängigkeiten zu anderen npm-Paketen mit deren Versionsnummern. Der Befehl *npm install* liest diese Abhängigkeiten aus *package.json* und installiert alle benötigten Bestandteile, so dass anschließend ein lauffähiges Projekt zur Verfügung steht.

Zusätzlich dokumentiert der Bereich *scripts* in der *package.json*-Datei den kompletten Workflow für das Arbeiten mit einem Svelte-Projekt. Der Befehl *npm run build* führt einen Build der Web-App durch und legt die Ergebnisse im Unterverzeichnis dist (Einsatz von Vite) des Projekts ab. Der im *scripts*-Bereich genannte Befehl *npm run dev* startet einen lokalen Webserver mit HMR (Hot Module Replacement) für Entwicklungstests. HMR bewirkt, dass alle im Quellcode durchgeführten Änderungen nach dem Speichern sich sofort in der Web-App ohne vorheriges Reload der Seite widerspiegeln. Dazu muss die Svelte-App nur einmal über die bei der Ausführung von *npm run dev* genannte URL im Browser gestartet werden. Der letzte Befehl *npm run preview* startet ebenfalls einen lokalen Webserver. Greift aber über die bei der Ausgabe genannte URL auf die im Deployment-Ordner *dist* abgelegt (produktionsnahe) Web-App zu.



Vite unterstützt für Frontend-Tooling (Scaffolding, Test, Build) neben Svelte weitere JavaScript-Frameworks (Bild 2)

Ein Svelte-Projekt enthält im Projektordner vier Unterverzeichnisse *dist, node_modules, public* und *src*. Der Ordner *dist* umfasst alle für das Deployment der Svelte-App erforderlichen Dateien. Dazu gehören die *index.html*-Datei mit dem *assets*-Unterverzeichnis sowie alle für die Funktionalität der Web-App benötigten Dateien und Ressourcen. Der Ordner existiert erst nachdem *npm run build* mindestens einmal ausgeführt wurde. Dieser *dist*-Ordner stellt den Ausgangspunkt für das Deployment der Web-App dar. Bei einem mit degit erzeugten Svelte-Projekt, legt Rollup seine Build-Ergebnisse im Ordner *public/build* ab.

In einem mit Vite erzeugten Svelte-Projekt kommt dem *public*-Ordner keine Bedeutung zu. Im Unterschied zum Build für Entwicklungstests ist das Production-Bundle für das Deployment in die Produktion auf ein Minimum reduziert und für den Menschen nicht mehr verständlich.

Im Unterverzeichnis *node_modules* findet man alle heruntergeladenen npm-Pakete. Dazu gehört unter anderem Svelte selbst, der Modul-Bundler Rollup, das npm-Paket für Entwicklungstests sirv oder das Vite-Frontend-Tool. Die Programmierung der Svelte-App erfolgt über die Dateien im *src*-Ordner.

In diesem Ordner legt man alle selbst geschriebenen Svelte-Komponenten, sowie weiteren Quellcode ab, der sich nicht direkt einer Komponente zuordnen lässt. In großen Projekten mit vielen Komponenten empfiehlt es sich, für diese einen eigenen Ordner anzulegen.

Eine Svelte-Komponente befindet sich immer in einer Datei mit dem Namen der Komponente und der Erweiterung *.svelte*. Im Scaffolding-Projekt enthält der src-Ordner die Datei *App.svelte*: Dabei handelt es sich um die Hauptkomponente der Web-App, welche die Funktionalität und/oder das User-Interface anderer Komponenten auf Bedarf benutzt. Die

Arbeitsweise des Svelte-Compilers im Überblick

Der Compiler von Svelte parst alle .svelte-Dateien, überführt diese in einen AST (Abstract Syntax Tree), analysiert ihn und generiert daraus JavaScript und CSS. Im Quellcode einer .svelte-Komponente kann über den Tag *<svelte:options ... />* der Compile-Vorgang gesteuert werden. Als Alternative ist eine Steuerung des Compile-Vorgangs über die *svelte.config.js*-Datei beim Eintrag *compilerOptions* möglich. Für die svelte:options-Einträge stehen als Props zur Verfügung: *immutable, accessors, namespace* und *tag*. So erreicht man im Falle nicht veränderbarer Komponenten mit *<svelte:options immutable=true>* eine Optimierung der Laufzeit für die Svelte-App. Um wiederverwendbare *Web Components* mit Svelte zu entwickeln, gibt man bei den compilerOptions die beiden Einträge *dev: !production* und *customElement: true* an. Zusätzlich benötigt man jetzt noch bei den Svelte-Komponenten den Eintrag *<svelte:options tag="custom-tag" />*. Der Wert des *tag*-Attributs entspricht dem Custom/HTML-Tag der Web-Komponente.

zweite Datei im *src*-Ordner *main.js* stellt den Einstiegspunkt der Web-App dar; sie lädt die Top-Level-Komponente *App.svelte* und führt sie aus.

Syntax und Struktur des Quellcodes einer Svelte-Komponente

Unter eine Svelte-Komponente versteht man einen Bestandteil des User-Interface, der eine bestimmte Aufgabe übernimmt. Eine Svelte-App besteht immer aus einer Anzahl von Komponenten, die aufeinander aufbauen. Dabei entspricht eine Svelte-Komponente einer .svelte-Datei mit Quellcode, der aus HTML, CSS und JavaScript besteht. Alles was die Svelte-Komponente für die Ausführung ihrer Funktionalität benötigt, befindet sich in dieser .svelte-Datei. Typischerweise speichert man innerhalb eines größeren Projekts Komponenten in einem speziellen Ordner zum Beispiel */src/components* ab.

Eine Komponente kann sich aus mehreren anderen Komponenten zusammensetzen. Die in einer Komponente enthaltenen anderen Komponenten nennt man *Nested/Geschachtelte Komponenten*. Die App.svelte-Komponente nennt man auch *Wurzel/Root-Komponente* der Web-App; sie dient als Container, um andere Komponenten aufzunehmen. Ein Entwickler baut nach diesem Prinzip die gesamte Web-App auf. Komponenten können untereinander kommunizieren und Daten austauschen. Diese Kommunikation erfolgt über *props* (Eigenschaften/Attribute), *events* (Ereignisse/Nachrichten) oder durch das Lesen von Daten aus externen Ressourcen (*get data*).

Die *.svelte*-Datei einer Komponente setzt sich aus drei verschiedenen Blöcken (JavaScript, HTML, CSS) zusammen (Bild 3). Für alle drei Blöcke steht die bekannte Syntax der jeweiligen Sprache zur Verfügung. Da jeder der drei Abschnitte eindeutig identifiziert werden kann, spielt deren Reihenfolge innerhalb einer Komponente keine Rolle. Für alle drei Blöcke stellt Svelte zusätzlich zu den üblichen Sprachelementen syntaktische Erweiterungen bereit. Jeder dieser drei Code-Abschnitte übernimmt eine bestimmte Aufgabe und muss innerhalb einer Komponente immer vorhanden sein:

- *<script>*-Block: enthält Daten und Funktionalität (Logik) einer Komponente, die mittels JavaScript oder TypeScript programmiert wird
- *HTML*-Block: realisiert die Struktur und den Aufbau des User-Interface der Komponente; dabei kommen beliebige HTML-Elemente zum Einsatz. Der HTML-Block ist nicht wie die beiden anderen durch ein bestimmtes Markup-Tag vorgegeben. Alles außerhalb des <script>- und <style>-Blocks kann HTML-Markup enthalten. Es empfiehlt sich dennoch alle HTML-Elemente über einen <main>-Block abzuhandeln
- *<style>*-Block: übernimmt die Gestaltung, das heißt das Aussehen der Komponente auf dem User-Interface, dazu greift man auf CSS-Code zurück. Damit realisiert dieser Block das Styling der Komponente mittels CSS-Markup.

HTML-Markup und <style>-Block nennt man auch View, in Abgrenzung dazu bezeichnet man den *<script>*-Block auch

Eine Svelte-Komponente besteht aus drei Code-Abschnitten: JavaScript-Quellcode, HTML-Elemente und CSS-Code (Bild 3)

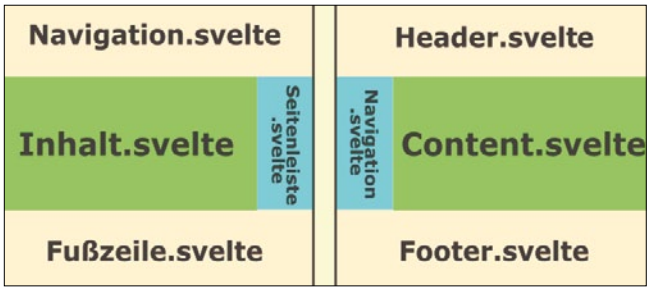
als Logik. Für alle Blöcke zusammen verwendet man gelegentlich auch den Begriff Template-Syntax. Der Quellcode einer Komponente ist von den anderen Komponenten gekapselt. Damit bleibt das Innere einer Komponente den anderen verborgen. So wirkt sich die Gestaltung einer Komponente mittels CSS nur auf diese selbst und nicht auf andere Komponenten aus. Trotzdem sind global gültige CSS-Regeln möglich. Diese legt man in einer Datei *global.css* ab und macht sie so innerhalb eines Projekts für alle Komponenten zugänglich.

Funktionsweise des Frameworks und Komponenten-Hierarchie in einer App

Eine in Svelte programmierte Web-App besteht aus einer Sammlung von Komponenten; damit teilt sich eine Web-App in einzelne wiederverwendbare Komponenten auf. Eine einfache Webseite besteht aus den Bereichen Navigation, Inhalt, Seitenleiste und Fußzeile (Bild 4).

In einer Svelte-App entspricht jeder dieser Bereiche einer eigenen *.svelte*-Datei. Konzeptionell kann man sich eine Komponente (also eine *.svelte*-Datei) wie eine JavaScript-Funktion vorstellen. Eine derartige Komponente nimmt Eingaben entgegen, die in Svelte *props* (Eigenschaften/Attribute) genannt werden und gibt Ausgaben zurück. Im Unterschied zu anderen Frameworks wie Angular, React oder Vue.js funktioniert eine Svelte-Komponente ohne zusätzliche Module des Frameworks.

Einzelne Komponenten lassen sich in anderen Komponenten wiederverwenden, importieren und ausführen. Dadurch entsteht eine Verschachtelung von Komponenten – der Einstiegspunkt einer Svelte-App stellt die *main.js*-Datei dar. Diese Datei legt im Wesentlichen die Haupt/Root-Komponen- ▶



Zwei Beispiele für den Aufbau einer Webseite mittels einzelner Svelte-Komponenten (Bild 4)

te der Svelte-App fest. In der Regel heißt die erste Komponente *app.svelte*. Innerhalb dieser Komponente verschachtelt der Entwickler alle weiteren Komponenten. So nimmt der Quellcode von *app.svelte* einen Import für die Navigation, den Inhalt oder die Fußzeile vor. Beim Kompilieren der fertigen Svelte-Komponenten fügt das Framework den größten Teil der Funktionen zum Erstellen und Aktualisieren des User-Interface als eigene JavaScript-Funktionen in die Komponenten ein.

Der Module-Bundler fasst am Ende den gesamten Svelte-Code zu einer einzigen JavaScript-Datei zusammen. So dass man sich die Funktionalität der Svelte-App in einer einzigen JavaScript-Datei widerspiegelt; diese heißt häufig *bundle.js*. Während bei den Frameworks Angular, React und Vue.js die auszuliefernde Web-App zusätzlich zum App-Bundle auch framework-spezifische Module enthält, benötigt das Bundle der Svelte-App keine Framework-Spezifika.

Vielmehr erledigt Svelte die Arbeit am HTML-DOM (Document Object Model) des Browsers, um die Benutzeroberfläche zu erstellen und die Interaktionen zu ermöglichen, mit nativen JavaScript-Features.

Den dazu erforderlichen JavaScript-Code für die Bearbeitung des DOM im Browser erzeugt der im Svelte-Framework enthaltene Compiler. Er generiert die dazu benötigten JavaScript-Funktionen basierend auf dem DOM-API der Webbrowser. Die generierten JavaScript-Funktionen der Svelte-App manipulieren entsprechend das DOM der Webseite. Im Bedarfsfall kann sich der Svelte-Compiler bei den kompletten Web-APIs bedienen. Damit entfällt in Svelte die Einbindung zusätzlicher framework-spezifischer Module für die Bearbeitung des Visual DOM – alles in der Svelte-App läuft in purem (Vanilla-)JavaScript. Als Programmierer muss man sich nicht mehr mit der Codierung derartiger Visual DOM-Spezifika befassen. Somit spart der Entwickler Arbeitszeit und für eine Svelte-App ergibt sich eine geringere Größe verbunden mit einer in der Regel viel besseren Performance.

Initialer Startvorgang einer Svelte-App erzeugt eine Komponenten-Hierarchie

Eine mit Svelte realisierte Web-App entsteht durch Verschachtelung mehrerer Komponenten, von denen die Root/Wurzel-Komponente den Startvorgang auslöst. Diese Verschachtelung definiert eine Hierarchie von Komponenten – dabei nutzt eine in der Hierarchie höherstehende Komponente (die Eltern-Komponente) die Funktionalität einer in der Hierarchie tieferstehende Komponente (einer Child/Kind-Komponente).

Die Root/Wurzel-Komponente stellt die Hauptkomponente dar; bei ihr handelt es sich um die erste Eltern-Komponente der Svelte-App. Da jeder Webserver beim Startvorgang einer Web-App eine *index.html*-Datei anzieht, referenziert diese die Root/Wurzel-Komponente.

Zunächst lokalisiert man in der *index.html*-Datei im *<body>*-Tag lediglich die Anweisung *<script type="module" src="/src/main.js"></script>*. Dieses HTML-Element macht deutlich, dass darüber die *main.js*-Datei ausgeführt wird. Erst der JavaScript-Quellcode in der *main.js*-Datei (Listing 1) referenziert die Root/Wurzel-Komponente *app.svelte* der Web-App. Die *main.js*-Datei entspricht also dem Einstiegspunkt der Web-App und wird zuerst geladen. Zu Beginn importiert die erste Zeile die Hauptkomponente und führt den dort stehenden Quellcode direkt aus. Über diesen Import wäre es auch möglich, weitere Komponenten der Svelte-App beispielsweise für die Navigation oder die Seitenleiste gleich mit zu laden.

Listing 1: Von degit-Template erzeugte main.js-Datei

```
import App from './App.svelte';

const app = new App({
        target: document.body,
        props: {
                name: 'world'
        }
});

export default app;
```

Verschiedene Attribute/Parameter werden festgelegt

Die auf den Import folgende Codezeile initialisiert die App. Mittels der *new App()*-Anweisung erzeugt die Svelte-App ein Objekt. Für dieses App-Objekt legt die Svelte-App verschiedene Attribute/Parameter fest. Der erste Parameter *target* beschreibt das Ziel des Elements im DOM. Aus der Anweisung *target: document.body* erzeugt der Svelte-Compiler die Anweisung *target: document.getElementById('app')* und fügt damit das Element in das HTML-* *-Tag mit dem Namen (der id) *app* ein. Damit legt der *target*-Parameter fest, an welcher Stelle die Ausgabe des Codes von *app.svelte* und allen nachfolgenden in der Hierarchie untergeordneten Komponenten gerendert werden.

Schaut man sich die vom Compiler für die Produktion erzeugte *index.html*-Datei im *build (dist)*-Ordner an, so spiegelt sich dort genau dieser Sachverhalt wider: * *. Dieses * *-Tag im *<body>*-Tag der HTML-Datei fügt die gesamte Anwendung ein und gibt sie anschließend auf der Webseite aus. Der zweite Parameter der *new App()*-Anweisung *props* steht für Property/Eigenschaft, die man sich als Eingabe oder als globale Variable vorstellen kann. Im vorliegenden mit degit erzeugten Grundgerüst für eine Svelte-App legt die *main.js*-Datei zusätzlich eine Variable *name* mit dem Inhalt *world* fest.

Alle derart über das *props*-Attribut des App-Objekts definierte Variablen stehen in jeder Komponente der gesamten Anwendung zur Verfügung, da die *main.js*-Datei als Einstiegspunkt immer als erstes von der Web-App geladen wird. Die letzte Anweisung am Ende der *main.js*-Datei nimmt einen Export des *app*-Objekts vor. Grundsätzlich kann jede Komponente einer Svelte-App andere Komponenten oder Variablen/Daten von anderen Dateien importieren.

Innerhalb einer Svelte-Komponente (Listing 2) macht die *export*-Anweisung im *<script>*-Tag eine Variable für andere Komponenten als Props bekannt und so über einen Import zugänglich. Um eine andere Komponente in einer Svelte-Komponente zu benutzen, bindet man diese mittels einer *import*-Anweisung im *<script>*-Tag (*import Kind from './Kind.svelte'*) ein und verwendet diese als Kind-Komponente im HTML-Abschnitt: *<Kind/>*.

Einrichten einer integrierten Entwicklungsumgebung/IDE für Svelte

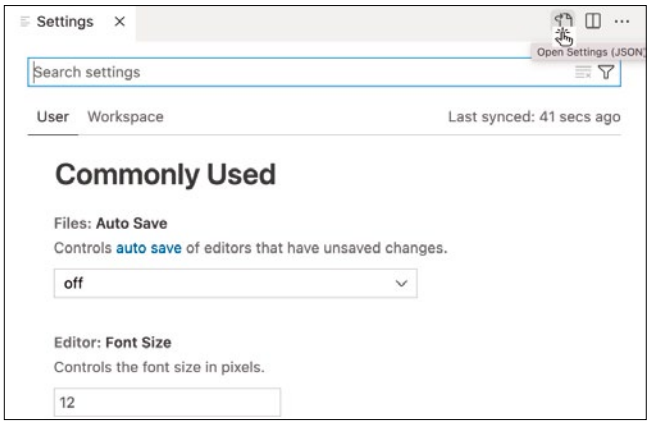
Als integrierte Entwicklungsumgebung (Integrated Development Environment/IDE) empfiehlt sich der Einsatz von VS Code (Visual Studio Code) oder VSCodium. Während VS Code lediglich unter MIT-Lizenz steht, mit der Berechtigung von Microsoft Nutzungsdaten zu übermitteln, basiert VSCodium ausschließlich auf Open-Source-Code und enthält nicht die Telemetrie-Funktionen von Microsoft.

Für beide IDEs steht im Marketplace (für VS Code) beziehungsweise in der Open VSX Registry (für VSCodium) die Extension Svelte for VS Code zur Verfügung. Diese Erweiterung nutzt den Svelte Language Server, um die Codierung von Svelte-Quellcode mit Syntax-Highlighting und dem Intellisense-Feature zur automatischen Code-Vervollständigung zu erleichtern.

Der Svelte Language Server unterstützt zusätzlich eine Codierung mit HTML, CSS/SCSS/LESS und TypeScript/JavaScript. Für HTML greift die Svelte-Extension auf *vscode-html-languageservice* und für CSS/SCSS/LESS auf *vscode-css-languageservice* zurück. Dadurch steht auch das Emmet-Feature für eine schnellere Codierung mittels Tasturkürzel zur Verfügung. Für die Formatierung von *.svelte*-Dateien hat Svelte for VS Code den Prettier for Svelte 3 components (prettier-plugin-svelte) integriert. Dieses Plugin nutzt zur Formatierung von HTML, CSS und JavaScript den bekannten Prettier Code-Formatter. Deshalb muss nur die Erweiterung Svelte for VS Code in der IDE installiert werden.

Listing 2: <script>-Tag einer Svelte-Komponente

```
…
<script>
// Beim Anlegen der Komponente führt Svelte diesen
// Code-Block aus.
// Variablen als prop der Komponente fuer andere
// zugaenglich machen
  export let name;
  export function get(name){}
…
/* Von der Komponente benoetigte Variablen anderer
   Komponenten ueber Import zugaenglich machen */
  import Counter from './lib/Counter.svelte'
/* Eine Kind- analog der Eltern-Komponente
   zugaenglich machen */
  import Kind from './Kind.svelte'
</script>
…
<main>
…
   /* Eine Kind-Komponente im HTML-Abschnitt
   verwenden */
   <Kind/>
…
</main>
```



Neben der GUI-Oberfläche für die Settings öffnet die IDE rechts oben über Open Settings (JSON) die zugehörige Textdatei (Bild 5)

Nach erfolgreicher Installation stehen alle genannten Features zur Verfügung – sobald man eine Datei mit der Erweiterung *.svelte* in der IDE öffnet und bearbeitet. Zusätzliche Einstellungen für das Arbeiten mit einer Svelte-App in der IDE definieren deren Settings.

Sollte Svelte for VS Code nicht wie erwartet funktionieren, gilt es zu prüfen, ob die Settings der IDE die gewünschten Features überschreiben. Die Settings bestehen aus zwei Bereichen User und Workspace; wobei Workspace nur in einem geöffneten Projekt vorkommt.

Neben dem GUI der Settings in der IDE gibt es auch noch eine textuelle Sicht über eine *settings.json*-Datei auf die Einstellungen. Mit der Textansicht lassen sich gesuchte Einstellungen in der Regel leichter als über das GUI finden, da das GUI wesentlich mehr nämlich alle möglichen und nicht nur die tatsächlich für die App gesetzten Einstellungen anzeigt.

Um die Vorgaben der *settings.json*-Datei in der IDE in ihrer Textform einzusehen, greift man in der GUI der Settings auf das erste oben links stehende Icon zu. Bewegt man den Cursor auf das zugehörige Icon-Symbol, so erscheint der Hover-Help/Tooltip *Open Settings (JSON)* (Bild 5).

Ein Klick auf dieses Icon-Symbol öffnet die *settings.json*-Datei in einem neuen Fenster. Unter dem Fenstertitel zeigt die IDE den Speicherort der *settings.json*-Datei in der File-Hierarchie des Betriebssystems an.

Klickt man auf das ···-Icon neben dem angezeigten Speicherort, so bietet die IDE alle vorhandenen Definitionsbereiche zur Auswahl an. Wählt man einen davon zum Beispiel *svelte.enable-ts-plugin* aus, so springt der Cursor an die zugehörige Stelle und zeigt den Bereich nach der zweiten Auswahl in gelber Farbe an. ▶

Svelte erleichtert die Programmierung von Single Page Applications (SPAs) – jede Quelldatei mit *.svelte*-Erweiterung entspricht einer eigenen GUI-Komponente. Der Quellcode in einer *.svelte*-Datei besteht immer aus den drei Abschnitten: JavaScript, HTML und CSS. Dadurch ist die Komponentenlogik (JavaScript), die Struktur (HTML) und die Gestaltung der Komponente eindeutig getrennt.

Nicht jeder der drei genannten Abschnitte muss zwingend vorhanden sein. Ein Vorteil der Komponenten aus Sicht der Software-Engineering ist ihre Modularisierung durch Abkapselung des Quellcode vor den anderen Komponenten (Information-Hiding). Jede Komponente besitzt ein eindeutiges Interface (Lokalität), wodurch sich die Wartung erleichtert und Seiteneffekte verhindert werden.

Sprachsyntax, Programmaufbau und flexible Arbeitsweise von Svelte

Was die Sprachsyntax für die drei Abschnitte betrifft, kann in jedem die gewöhnliche Syntax (sprich JavaScript/TypeScript, HTML, CSS) verwendet werden. Zusätzlich stellt Svelte eine Reihe von syntaktischen Erweiterungen bereit, die nachfolgend erläutert werden.

So erlaubt Svelte die Verwendung von JavaScript-Variablen im HTML-Code; diese führen zu einer Aktualisierung des DOM mit den Werten der JavaScript-Variablen. Eine Aktualisierung findet immer dann statt, wenn die Variable einen neuen Wert erhält. Im HTML-Codeabschnitt muss die JavaScript-Variable dabei mit geschweiften Klammern eingebunden sein: *{<variable-name>}*. Allerdings funktioniert diese Aktualisierung nur über eine direkte Zuweisung mit dem '='-Operator.

Eine Änderung der Elemente eines Arrays oder der Attribute eines Objekts hat keinerlei Auswirkungen auf das DOM. Jedoch führt eine wiederholte Zuweisung eines Werts innerhalb einer Schleife zu einer mehrfachen Aktualisierung des DOM. Zusätzlich kann man die gängigen Kontrollstrukturen von JavaScript wie Verzweigungen oder Schleifen für den Aufbau des DOM im HTML-Abschnitt nutzen.

Somit unterstützt Svelte ein konditionales Rendering für HTML-Elemente. Für eine Verzweigung führt Svelte ein *{#if …}… {/if}*-Konstrukt ein – die innerhalb des Konstrukt stehenden HTML-Anweisungen kommen nur zur Anwendung, wenn die bei *{#if…}* stehende Bedingung erfüllt ist. Um mittels einer Schleife gewisse HTML-Elemente wiederholt anzuzeigen, macht Svelte dem Programmierer eine spezielle *{#each … as …}…{/each}*-Syntax zugänglich. Bei jedem Schleifendurchlauf führt Svelte die innerhalb des Blocks stehenden HTML-Anweisungen aus.

Die Sprachsyntax von Svelte unterstützt direkt die reaktive Programmierung sowohl für das DOM als auch für die eigenen Datenstrukturen. Grundsätzlich aktualisiert Svelte Änderungen des Zustands einer Komponente im DOM automatisch – der integrierte Compiler erzeugt passende Aufrufe des DOM-APIs. Dazu muss der Programmierer im Unterschied zu anderen Frameworks keinerlei weitere Vorkehrungen treffen. Zusätzlich erklärt das Label *$:* vor einer Quellcode-Zeile, das Statement als deklarativ und aktiviert so die Reaktivität. Damit schaltet Svelte die Reaktivität für diese Codezeile an, so dass deren Änderungen ebenfalls automatisch aktualisiert und damit ausgeführt werden. Die Reaktivität mittels $: gilt sowohl für *Variable* und ihre Werte als auch für die reaktive Ausführung eines *Statements* wie: $: *console.log("Geänderte Zeichenkette", text)*.

VS Code/VSCodium–Konfiguration wiederverwenden

Arbeitet man an unterschiedlichen Arbeitsplätzen mit VS Code oder VSCodium, so erleichtert das Speichern der aktuellen Konfiguration den Wechsel eines Arbeitsplatzes. Beim Einsatz von VS Code oder VSCodium trifft man in der Regel immer individuelle Vorgaben, um selbst optimal mit der IDE zu arbeiten. Zu diesen individuellen Vorgaben gehören die Einstellungen in den Settings, für die Tastatur oder die Installation spezieller Erweiterungen.

VS Code und VSCodium besitzen das Feature Settings Sync, um die individuelle Konfiguration der IDE auf einen anderen Arbeitsplatz zu übertragen. Dazu benötigt man allerdings ein Benutzerkonto bei Microsoft oder GitHub. Das Settings Sync–Feature unterstützt neben einer vollständigen Übernahme der Konfiguration auch die Auswahl bestimmter Einstellungen zur selektiven Übernahme in die IDE eines anderen Arbeitsplatzes.

Binding, Events, Props reichen Werte von Eltern– an die Kind–Komponente

Um den Datenfluss zwischen JavaScript-Variablen und HTML-Elementen oder zwischen Eltern- und Kind-Komponenten in beide Richtungen zu synchronisieren, führt Svelte eine Binding-Syntax ein. Ein Label *bind:* vor der Eigenschaft eines HTML-Elements definiert eine Verknüpfung zwischen diesem und der nachfolgenden Variablen. Änderungen zwischen der HTML-Eigenschaft und der genannten Variablen werden automatisch synchronisiert.
Damit findet eine gleichzeitige Änderung von HTML-Element und Variable oder von Eltern- und Kind-Komponente statt. Svelte führt aufgrund der Änderung intern weitere Zuweisungen aus. Dieses Bindung macht die Realisierung einer eigenen Synchronisation durch die Programmierung von Events hinfällig.

Mittels Events kann eine Komponente einer anderen etwas mitteilen, ohne dass zwischen den beiden Komponenten überhaupt eine Beziehung besteht. Die Möglichkeit der Kommunikation über Events macht die Komponenten unabhängiger voneinander. Diese Kommunikation erweist sich für das Testen und die Wiederverwendbarkeit der Komponenten als vorteilhaft.

Natürlich kennt Svelte alle bereits in HTML und JavaScript vorhandenen DOM-Events. Zusätzlich führt Svelte weitere Events zwischen Kind- und Eltern-Komponenten sogenannte Component- sowie zusätzlich Lifecycle-Events ein. Für DOM-Events unterstützt Svelte die Registrierung einer Handler-Funktion direkt als Attribut im HTML-Tag mit der *on:event*-Syntax. So führt im HTML-Tag *<button*

on:click={handleClick}> Anklicken </button> ein Klick auf die Schaltfläche die Handler-Funktion handleClick aus.

Komponenten, die andere Komponenten verwenden, stehen in einer Eltern-Kind-Beziehung. Eine *import*-Anweisung macht die Kind- der Eltern-Komponente zugänglich. Mithilfe von Props/Eigenschaften kann die Eltern-Komponente der Kind-Komponente Startwerte mitgeben:

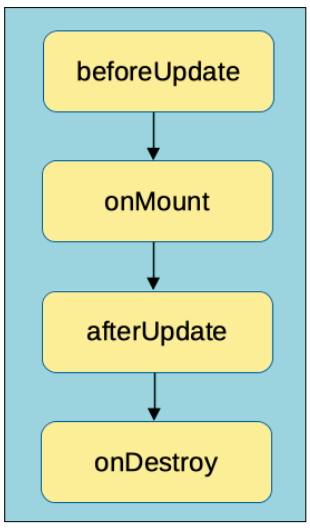
```
// Eltern-Komponente
<script>
    import Kind from './Kind.svelte';
</script>
<Kind wochentag={'Mittwoch'}/>
```

Props haben einen Namen und einen Wert. In der Kind-Komponente muss eine gleichnamige Variable mit dem Zusatz *export* deklariert sein. Bei der Erzeugung der Kind- über die Eltern-Komponente initialisiert Svelte automatisch die Variable mit dem zugehörigen Wert der übergebenen Props:

```
// Kind-Komponente: Kind.svelte
<script>
    export let wochentag;
</script>
<p>Heute ist {wochentag}</p>
```

Da von der Kind- zur Eltern-Komponente keine Referenz existiert, muss das Kind mit der Eltern-Komponente über Events kommunizieren. Diese Events tauschen Nachrichten/Ereignisse zwischen zwei Komponenten aus, daher nennt man sie in Svelte *Component-Events*.
Die Erzeugung der Events in der Kind-Komponente muss der Programmierer selbst durchführen. Dazu importiert man die Funktion *createEventDispatcher* von Svelte und ruft diese auf, um eine *dispatch*-Funktion zu erzeugen. Mittels eines Aufrufs von *dispatch()* sendet man Events/Ereignisse an andere Komponenten:

```
// Kind-Komponente: Kind.svelte
<script>
   import
   {createEventDispatcher}
   from 'svelte';
   let dispatch =
   createEventDispatcher();
   function handleClick() {
        dispatch('message',
        {
          text: 'Hallo!'
        });
    }
</script>
<button on:click={handleClick}>
    Anklicken, um Hallo zu sagen!
</button>
```



Die Lifecycle-Events einer Komponente treten immer in einer bestimmten Reihenfolge auf (weitere Untergliederung möglich) (Bild 6)

Damit eine andere Komponente den Event hört, das heißt zur Verarbeitung empfängt, muss mit der *on:event*-Syntax ein Listener registriert und eine Handler-Funktion definiert werden. Die Handler-Funktion verarbeitet in der Eltern-Komponente den Component-Event der Kind-Komponente.

```
// Eltern-Komponente
<script>
    import Kind from './Kind.svelte';
    function handleMessage(event) {
          alert(event.detail.text);
    }
</script>
<Kind on:message={handleMessage}/>
```

Events, die Svelte während der Lebensdauer einer Komponente selbst erzeugt, heißen Lifecycle-Events. Jede Komponente besitzt einen Lebenszyklus, der bei ihrer Erzeugung beginnt und durch ihr Destroy (Beseitigung aus dem DOM) endet. Seitens des Svelte-Frameworks existieren vier verschiedene Typen von Lifecycle-Events, die in einer bestimmten Reihenfolge (Bild 6) durchlaufen werden:

- *<beforeUpdate>*: vor der Aktualisierung der Komponente über das DOM löst das Svelte-Framework diesen Event aus
- *<onMount>*: dieser Event kommt zur Ausführung nachdem die Komponente zum ersten Mal über das DOM gerendert wird, oder anders ausgedrückt: sobald die Komponente dem DOM hinzugefügt wird, wird onMount ausgelöst
- *<afterUpdate>*: diesen Event erzeugt das Svelte-Framework nach der Aktualisierung der Komponente über das DOM
- *<onDestroy>*: dieser Event wird ausgelöst, sobald die Komponente zerstört oder anders ausgedrückt: aus dem DOM entfernt wird.

Svelte kennt, wie bereits beschrieben, verschiedene eigene Konstrukte, um den Aufbau des DOMs in der App zu steuern. Das heißt grundsätzlich lässt sich das DOM innerhalb einer Komponente mittels der eigenen Kontrollstrukturen von Svelte aufbauen.

Allerdings kann eine Eltern-Komponente von außen nicht das DOM der Kind-Komponente beeinflussen. Mithilfe von Slots lässt sich jedoch im DOM der Kind-Komponente ein Container erstellen, den die Eltern-Komponente mit eigenen Elementen füllen kann. Mit dieser Svelte-Technik erhält man sehr flexible Kind-Komponenten:

```
<script>
…
</script>
<!-- Kind-Komponente: Behaelter.svelte -->
```

```
<h3>
    <slot>
       <p>Dieser Text erscheint,
          wenn keiner bereitgestellt wird.</p>
    </slot>
</h3>

<h4>
    <slot name="ueberschrift" />
    <slot name="beschreibung" />
</h4>

<slot name="schluessel" />
    {#if $$slots.schluessel}
        <h6>
           Ein Slot-schluessel ist vorhanden!
        </h6>
    {/if}

<style>
…
</style>
```

Die obige Kind-Komponente *Behaelter.svelte* enthält vier verschiedene *<slot/>*-Elemente. Der erste *<slot/>* ohne Namen erscheint nur, wenn dieser Komponente *Behaelter.svelte* kein Wert für diesen *<slot/>* übergeben wurde. Die nächsten beiden *<slot/>*-Elemente erhalten einen Attributnamen *ueberschrift* und *beschreibung*. Initialisiert man diese Komponente, so kann man die beiden Slots über diesen Attributnamen ansprechen und ihnen Werte übergeben.

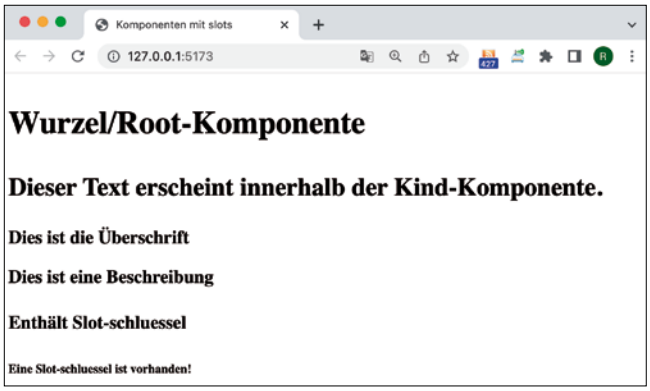
Der letzte, vierte Slot entspricht einer weiteren, eigenen Kontrollstruktur von Svelte. Sollte der Slot mit dem Attributnamen *schluessel* in der Eltern-Komponente nicht vorkommen, so bleibt er unberücksichtigt. Ansonsten wird der in *<h6> </h6>* stehende Text zusätzlich zu einem eventuell übergebenen Wert ausgegeben. Die nachfolgende Eltern-Komponente *Element.svelte* bestückt die Solts der obigen Kind-Komponente mit Werten:

```
<!-- Eltern-Komponente: Element.svelte -->
<script>
    import Behaelter from './Behaelter.svelte';
</script>

<Behaelter>
   <h3>
       Dieser Text erscheint innerhalb der K
       ind-Komponente.
   </h3>
   <p slot="ueberschrift">Dies ist die Überschrift</p>
   <p slot="beschreibung">Dies ist eine Beschreibung</p>
    <p slot="schluessel">Enthält Slot-schluessel</p>
</Behaelter>
```

Um eine Ausgabe im Webbrowser der *Element.svelte*-Komponente zu erzeugen (Bild 7), benötigt man die nachfolgende *App.svelte*-Komponente. Diese benutzt *Element.svelte* (die Eltern-Komponente), welche wiederum an die Slots der Kind-Komponente Werte weiterreicht:

```
<script>
// App.svelte
import Element from './Element.svelte'
</script>

<main>
  <h1>
    <p>Wurzel/Root-Komponente</p>
  </h1>
  <h3>
    <Element />
  </h3>
</main>
```

Svelte unterstützt sowohl Multi-Page- als auch Single-Page-Webanwendungen (Single-Page-Apps/SPAs). Eine Einzelseiten/Single-Page-App basiert auf einem einzigen HTML-Dokument, deren Inhalte dynamisch nachgeladen werden. Anstatt wie bei einer klassischen Web-App einzelne, untereinander verlinkte HTML-Dokumente vom Server nachzuladen, findet dieser Vorgang nahezu vollständig auf dem Client statt. Diese vorwiegende Ausführung der Web-App am Client reduziert zum einen die Serverlast, ermöglicht gleichzeitig auch einen gewissen Grad an Offline-Unterstützung. Um eine derartige Navigation für die Anzeige verschiedener Webseiten in einer SPA einfacher als mit einer global zu verwaltenden Zustandsvariablen zu unterstützten, hat man neue APIs geschaffen, die den Browser in die Lage versetzen, derartige Übergänge durchzuführen.



Eine Eltern-Komponente erhält über Slots Zugriff auf das DOM einer Kind-Komponente und kann dieses so nach eigenen Vorstellungen aufbauen (Bild 7)

Routing in einer Svelte-Webanwendung implementieren

Das dazu benötigte Routing zwischen den einzelnen Webseiten führt ein sogenannter Router durch. Ein JavaScript-Rou-

ter führt Buch über den Zustand der SPA und simuliert die Übergänge zwischen den Webseiten bei Änderungen in der URL. Immer wenn sich die URL der SPA ändert, bemerkt der Router dies und zeigt die mit der neuen URL verbundene View/Webseite an. Es gibt verschiedene Lösungsmöglichkeiten, das Routing in einer SPA zu implementieren.

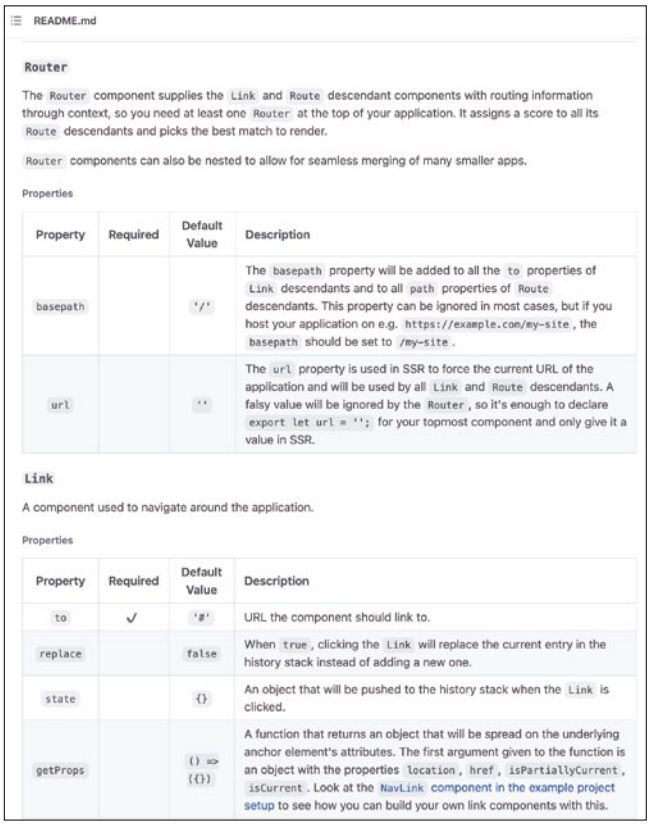
Häufig findet ein Mapping zwischen einem Route-Pfad (URL) und einem dazugehörigen Template statt. Ein derartiges Template beschreibt das für den Übergang von einer Route zu einer anderen zu generierende DOM. Bekannte Vanilla-JavaScript-Libraries für das Routing sind Navigo, page.js oder router.js. Ebenso besitzen bekannte JavaScript-Frameworks eigene Router wie Angular Router Module, React Router oder Vue Router.

Die Svelte-Community favorisiert keinen bestimmten Router

Für das reine Svelte-Framework favorisiert derzeit die Svelte-Community keinen bestimmten Router – in der Regel kommt eine der nachfolgenden Libraries zum Einsatz:

- *page.js*: ein kleiner vom Express.js-Framework inspirierter client-seitiger Router, der auch Content-Delivery-Networks (CDN) unterstützt. Offiziell bietet *page.js* einen CDN-Service für cdnjs und unpgk an. Innerhalb einer SPA kann man *page.js* über Global-Script-Tags oder über Module einsetzen
- *Sapper (Svelte app maker)*: dabei handelt es sich um ein auf Svelte aufbauendes Toolkit für die schnellere Entwicklung performanter Svelte-Apps. Es implementiert auch einige Features, die in Svelte nicht vorhanden sind, wie Routing, Server-Side-Rendering (SSR) oder Offline-Support. Sapper orientiert sich an Next.js, wurde aber zwischenzeitlich von der Svelte-Community durch SvelteKit abgelöst und aktuell nicht mehr weiterentwickelt
- *svelte-routing*: dieser Router wurde speziell für das Routing in Svelte von Emil Tholin entwickelt. Es handelt sich dabei um eine deklarative Routing-Library für Svelte mit SSR (Server-Side-Rendering). SSR verbessert die Performance für die Bereitstellung einer HTML-Seite und berücksichtigt Aspekte für SEO wie das Crawling von statischen HTML-Seiten
- *svelte-spa-router*: ein von Alessandro Segala programmierter Router, der speziell auf Svelte-SPAs ausgerichtet ist und sowohl RegExp (Regular Expressions) als auch Hash-based-Routing unterstützt. Die Idee reguläre Ausdrücke (RegExps) in das Routing einzubeziehen stammt ursprünglich von Navigo. Hash-based-Routing erlaubt wie beim Einsatz des HTML5 History-API die Verwendung der im Browser integrierten Schaltflächen wie Eine *Seite vor* (Forward), *Eine Seite zurück* (Backward) oder *Aktuelle Seite neu laden* (Reload/Refresh). Allerdings eignet sich Hash-based-Routing nicht direkt für SEO (Search-Engine-Optimization)
- *svelte-navigator*: ein Fork von svelte-routing, der von react-router und *@reach/router* inspiriert ist. Es handelt sich um einen leichtgewichtigen auf SPAs ausgerichteten Router, mit Konfigurationsoptionen und Router-Kontexts
- *svelte-router, Routify* und *svelte-page*: diese drei npm-Pakete besitzen nur eine geringe Verbreitung. Während *svelte-router* bereits die volle Funktionalität eines Routers für Svelte-SPAs abdeckt, befindet sich Routify über das Community-Projekt *+Svelte Add* noch in Entwicklung – im Unterschied dazu ist svelte-page ausschließlich auf Svelte Version 2 ausgerichtet und wurde bisher nicht auf Svelte 3 portiert.

Das svelte-routing-Package nutzt SSR, die Svelte zwar unterstützt, aus Performancegründen muss man allerdings für das HTML-DOM die sogenannte Hydration in Svelte aktivieren. In diesem Fall nimmt Svelte ein Update für alle bereits im DOM existierenden HTML-Elemente vor, ohne dabei neue



README.md

Router

The `Router` component supplies the `Link` and `Route` descendant components with routing information through context, so you need at least one `Router` at the top of your application. It assigns a score to all its `Route` descendants and picks the best match to render.

`Router` components can also be nested to allow for seamless merging of many smaller apps.

Properties

Property	Required	Default Value	Description
`basepath`		`'/'`	The `basepath` property will be added to all the `to` properties of `Link` descendants and to all `path` properties of `Route` descendants. This property can be ignored in most cases, but if you host your application on e.g. `https://example.com/my-site`, the `basepath` should be set to `/my-site`.
`url`		`''`	The `url` property is used in SSR to force the current URL of the application and will be used by all `Link` and `Route` descendants. A falsy value will be ignored by the `Router`, so it's enough to declare `export let url = '';` for your topmost component and only give it a value in SSR.

Link

A component used to navigate around the application.

Properties

Property	Required	Default Value	Description
`to`	✓	`'#'`	URL the component should link to.
`replace`		`false`	When `true`, clicking the `Link` will replace the current entry in the history stack instead of adding a new one.
`state`		`{}`	An object that will be pushed to the history stack when the `Link` is clicked.
`getProps`		`() => ({})`	A function that returns an object that will be spread on the underlying anchor element's attributes. The first argument given to the function is an object with the properties `location`, `href`, `isPartiallyCurrent`, `isCurrent`. Look at the `NavLink` component in the example project setup to see how you can build your own link components with this.

Das README.md im GitHub-Repository von svelte-routing gibt ausführliche Anleitungen zum Einsatz der Routing-Library (Bild 8)

zu erzeugen. Dieser Prozess der Überprüfung nennt sich Hydration; erst wenn diese vollständig durchgeführt wurde, findet im Browser das DOM-Update statt. Standardmäßig ist Hydration in Svelte ausgeschaltet – bereits bei der Initialisierung der Haupt-Komponente durch die *main.js*-Datei ist deren Attribut *hydrate* auf *false* gesetzt.

Die Hydration selbst führt der *Svelte-Compiler* durch, diese findet für eine Svelte-Komponente nur statt, wenn deren *hydrate*-Attribut auf *true* gesetzt ist. Kommt SSR zum Einsatz, so erreicht man Hydration für die betroffenen HTML-Elemente ebenfalls indem deren *hydrate*-Attribut den Wert *true* enthält. Dann erkennt der Svelte-Compiler, welche Elemente er für die DOM-Aktualisierung berücksichtigen muss. Das Setzen des *hydrate*-Attribut direkt bei der Komponente über *component.$set(hydrate:true)* eignet sich, da die Aktuali- ▶

sierung des DOM nicht synchron stattfindet. Vielmehr muss während der Build-Time ein Prerendering stattfinden, so dass zur Laufzeit ein SSR möglich wird.

Nach der Installation des *svelte-routing*-Package: *npm i svelte-routing* im Projekt, setzt man in der *main.js*-Datei das *hydrate*-Property auf *true*. Danach legt man im Projekt-Ordner, die seitens *svelte-routing* für SSR benötige *server.js*-Datei an, diese findet man im GitHub-Repository im README.md. Anschließend muss die Haupt-Komponente *App.svelte* den Router konfigurieren. Dafür benötigt man einen Import der zugehörigen Objekte (Router, Link, Route) mit den zugehörigen HTML-Seiten für das Routing. Der Router ermittelt die beste Übereinstimmung für eine übergebene URL. Link führt die Navigation innerhalb der *Svelte-App* durch. Die Route-Komponente nimmt das Rendering entsprechend der von Router bestimmten Seite vor. Zum Schluss muss die zugehörige URL exportiert werden.

Ein passendes Beispiel für diese Anpassung der Haupt-Komponente enthält wiederum das README.md im GitHub-Repository von *svelte-routing*. Dort sind auch die Attribute/Eigenschaften der Routing-Objekte, deren Funktionsweise und ihre Vorbelegungen beschrieben (Bild 8). Um für das SSR die Hydrate-Option zu aktivieren, muss in der *rollup.config.js*-Datei die Compiler-Option *hydratable: true* hinzugefügt werden. Zusätzlich muss man in der *package.json*-Datei im *script*-Bereich dem *sirv*-Webserver bei der public-Option der Parameter *-single* übergeben: "*start*": "*sirv public -single*". Damit teilt man *sirv* mit, dass eine SPA mit der *index.html*-Datei zu erzeugen ist. Entsprechend ist die Vorgehensweise bei den anderen Bundlern (*snowpack, vite: vite-plugin-ssr, vite-plugin-svelte*).

Routing für Svelte-SPAs mittels der svelte-spa-router-Library

Das npm-Paket *svelte-spa-router* eignet sich für alle modernen Browser wie Chrome, Edge, Firefox oder Safari. Jede normale Svelte-Komponente mit sämtlichen Svelte-Konstrukten kann das Ziel einer Route darstellen. Im Zentrum der Routing-Library steht ein Dictionary-Objekt, dessen Schlüssel-Feld entspricht dem Pfad (die URL) und der zugehörige Wert die über eine Route anzuzeigenden Svelte-Komponente. Das JavaScript-Dictionary definiert man als *const routes* in eine eigene *routes.js*-Datei ab, welche die Datenstruktur auch exportiert. Die *routes.js*-Datei legt man im Projektordner an, damit sie von dort allgemein zugänglich ist.

Zu Beginn importiert man in *routes.js* alle zu verwendenden Svelte-Komponenten. Es hat sich eingebürgert, diese in einem eigenen Unterverzeichnis zum Beispiel *routes* oder *components* im Projektordner abzulegen. So lautet eine Import-Anweisung beispielsweise *import Home form './routes/Home.svelte'*. Nach dem Import der Komponenten deklariert man das *routes*-Dictionary. Dabei muss man darauf achten, dass *svelte-spa-router* für eine Route als Default-Einstellung immer den Eintrag aus dem Dictionary verwendet, auf den die gesuchte URL zuerst passt. Der letzte Eintrag sollte in der *routes.js*-Datei immer '*': *NotFound* mit der NotFound-Komponente (Listing 3) sein.

Für das Matching der Routes in der *routes.js*-Datei sind auch reguläre Ausdrücke erlaubt, da das *svelte-spa-router*-Package diese unterstützt. Um die Route-Komponente in einer Svelte-App einzusetzen, müssen die zugehörigen Objekte, Komponenten und Datenstrukturen in die Hauptkomponente importiert werden (Listing 4). Dazu gehört der Router und die *link*-Komponente des *svelte-spa-router*-Pakets und das *routes*-Dictionary der *routes.js*-Datei. Nach dem Start der SPA zeigt diese anfangs die im *main*-Tag definierte HTML-Elemente an. Anschließend führt die Einbettung von *<Router {routes}/>* die Router-Komponente aus; sie entscheidet, welche der vordefinierten URLs als aktive zu rendern ist.

Listing 3: NotFound.svelte

```
// Sollte kein Pfad zur aktuelle URL passen,
// so zeigt svelte-spa-router diese
// Svelte-Komponente an

<h2>Nicht gefunden!</h2>

<p>Die gewünschte URL existiert nicht!</p>
```

Listing 4: Routing mit svelte-spa-router-Package

```
<script>
  import Router, { link } from "svelte-spa-router";
  import { routes } from "./routes.js";
</script>

<main>
    <h1> …
…
    <Router {routes}/>
</main>

<style>
   …
</style>
```

Stores erleichtern die Kommunikation zwischen den Komponenten

Stores in Svelte stellen ein einfaches, leicht zu benutzendes und direkt im Framework integriertes, Hilfsmittel dar (vergleichbar den React Hooks in React). Mit einem Store lassen sich Daten (beispielsweise über den Zustand einer App vergleichbar in Angular mit @ngrx/store, React mit Redux oder Vue mit Vuex) speichern und verwalten, um schnell Informationen zwischen den Komponenten auszutauschen. Dabei müssen die Komponenten nicht auf die Props/Eigenschaften von anderen Komponenten zugreifen. Auch ist kein zusätzlicher Mechanismus wie der Austausch von Nachrichten/Events oder ein genau zu spezifizierendes Bindung mit wei-

teren Aktionen notwendig. Stores eigenen sich besonders für Komponenten, die untereinander in keinerlei Beziehung stehen und dennoch Daten austauschen wollen.

Ein Store entspricht einem Speicherbereich mit Methoden wie *subscribe* oder *update*, um auf ihn zuzugreifen oder mittels *set* zu bearbeiten. Alle im Store enthaltenen Daten stehen global in der App zur Verfügung; das heißt eine App kann von jeder beliebigen Stelle auf die Daten/Funktionen eines Stores zugreifen. Die Daten/Funktionen müssen nur von der Komponente, die auf sie zugreifen will, importiert werden. Ein Store unterstützt Reaktivität, das bedeutet er informiert automatisch alle Komponenten, welche seine Methoden benutzen und aktualisiert eventuell die betroffenen Daten in der Komponente. Die Ablage eines Stores erfolgt in einer eigenen JavaScript-Datei mit der Erweiterung *.js*. Der Namen dieser Datei sollte sich am semantischen Inhalt ihrer Daten oder Informationen orientieren.

Das Svelte-Framework kennt vier verschiedene Typen von Stores, die man auch innerhalb einer einzigen *.js*-Datei deklarieren kann. Den Typ des Stores legt eine *import { <store_type>,... } from 'svelte/store'*-Anweisung fest. Alle dieser vier Store-Typen stellen verschiedene Methoden entsprechend ihrem Einsatzgebiet automatisch für den Entwickler bereit:

- *writable*: ein Writable-Store erlaubt allen Komponenten, dessen Inhalt zu ändern. In der Praxis trifft man diesen Store-Typ am häufigsten an
- *readable*: ein Readable-Store ändert seine Daten selbst, er erlaubt anderen Komponenten nicht, seine Daten zu ändern
- *derived*: dieser Store-Typ leitet, wie bereits der Name derived im Englischen zum Ausdruck bringt, die Werte seiner Daten aus anderen Stores ab. Seine Werte werden automatisch aktualisiert, wenn die zugrundeliegenden Werte sich ändern
- *custom*: ein derartiger Store besitzt den größten Freiheitsgrad; ein Custom-Store erlaubt alles software-technisch denkbare, dazu stellt dieser Store-Typ ein passendes API zur Verfügung, auch Custom-API genannt. Diesen Typ von Store benötigt der Programmierer recht selten.

Bindet man die Preis.svelte-Komponente in die über Vite generierte App.svelte-Komponente mit <Preis /> ein, so erfolgt deren Ausgabe in der Svelte-App (Bild 9)

Innerhalb der IDE legt man eine JavaScript-Datei *stores.js* an; diese deklariert einen Writable- und einen Derived-Store:

```
import { writable, derived } from 'svelte/store';

const steuer = 0.19

export const nettoPreis = writable(0);

export const bruttoPreis =
derived(nettoPreis, $nettoPreis =>
                    $nettoPreis *(1+steuer));
```

Der Wirtable-Store besitzt die Variable *nettoPreis*, die mit Null initialisiert wird. Der Derived-Store bestimmt den Bruttopreis einer Ware, indem er den Nettopreis mit dem Umsatzsteuersatz von 19 % multipliziert und zum Nettopreis addiert. Eine *derived()*-Funktion in Svelte besitzt zwei Argumente: die Store-Variable von dem der Derived-Store seinen neuen Wert abgeleitet und eine sogenannte *callback*-Funktion, welche den abgeleitet Wert berechnet oder bestimmt.

Anhand des $-Zeichens bei der *callback*-Funktion teilt dieser Quellcode dem Svelte-Framework mit, eine Subscription zu diesem Wert des Stores (*nettoPreis*) vorzunehmen. Damit deklariert der obigen Quellcode die beiden Stores vollständig, so dass sie sich in einer Svelte-Komponente verwenden lassen. Die nachfolgende Komponente *Preis.svelte* nimmt einen Import der Store-Variablen vor:

```
<script>
    import { nettoPreis, bruttoPreis }
    from './stores.js';
</script>

<main>
    <h3>
        <label> Nettopreis:
                <input type="number"
                bind:value={$nettoPreis}>
        </label>
        Bruttopreis: {$bruttoPreis}
    </h3>
</main>
```

Über ein *<input>*-HTML-Element liest die *Preis.svelte*-Komponente einen nummerischen Wert ein, den der Benutzer eingibt. Dabei verbindet das *<input>*-HTML-Element diesen eingegebenen Wert mit dem Writable-Store *nettoPreis* aus der store.js-Datei. Um die Variable eines Stores im HTML-Quellcode anzusprechen, muss man diese in Svelte über *{$<store-variable>}* einbetten. Da der Writable-Store *nettoPreis* nachdem der Benutzer einen Wert in das Eingabefeld eingegeben hat, seinen Wert ändert, passt auch der Deri- ▶

ved-Store *bruttoPreis* seinen Wert automatisch an. Der durch diese automatische Anpassung des Derived-Store ermittelte Wert gibt die Referenz auf ihn *{$bruttoPreis}* neben der Zeichenkette *Bruttopreis* über die Preis.svelte-Komponente aus. Die Preis.svelte-Komponente nimmt also eine Initialisierung des Writeable-Store vor, die anschließend eine automatische Initialisierung des Derived-Store auslöst (Bild 9).

Formular-Verarbeitung in einer Svelte-App mithilfe spezieller Form-Libraries

In fast jeder Web-App nehmen Formulare für die Verarbeitung von Daten eine wichtige Rolle ein. Formulare dienen dazu, Daten erstmals zu erfassen, zu speichern, aus dem Backend zu lesen oder zu ändern. Um die Korrektheit und Vollständigkeit der Daten in den Formularen zu gewährleisten, kommen Validierungen innerhalb der Web-App zum Einsatz. Natürlich lassen sich Formulare und deren Funktionalität direkt mit den Mitteln von Svelte realisieren. Dazu setzt man die gängigen HTML-Elemente mit den üblichen JavaScript-Features, CSS-Gestaltung und der Binding-Technik von Svelte ein. Diese Vorgehensweise führt allerdings bald zu redudantem Quellcode, vor allem wenn man öfters auf Formulare innerhalb der Web-App zurückgreifen muss.

Wesentlich produktiver arbeitet man mit einer der für Svelte verfügbaren Form-Libraries. Derartige Bibliotheken reduzieren wesentlich den dafür erforderlichen Quellcode. Zu den bekannteren für Svelte verfügbaren Formular-Libraries zählen:

- *sveltejs-forms*: mit dieser Formular-Bibliothek programmiert man Formulare auf deklarative Art und Weise. Innerhalb einer <*Form* />-Komponente nimmt man die Deklaration des Formulars vor. Zusätzliche Komponenten wie *Input, Select* oder *Choice* reduzieren erheblich den Quellcode. Eine Definition eigener sogenannter Custom-Komponenten wird unterstützt. Für eine optionale auf dem Datenschema basierende Validierung kommt Yup zum Einsatz. Mithilfe Slot-Props greift man auf bestimmte Eigenschaften, Objekte oder Funktionen der Formulare zu
- *Svelte forms lib*: die Realisierung dieser Formular-Bibliothek orientiert sich an Formik (einer Formular-Bibliothek für React). Damit ähnelt das API der <*Form* />-Komponente von Svelte forms lib dem Formik-API, das als leicht erlernbar gilt. Neben einem eigenen Validation-Feature lässt sich auch Yup einsetzen. Mittels eines Form-Arrays kann man dynamische Formulare durch Hinzufügen und Entfernen von Feldern oder Objekten realisieren. Für den <*style* />-Block stehen eigene CSS-Klassen für die Formular-Elemente zur Verfügung
- *Svelte Formly*: dieser Formular-Generator erzeugt anhand von Formular-Deklarationen aus Feld-Attribut-Listen dynamisch zur Laufzeit Formulare. Für die Prüfung der Eingabewerte in den Feldern verfügt diese Bibliothek über sogenannte Core Rules. Zusätzlich lassen sich eigene Feldtypen und Validation-Rules mit Wiederverwendung der Core Rules definieren. Da keine eigenen CSS zum Einsatz kommen, kann man jedes CSS-Framework integrieren. Neben Svelte lässt sich Svelte Formly auch mit SvelteKit (ehemals Sapper) und Routify verwenden
- <*Sveltik* />: diese Bibliothek stellt ein Objektmodell von Formular-Komponenten zur Verfügung (angelehnt an Formik). Damit ähnelt das API weitestgehend dem Formik-API; seit Svelte 3 nutzt <*Sveltik* /> die *let:*-Direktive, welche zu einer wesentliche Reduktion des Quellcode führte. Insbesondere, da die *let:*-Direktive auch für alle Kind-Komponenten zur Verfügung steht
- *Svelte Use Form*: diese auf HTML-Elemente aufbauende Formular-Library benötigt für die Realisierung eines Formulars recht wenig Quellcode. Die Funktionalität kapselt sich um das jeweilige Formular-Objekt, anstatt sie auf den ganzen Quellcode zu verteilen – was die Lesbarkeit erhöht und die Wartung erleichtert. Die Bibliothek enthält sofort einsetzbare Prüfroutinen; eigene implementiert man als JavaScript-Funktionen – zusätzlich lässt sich *Validate.js* nutzen
- *Felte*: diese Bibliothek eignet sich für Svelte, Solid und React/Preact. Sie basiert auf HTML5 und bietet Reaktivität für Formulare. Der Aufbau eines Formulars lässt sich dynamisch zur Laufzeit steuern. Eine extender-Funktion erweitert der Funktionalität: Sie kommt beim Aufbau des Formulars, einer Formular-Aktion oder bei jeder Änderung des Formulars zum Einsatz. Felte unterstützt über eigene npm-Packages die Integration einer Validation-Bibliothek wie yup, zod, superstruct oder vest.

Die Web-App Vitest UI zeigt die Durchführung der Tests über verschiedene Registerreiter an (Bild 10)

Aufgrund der Integration von Svelte in das Front-End-Tooling von Vite bietet sich für das Testen von Svelte-Apps der Einsatz des Test-Frameworks *Vitest* an. Vitest basiert auf dem Tooling von Vite, so dass man innerhalb eines Svelte-Projekts alle vorhandenen Vite-Features nutzen kann.

Somit steht direkt auch Hot Module Reload (HMR) von Vite für Test-Runs zur Verfügung. Während Entwicklung, Build und Test nutzt ein Testrunner von Vitest über Vite-Tooling dieselbe Konfiguration wie die Svelte-App in der *vite.config.js*/ts-Datei. Damit entfällt der mit anderen Testing-Frameworks häufig zusätzliche Aufwand für Einrichten und Pflege eines separaten Testing-Workflows.

Für Unit/Component- und Integration-Testing im JavaScript/TypeScript-Umfeld kommt oft Jest zum Einsatz. Deshalb haben die Vitest-Entwickler auf Kompatibilität zu

Listing 5: Testfälle für Counter.svelte-Komponente

```
// Counter.spec.ts

import { tick } from 'svelte';
import { describe, expect, it } from 'vitest';
import Counter from '../src/lib/Counter.svelte';

let host: HTMLElement

describe('Counter component', function () {
  it('Eine Instanz von Counter anlegen', function () {
    host = document.createElement('div');
    host.setAttribute('id', 'host');
    document.body.appendChild(host);
    const instance = new Counter({ target: host });
    // Prüfen, ob Counter-Instanz vorhanden
    expect(instance).toBeTruthy();
  });

  it('Erste Anzeige von Counter durchführen',
  function () {
    host = document.createElement('div');
    host.setAttribute('id', 'host')
    document.body.appendChild(host);
    new Counter({ target: host });
    // Testergebniss prüfen
    expect(host.innerHTML).toContain('count is 0');
  });

  it('Beim Anklicken der Schaltfläche count
    aktualisieren', async function ()
    {
    const host = document.createElement('div');
    document.body.appendChild(host);
    new Counter({ target: host });
    expect(host.innerHTML).toContain('count is 0');
    const btn = host.getElementsByTagName('button')
    [0];
    btn.click();
    await tick();
    // Prüfen, ob der korrekte Text auf Schaltfläche
    // erscheint
    expect(host.innerHTML).toContain('count is 1');
  });
});
```

Jest geachtet. Kenntnisse zu Jest führen zu einem schnelleren Einstieg in Vitest und erleichtern auch eine Migration von Jest auf das Test-Framework Vitest. Um Vitest in ein Svelte-Projekt zu integrieren, muss zunächst geprüft werden, ob die benötigten Versionen von Vite (>=2.7.10) und Node (>=14) vorhanden sind. Während *node -v* die Versionsnummer des aktuell verwendeten Node.js-Systems ausgibt, zeigt *npm list vite* die im Svelte-Projekt installierte Versionsnummer von Vite an. Zur Messung der Code-Überdeckung (Code-Coverage) durch die Testfälle, hat Vitest das npm-Package c8 – Native V8 Code-Coverage integriert.

Zusätzlich besitzt Vitest Features wie Snapshots und Mocking. Snapshots vergleichen aktuelle Testausgaben mit bereits vorliegenden, zu erwartenden Ergebnissen. Dabei unterstützt Vitest für Snapshots auch den Vergleich von Images und Serialisierern für die Speicherung von Daten. Mocks simulieren durch Platzhalter noch nicht implementierte Schnittstellen. Durch Mocking lassen sich Objekte testen, die noch keine Implementierung besitzen.

Die Entwickler von Vitest empfehlen für das DOM-Mocking den Einsatz eines der npm-Pakete happy-dom oder jsdom. Vitest besitzt auch ein eigenes CLI (Call Level Interface), um Tests im Hintergrund auszuführen. Bei Einsatz des CLI ist das während der Programmierung eingeschaltete HMR zu deaktivieren. Das CLI kann mehrere Tests durch einen Aufruf ausführen; so führt der Befehl *vitest kunde* alle Testfälle aus, die im Dateinamen die Zeichenkette kunde enthalten.

Ergänzend verfügt Vitest über eine eigene Web-App *Vitest UI*, um Testfälle interaktiv auszuführen. Die zugehörige Software ist nicht im Vitest-Paket enthalten, sondern muss separat als optionales Package installiert werden. Der Terminalbefehl *vitest --ui* startet die Web-App und öffnet die zugehörige URL: *http://localhost:51204/__vitest__/* (Bild 10).

Um das Verhalten von Vite und Vitest bei der Ausführung der Svelte-App einzustellen, legt man im Projekt eine *vitest.config.ts*-Datei an. Alle Einträge dieser Datei besitzen seitens des Test-Frameworks gegenüber den Vorgaben in *vite.config.js/ts* eine höhere Priorität. Bei einem CLI-Aufruf liest die *--config*-Option über den Pfad mit Angabe des Dateinamens darin enthaltene Vorgaben ein.

Testfälle für eine Svelte-App in der IDE mittels Vitest ausführen

Ausgangspunkt stellt die seitens Vite über das Kommando *npm init vite@latest* generierte Svelte-App mit dem Projektnamen demo-vitest dar. Um die für Vitest benötigten Packages zu installieren, empfiehlt es sich, die als npm-Package realisierte Web-App *npm-gui* einzusetzen. Nach einer globalen Installation des npm-Package npm-gui startet der Befehl *npm-gui* das benutzerfreundliche npm-Tool.

Um Vitest mit npm-gui in einem Svelte-Projekt zu installieren, öffnet man deren *package.json*, klickt auf die linke Schaltfläche *+SEARCH/ADD*, gibt *vitest* in das Suchfeld ein und klickt anschließend in der Spalte *install* auf die *DEV*-Schaltfläche.

Dies führt den Befehl *npm install -D vitest* durch, nimmt also eine Installation von Vitest im Svelte-Projekt als *devDependency* vor. Entsprechend installiert man die npm-Packages @vitest/ui für die Web-App vitest-ui, c8 und jsdom. ▶

Für die Ausführung von Tests mit Vitest über den *npm run*-Befehl sollte man im *scripts*-Bereich der *package.json* die nachfolgenden Eintrage aufnehmen:

```
"test": "vitest",
"test:ui": "vitest --ui",
"coverage": "vitest run --coverage",
```

Für den Einsatz von Vitest in VS Code oder VSCodium installiert man die Extension Vitest for VSCode, diese steht auch für IntelliJ/WebStorm zur Verfügung. Neben der Ausführung von Testfällen unterstützt diese Extension auch einen Debug-Modus und die Anzeige des Console-Outputs. Nach erfolgter Installation erscheint in VS Code/VSCodium in der linken Toolleiste das Symbol-Icon eines Erlenmeyerkolbens, der einen kleinen Füllstand andeutet. Anschließend legt man im Projektordner *demo-vitest* die Datei *vitest.config.ts* mit folgenden Definitionen an:

```
import { defineConfig } from 'vitest/config'
import { svelte } from '@sveltejs/vite-plugin-svelte'
export default defineConfig({
  plugins: [
    svelte({ hot: !process.env.VITEST }),
  ],
  test: {
    globals: true,
    environment: 'jsdom', },
})
```

Die Testfälle sollen das korrekte Verhalten der mit Svelte programmierte Komponente *Counter.svelte* prüfen. Diese Komponente zeigt eine Schaltfläche mit dem Text *count is {count}* an. Wobei anfangs *count* auf Null steht und damit der Text *count is 0* als Ausgabe erfolgt. Jeder Klick auf die Schaltfläche erhöht *{count}* um eins, gibt also die Anzahl ihrer Klicks aus. Mit Vitest kann man mehrere Testfälle als Suite in einer Datei ablegen, diese erhalten gemäß Konvention die Datei-Endung *spec.ts*. Es empfiehlt sich, die Testdateien getrennt vom Quellcode in einem eigenen Unterverzeichnis mit dem Namen *test* abzulegen. Die Testfälle der Komponente Counter.svelte enthält die Datei *Counter.spec.ts* (Listing 5).

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE                    zsh
macmini@macMinis-Mini demo-vitest % npm run coverage

> demo-vitest@0.0.0 coverage
> vitest run --coverage

 RUN  v0.21.1 /Users/macmini/development/svelte/demo-vitest

 ✓ test/Counter.spec.ts (3)

Test Files  1 passed (1)
     Tests  3 passed (3)
  Start at  15:27:04
  Duration  2.38s (setup 0ms, collect 446ms, tests 14ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 Counter.s|     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
macmini@macMinis-Mini demo-vitest %
```

Die Abdeckung des Quellcodes der Svelte-App durch die Testfälle (Testabdeckung) ermittelt nach erfolgreicher Testdurchführung das npm-Package c8 (Bild 11)

Um die Testfälle für die Komponente in VS Code/VSCodium auszuführen, klickt man in der linken Toolleiste auf das Icon mit dem Erlenmeyerkolben. Es erscheint im IDE-Fenster die Toolbar *TESTING* mit verschiedenen Bedienelementen. Ein Klick auf das zweite Bedienelement *Run Tests* führt alle im Projekt vorhandenen Vitest-Testdateien aus. Um gezielt eine Testsuite (hier: *Counter component*) auszuführen, wählt man diese unterhalb des Eintrags *test/Counter.spec.ts* aus und klickt auf das erste Icon-Symbol *Run Test*. Bei Auswahl einer Testsuite zeigt VS Code/VSCodium alle darin enthaltenen Testfälle an.

Nach Auswahl eines Testfalls der Testsuite führt Vitest durch Klick auf *Run Test* die zugehörige Testdatei mit dem Testfall immer komplett aus. Um direkt den Quellcode des Testfalls zu bearbeiten, markiert man ihn und klickt das dritte Icon *Go to Test* an. Darauf öffnet die IDE die zugehörige Testdatei mit dem Quellcode und positioniert den Cursor direkt auf die Deklaration des Testfalls. Die Code-Überdeckung des Quellcodes durch die Testfälle bestimmt der Befehl *npm run coverage* im Projektordner. Dabei führt Vitest alle Testdateien mit sämtlichen Testfällen aus; c8 ermittelt die Code-Überdeckung der Svelte-App durch die Testfälle und zeigt das Ergebnis in einer Liste an (Bild 11).

Im Rahmen der Entwicklung gilt es, regelmäßig das Ergebnis des Build-Prozesses zu testen. Die Zielrichtung einer Svelte-App stellt eine SPA (Single-Page-App) dar. Deren Betrieb findet in der Regel über einen Webserver oder in einer Cloud-Plattform über einen Docker-Container statt. In beiden Fällen erfolgt der Build einer Svelte-App mittels des Terminalbefehls *npm run build*. Die Ergebnisse des Builds für das Projekt legt die Vite-Toolchain im Ordner *dist* ab. Für die Bereitstellung der Svelte-App in einem Docker-Container beschreibt ein Dockerfile die Erzeugung eines Container-Images.

In VS Code/VSCodium benötigt man die Extension Docker von Microsoft, um in der IDE mithilfe des Dockerfile das Container-Image für die Svelte-App zu erzeugen. Zusätzlich

muss Docker Desktop auf dem Entwicklungsrechner installiert und vor dem ersten Aufruf der IDE bereits gestartet sein. Alle für den Build des Docker-Containers in VS Code/VSCodium nicht zu berücksichtigen Verzeichnisse des Projekts legt man im Projektordner in der Datei *.dockerignore* ab.

Im Dockerfile (Listing 6) befinden sich die Anweisungen aus der die Docker-Extension über die IDE das Container-Image erzeugt. Zu Beginn lädt Docker ein Base-Image node und vergibt für dieses den Namen *build*. In diesem Image legt der *mkdir*-Befehl den Ordner */app* an und erklärt diesen zum Arbeitsverzeichnis. Anschließend kopiert COPY alle Ordner und Dateien aus dem Projekt ohne die in *.dockerignore* angegebenen Verzeichnisse in das Arbeitsverzeichnis. Die besondere Bedeutung der beiden Dateien *package.json* und *package-lock.json* soll der separate *COPY*-Befehl hervorheben. Da der *npm ci*-Befehl schneller als das übliche *npm install*-Kommando arbeitet, kommt dieser für den Image-Build zum Einsatz. Nun erfolgt für die Zielumgebung mittels *npm run build* ein Build der Svelte-App.

Nach der Einrichtung des nginx-Webservers kopiert der *COPY --from*-Befehl die Ergebnisse des Builds vom /app/dist-Verzeichnis in den *DocumentRoot*-Ordner von nginx. Markiert man in der Dateiliste der IDE das Dockerfile, so startet in dessen Kontextmenü der Befehl *Build Image…* die Generierung dieses Docker-Images. Anschließend steht in Docker Desktop das erzeugte *demovtest*-Image zur Ausführung zur Verfügung. Nach Klick auf die *Run*-Schaltfläche in Docker Desktop vergibt man nach Auswahl der *Optional Settings* für *Local Host* einen Port, zum Beispiel *8080*. Anschließend startet das Image in einem Container; über dessen URL *localhost:8080* erreicht man die laufende Svelte-App.

Um schnell und ohne viel Aufwand die Funktionsfähigkeit einer verteilten Svelte-App zu testen, empfiehlt es sich, eine der beiden Hosting-Plattformen surge oder Vercel einzusetzen. Beide Cloud-Plattformen unterstützen das Hosting von Websites und Single-Page-Apps (SPAs) mit automatischer Skalierung. Zudem fallen so gut wie keine Aufgaben für deren Konfiguration an. Auch gibt es für surge und Vercel unterschiedliche Preismodelle, die mit einem kostenlosen Account beginnen. Am einfachsten arbeitet man mit surge und Vercel über deren CLI, das als npm-Package realisiert ist. Das Terminal-Kommando *npm install --global [surge | vercel]* installiert deren CLI.

Nach erfolgter Installation des CLI von *surge*, öffnet man ein Terminalfenster und wechselt in den *dist*-Ordner, der den Build-Code der Svelte-App enthält. Ruft man jetzt mit *surge* das CLI auf, beginnt der Deployment-Dialog. Sollte noch kein eigener Account auf *serge.sh* existieren, so legt das CLI nach Eingabe von *email* und *password* automatisch ein neues Benutzerkonto an. Bestätigt man anschließend das über *project* angezeigte Projektverzeichnis und akzeptiert den gewählten Domain-Namen, so führt *surge* das Deployment durch. Über den beim Deployment erzeugten Domain-Namen öffnet ein Webbrowser die programmierte Svelte-App.

Listing 6: Dockerfile für Image-Build der Svelte-App

```
FROM node AS build
# Arbeitsverzeichnis anlegen
RUN mkdir -p /app
WORKDIR /app
COPY . .
# Auch die package-lock.json kopieren
COPY package*.json ./
# Notwendige npm-Packages installieren
RUN npm ci
# Build für Svelte-App durchfühen
RUN npm run build
# Webserver nginx einrichten
FROM nginx
COPY --from=build /app/dist /usr/share/nginx/html
```

Verwaltung der Domain-Namen

Der Befehl *surge --domain <Domain-Name>* speichert diesen persistent für alle nachfolgende CLI-Befehle ab. Da man in der Regel aber nicht nur ein, sondern mehrere Projekte besitzt, legt man besser eine Textdatei mit dem Namen CNAME an und trägt dort die verschiedenen Domain-Namen ein. So muss man sich den Domain-Namen nicht merken und das CLI liest diesen bei einem erneuten Deployment aus der CNAME-Datei ein. Die Einträge in der CNAME-Datei entsprechen dem Canonical Name Resource Record (CNAME RR) des Domain-Name-Systems (DNS). Dies ermöglicht in *surge* auch den Einsatz eines eigenen Domain-Namen.

Nach Einrichtung eines Kontos für Vercel und einer globalen Installation des CLI benötigt VS Code/VSCodium für den Zugriff auf Vercel eine Autorisierung. Diese erfolgt beim ersten Aufruf von *vercel* im Svelte-Projekt. Nach erfolgreichem Deployment gibt das CLI den Domain-Namen der Svelte-App aus (beispielsweise https://demovitest.vercel.app/). Ein Klick darauf startet die Web-App im Browser, um Tests des Deployments durchzuführen. Den Domain-Namen zeigt die Vercel-Site auch bei den *Project Settings* im Bereich *Domains* an.

Vercel unterstützt für verschiedene Git-Repository-Systeme (GitHub, GitLab, Bitbucket) ein automatisches Deployment. Sobald Entwickler durchgeführte Änderungen über die Entwicklungsumgebung an das Git-Repository übergeben, führt Vercel selbständig ein Deployment durch. Dazu benötigt Vercel allerdings eine einmalige Autorisierung für die Zugriffe auf das Git-Repository. Bei der Durchführung des Deployments zeichnet Vercel außerdem automatisch Logs im Bereich *Overview* > *Deployment status* > *Building* auf. ■

Dr. Manfred Simon
unterstützt Projekte in den Arbeitsgebieten: Analyse, Spezifikation, Entwicklung, Programmierung, Test & Debugging in Cloud-, Mobile- und Web-Umgebungen inklusive deren System-Management.
web_mobile_developers@gmx.eu