

Architektur: Client Schnatter



WWI 21SEB
Projektrealisierung
Semester 5&6
DHBW Mannheim

Studenten:
Marcel Bulling, Lisa Reiß-Park, Dennis Bollian
und Nora Klemp

Architektur Beschreibung

Grundlegend wurde unsere Webseite mit dem Svelte Framework entwickelt. Dazu haben wir die Skelten.UI Library für das Styling und die grundlegenden Komponenten verwendet. Zudem verwenden wir für unser Custom-Styling Tailwind.css.

Als Packagemanager haben wir uns für PNPM entschieden, was dazu führt das alle im Folgenden beschriebenen Node befehle die normalerweise mit „npm“ starten in unserem Fall mit „pnpm“ starten. Um keine wichtigen Updates für einzelne Packages zu übersehen und immer auf dem neusten Stand zu sein, haben wir zusätzlich noch renovate zu unserem Github hinzugefügt, um immer über Dependency Updates informiert zu werden.

Die Tests sind mit Hilfe des Playwright Frameworks erstellt worden (dazu mehr beim Testing in den QS).



Abb. 1 Übersicht der Architektur

Zu Svelte:

Svelte stellt zusätzlich neben der Clientseite Svelte auch die Möglichkeit mit Hilfe von SvelteKit eine Serverseite zu implementieren. Aufgrund der niedrigen Komplexität der Datenverarbeitung in diesem Projekt, haben wir uns gezielt dazu entschieden nicht mit SvelteKit, und der damit verbundenen Serverseite, zu arbeiten. Wir haben ausschließlich mit der Clientseite gearbeitet und halten auch dort alle Daten vor.

Projekt Struktur:

In unserer Projektstruktur gibt es zum einen den „src“-Ordner, der wiederum einmal einen Ordner enthält, in dem alle unsere Komponenten hinterlegt sind, die Teil des Frontends sind. Ebenfalls befindet sich hier ein Ordner „lib“ in dem zum einen alle Typen hinterlegt sind, die wir in diesem Projekt verwendet haben und zum anderen alle Utils (Funktionen mit Requests und ausgelagerte Logik). Die Datei wurde unter dem Punkt Store genauer beschrieben.

In dem Ordner Routes befinden sich die eigentlichen Routen unseres Projektes. Bei Svelte handelt es sich dabei um eine Single-Side-Application auf der die unterschiedlichen Routen gerendert werden.

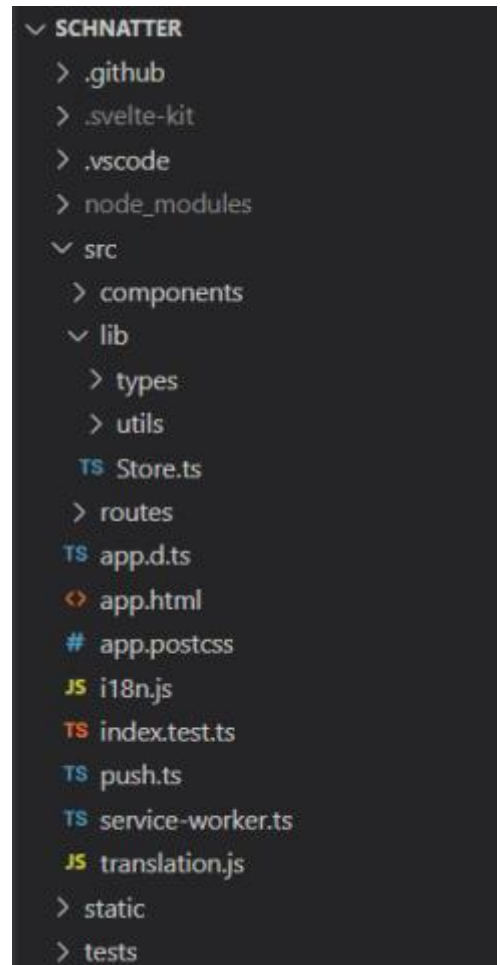
Die Ordernamen und -strukturen bestimmen hierbei in Svelte die Pfade in der Domain. Die Dateien selbst müssen dann immer „+page.svelte“ heißen.

Bei der „i18n“ handelt es sich, um die Konfigurationsdatei zu verwenden von Übersetzungstexten, die in der translation.js gepflegt werden.

Statische Bilder oder andere statische Daten sind im „static“-Ordner zu finden.

Die „service-worker.ts“ beinhaltet die Implementierung des Service-Workers.

Ansonsten sind im Hauptverzeichnis noch einige Konfigurationsdateien enthalten und noch ein Ordner für die Testes

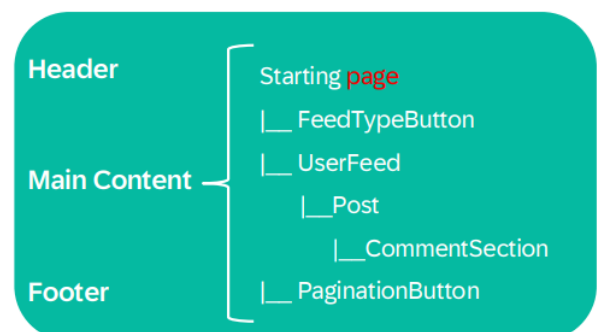


Komponenten Architektur

Wie bereits beschrieben, handelt es sich bei Svelte um eine Single-Side-Application, bei der sogenannte Pages auf eine HTML-Seite gerendert werden. Diese Pages können zusätzlich noch beliebig viele Subkomponenten haben.

Um jedoch eine grundlegende Struktur mit Header und Footer zu erzeugen gibt es die Möglichkeit eine „+layout.svelte“ anzulegen.

Die Abbildungen zeigt zunächst grundlegend unser Layout. Mit dem Header und Footer, welche auf jeder Seite zu sehen sind. In dem Main Content wird der Inhalt der Pages erstellt. Hier am Beispiel der Feed-Page, mit den einzelnen verwendeten Komponenten und Bestandteilen. Die Feed-Page entspricht unserer Home-Page.



Stores:

Stores sind in Svelte dazu da, um Variablen zu definieren, die im localStorage gespeichert werden. Unterschieden werden hier Writables, die nach dem Neu-Laden der Seite wieder mit dem initialen Wert, der in der Store-Datei definiert wurde, überschrieben wird.

Persisted Variablen, bleiben auch nach einen Neu-Laden erhalten. In der Abbildung sind einige Beispiele zu sehen, in welchen Bereichen wir den Store genutzt haben.

User

- **RegisterUsername** (writable)
- **Token** (persisted)
- **RefreshToken** (persisted)
- **GlobalUsername** (persisted)

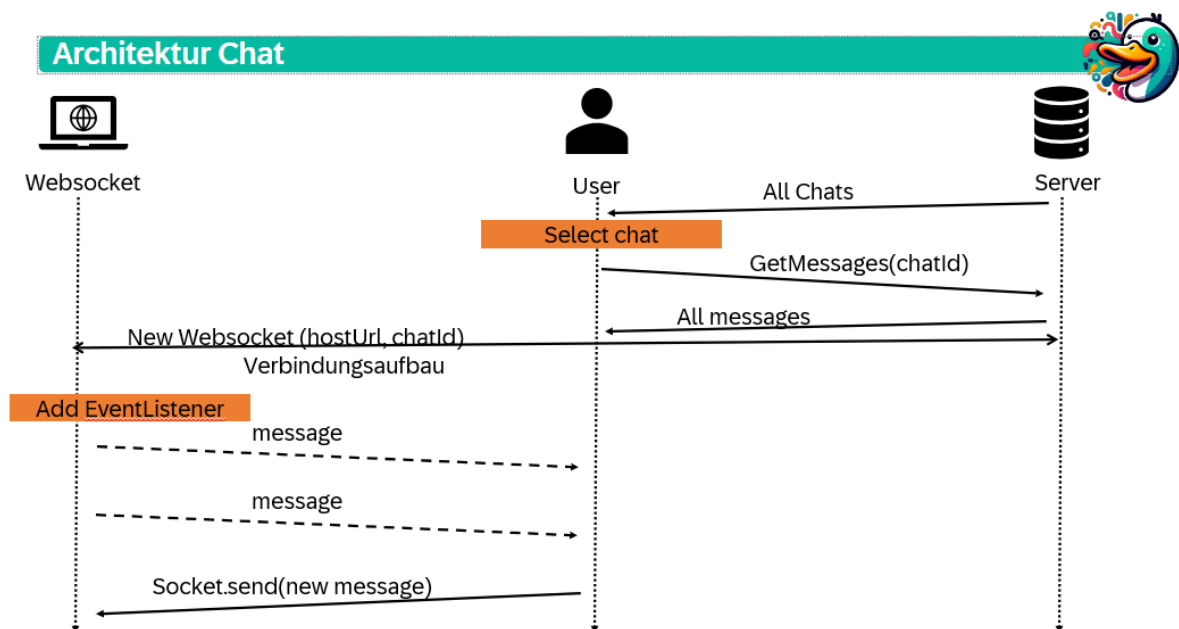
Chat

- **ChatIdNewChat** (persisted)

Notifications

- **NotificationCount** (persisted)
- **NotificationList** (persisted)

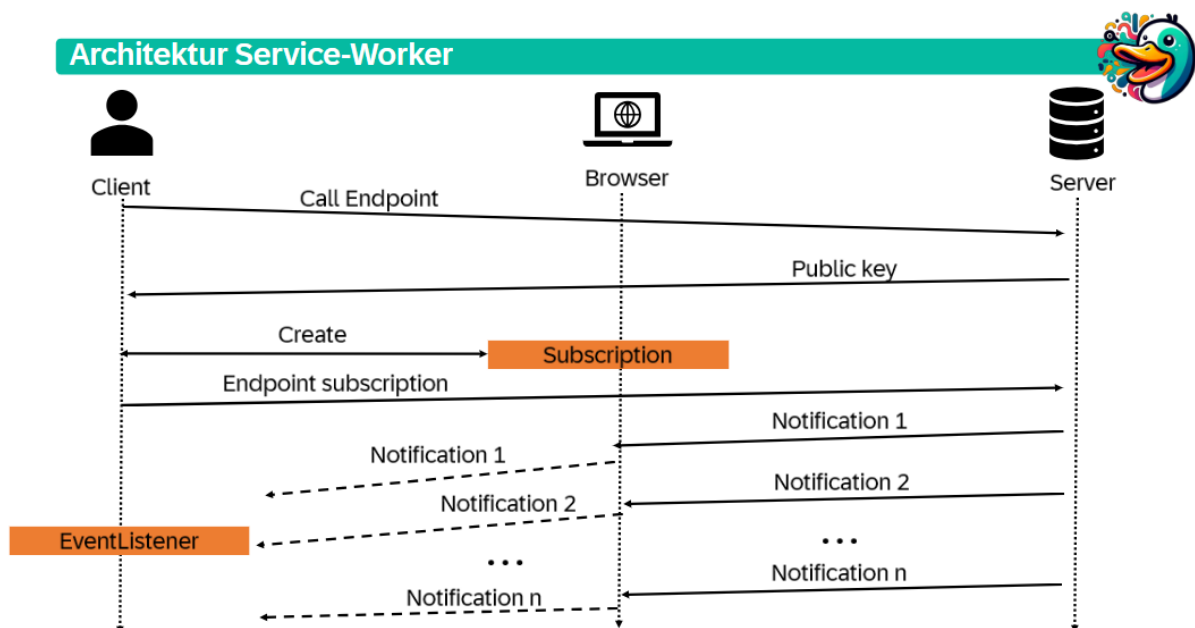
Erläuterung Websockets:



Wenn ein Chat geöffnet wird, werden zuerst einmal alle bereits geschriebenen Nachrichten vom Server geladen.

Danach wird der Websocket erstellt, die Verbindung zum Backend hergestellt und der EventListener hinzugefügt. Dieser erhält dann alle weiteren Nachrichten vom Server und gibt diese an den Client weiter. Wenn eine Nachricht geschrieben wird, wird diese auch über den Websocket an das Backend weitergeleitet.

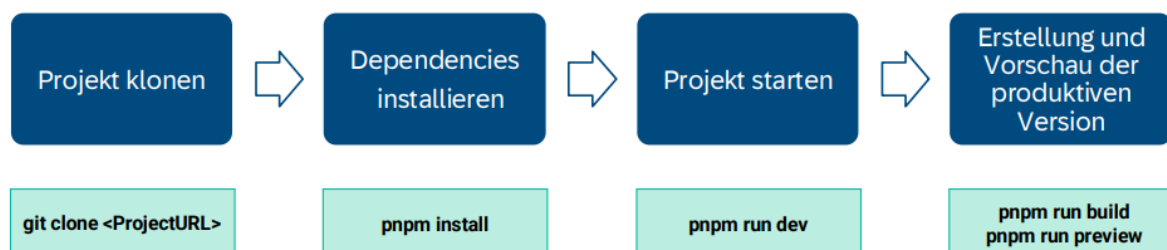
Erläuterung Service-Worker:



Bei dem Service-Worker stellt uns das Backend einen Public-Key bereit den wir durch einen Endpunkt abfragen können. Danach erstellen wir im Browser eine Subscription die einen Endpunkt und die notwendigen Keys enthält, an den das Backend die Benachrichtigungen schicken kann. Anschließend sendet das Backend alle Benachrichtigungen an den Browser. Und der Client kann sich die Daten daraus über entsprechende EventListener auslesen, um wiederum als Event an die Stellen zu übergeben, an denen sie gebraucht werden. In unserem Fall gibt es in der NavBar einen weiteren EventListener, der die Daten ausliest, die im UI angezeigt werden.

Build-Prozess:

Die folgende Abbildung beschreibt einmal den lokalen Build-Prozess. Voraussetzung hierfür sind mindestens Node.js V18 und pnpm muss installiert sein.



Hier der Link zu weitem Infos diesbezüglich.

<https://svelte.dev/blog/svelte-for-new-developers>

Einmal-Deploy:

In unserem Falle läuft die Anwendung auf einem Ubuntu-Server auf dem Node installiert ist. Wenn man die Webseite nur einmal deployen möchte. Reicht es den Ordner, der im











Build der Anwendung erzeugt wurde auf den Server zu kopieren und dort mit dem entsprechenden Befehl auszuführen. Natürlich muss noch die Domain richtig konfiguriert werden, damit sie auch unter dem gewollten Domain-Name erreichbar ist. Wir verwenden hierfür zusätzlich noch NGINX auf dem Server. (https://nginx.org/en/docs/http/configuring_https_servers.html)

Unser HTTPS Zertifikat beziehen wir über Certbot

Deploy über Pipeline:

Hierfür muss zum einen auf dem Ubuntu Server ein Service erstellt werden über den man den Produktive-Build starten kann. Dies geschieht mit Hilfe einer .service und einer .sh Datei. Zudem muss eine SSH-Verbindung auf den Server ermöglicht werden.

Zusätzlich müssen noch auf Github in dem Repository die entsprechenden Repository Secrets in den Einstellungen geändert werden.

Repository secrets			New repository secret	
Name ↕			Last updated	
IP			last month	 
PROJECT_PATH			5 months ago	 
SSH_PRIV_KEY			6 months ago	 
SSH_PUB_KEY			6 months ago	 
USER_IP			last month	 

Die IP enthält entweder die IP-Adresse des Servers oder bei dynamischen IP-Adressen den Domain-Namen, unter dem der Server erreichbar ist.

Der „project_path“ enthält den Projekt-Pfad, wo das Projekt auf dem Server liegt.

Der Private und Public Key der SSH-Verbindung muss auf dem Server erzeugt werden und hier entsprechend eingefügt werden.

User_IP enthält den Ubuntu-User, auf dem die Anwendung läuft im Format <Username>@<IP-Adresse/Domain-Name>

Die Pipeline stellt via SSH dann eine Verbindung zum Server her, kopiert den durch den Buildbefehl erstellten Ordner auf den Server und startet dort den Service neu der die Anwendung startet.

Anmerkung:

Wenn keine Server-Konfiguration vorgenommen werden soll, gibt es für Studenten mit dem Studenten Abo von Github die Möglichkeit das Ganze auf der Plattform „Vercel“ zu deployen. Hier muss nur das Repository angegeben werden und eine Konfiguration getätigt werden. Es bedarf somit keiner weiteren Kenntnisse über das Deployen von Anwendungen auf Servern.