Scientific Software Center

IWR Interdisciplinary Center for Scientific Computing

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

svelte-easydb-detail-view Knowledge Transfer Meeting

Dr. Dominic Kempf, SSC

ssc.iwr.uni-heidelberg.de

Svelte - ein komponenten-basiertes Frontend Framework

- Svelte ist ein modernes Frontend Framework (vgl. React, vue.js etc.)
- Ein Abstraktionslevel über HTML+JS: Eigene Sprache
  - Compiler Toolchain übersetzt in HTML+JS: vite
- Zentrales Element ist die “Komponente”
  - Verquickung von Markup und Logik
  - Wenn man die gesamte Anwendung in Svelte entwickelt, ist sie eine Komponente.
  - Komponenten werden als zentrale Abstraktion zur Vermeidung von Codeduplizierung benutzt
  - Komponenten werden in Svelte entwickelt, können aber nach JS transpiliert werden
  - Dementsprechend kann man sie auch aus NPM-Paketen laden
  - Für Standardaufgaben verwendet man meist eine große Library z.B. flowbite-svelte
- Es ist sehr üblich eigene Komponenten zu entwickeln

Scientific Software Center

Wie sieht eine Komponente `Beispiel` aus

- Schreibe neue Datei `Beispiel.svelte`
- Diese enthält 2 Blöcke:
  - Einen JS/TS Skript Teil mit Logik
    - Umfasst von `<script>...</script>`
    - Exportierte Variablen werden Properties, welche (reaktiv) von außen gesetzt werden.
  - Einen Markup Teil
    - Verwendet andere Komponenten
    - Kann auch HTML Tags enthalten
    - Variablen aus dem Logikteil können hier direkt verwendet werden mit `{variable}`
    - Spezielle Direktiven erlauben Interaktion des Markups mit den Daten z.B.
      - `{#if condition}` .. `{/if}`
      - `{#each list as item}` .. `{/each}`
      - `etc.`

Scientific Software Center

Verwendung von eigenen Komponenten

- Importieren
  - Wenn lokal: `import Beispiel from "./Beispiel.svelte"`
  - Wenn Package: `import { Beispiel } from "beispiel-package"`
  - Im lokalen Fall wird vite den Transpilationsprozess entsprechend managen
- Verwenden im Markup-Teil: `<Beispiel bla=42 />`

Scientific Software Center

Komponenten in svelte-easydb-detail-view

- Die Detailansicht selbst ist Komponente `EasydbDetailView`
  - Teilweise in `DetailViewImpl` implementiert (für Popover)
- Jeder DataType hat seine eigene Komponente (built-in und custom)
- Jeder Splitter hat seine eigene Komponente (built-in und custom)
- Jeder AssetViewer hat seine eigene Komponente (im Moment: Image, Video)
- Die obigen haben eine “Dispatch” Komponente
  - Importiert alle verfügbaren Komponenten
  - Mappt die in EasyDB verwendeten Strings auf die entsprechenden Komponenten
- Zusätzliche Komponenten für Teile der Detailansicht: `DetailControls`, `AssetViewer`, `TitleDisplay`, `HierarchyViewer` etc.
- Besondere Rolle (später mehr): `RecursiveEasyDBDetailView`
- Für Testzwecke: `App`

Scientific Software Center

EasyDBDetailView Komponente - Properties

- `systemid`: Welches Objekt gerade angezeigt wird
- `appLanguage:` Systemsprache z.B. `de-DE`
- `dataLanguages` ausgewählte Datensprachen z.B. `["de-DE", "en-US"]`
- `easydbInstance` : URL der EasyDB Instanz;
- `mask`: Maske welche für das Rendering verwendet wird
- `masksToRender`: Welche Masken nativ (anstatt als Popover) angezeigt werden
- `token`: Ein Access Token für die EasyDB Instanz (wenn privat)
- `seededInitialId`: (Ent)kopplung verschiedener Detailansichtinstanzen

Datengetriebene Generierung

- Folgende Daten werden von der EasyDB Instanz über die API gezogen:
  - Maskendefinition
  - Datenbankschemadefinition
  - L10N Information
- Da dies sehr lange dauern kann, kann dies auch zur Compilezeit gemacht werden und die Daten mit dem Bundle verteilt werden
- Die Generierung der Komponente beinhaltet dann ein koordiniertes Iterieren über die Maske und die Objektdaten und das Einfügen der entsprechenden Komponenten in die Detailansicht.
- Fast jede Komponente bekommt daher diese Properties
  - `data`: Der relevante Teil des Daten JSON
  - `field`: Der relevante Teil der Maske
  - `table`: um welchen Table im Datenschema es sich handelt

Gängige Erweiterungen

- Neuer Custom Datentyp
  - Komponente in `./src/components/fields` implementieren
  - In `./src/components/logic/FieldDispatch.svelte` hinzufügen
- Neuer Custom Splitter
  - Komponente in `./src/components/splitter` implementieren
  - In `./src/components/splitter/splitterMapping.js` hinzufügen
  - Dort auch angeben: Ist dies ein Start/Ende Splitter oder nicht?
- Neuer Asset Typ
  - Komponente in `./src/components/viewer` implementieren
  - In `./src/components/logic/AssetDispatch.svelte` hinzufügen

Scientific Software Center

Styling

- Innerhalb der Komponenten werden Tailwind CSS Klassen benutzt
- Tailwind ist ein Abstraktionslayer über CSS, der verschiedenste Klassen bereitstellt
- An wenigen Stellen wird auch Custom CSS direkt - per Import - eingebunden, siehe z.B. `./src/components/logic/bracket.css`
- Tailwind Themes können in `tailwind.config.cjs` konfiguriert werden
- Das Hinzufügen weiterer eigener CSS-Klassen in der Codebase ist möglich
- Problematisch wird dies beim Bundlen (siehe gleich)

Scientific Software Center

Bundling

- Wenn die Zielanwendung keine Svelte App ist, bundlen wir den Code in eine Custom Web Component (standardisiert von W3C).
- Vorher möglich: `node src/generate.js --instance=<URL>`
- Bundling mit `npm run bundle`
- Prozess wird von Rollup gesteuert und in `rollup.config.js` konfiguriert
  - Eine Reihe von Rollup Plugins wird benötigt um dies korrekt zu tun
  - Die Notwendigkeit dynamischer Imports um einen zyklischen Import zu durchbrechen verkompliziert diesen Prozess
- Das Handling von CSS ist hier mit das Schwierigste
- Benutzung der Custom Web Component mit:
  - `<script src="easydb.js"></script>`
  - `<easydb-detail-view systemid="6493"/>`

Bundling - CSS Issues

- Custom Web Components verwenden üblicherweise Shadow DOM
  - Gekapseltes Styling
- Wir müssen daher unser CSS in dieses Shadow DOM injecten:
  - Siehe wenig intuitiver Beispiel-Code in `./bundle/index.html`
- Auf gleiche Art und Weise könnte man auch zusätzliches CSS hineinbringen
- Dies ist keine besonders schöne Lösung, aber die einzige, die wir gefunden haben.