Beispielaufgabe: Vier Gewinnt

In dieser Beispielaufgabe entwickeln wir eine einfache Webanwendung mit **SvelteKit**, die das klassische Spiel „Vier Gewinnt“ als lokale Zwei-Spieler-Variante umsetzt. Diese Version enthält ausschließlich Frontend-Logik und basiert auf dem **Static Adapter** von SvelteKit. Sie bildet die Grundlage für die spätere Erweiterung um eine Multiplayer-Funktion mit Datenbankanbindung im nächsten Kapitel.

Zielsetzung

Ziel dieser Aufgabe ist es, ein funktionierendes Spiel zu bauen, das komplett im Browser läuft, keine Verbindung zu einem Server benötigt und als statische Website bereitgestellt werden kann. Die Umsetzung soll die Stärken von **Svelte** bei der Entwicklung interaktiver Komponenten demonstrieren und gleichzeitig ein übersichtliches Projekt mit nachvollziehbarer Struktur liefern.

Projektgrundlage

Wir starten mit der Initialisierung eines neuen SvelteKit-Projekts. Dafür verwenden wir das SvelteKit CLI.

```
npm create svelte@latest vier-gewinnt-frontend
```

Im Setup wählen wir das SvelteKit minimal Setup, aktivieren TypeScript, ESLint, Prettier, Tailwind CSS und wir installieren wir den Static Adapter. Eine genauere Dokumentation des CLI findest du in dem Artikel Projektsetup mit SvelteKit.

Um die Anwendung während der Entwicklung im Browser zu testen, starten wir den Entwicklungsserver:

```
npm run dev --open
```

Projektstruktur

Der Einstiegspunkt für unsere Anwendung befindet sich in der Datei `src/routes/+page.svelte`.

In dieser Komponente schreiben wir sowohl die Spiellogik als auch das HTML-Markup und das grundlegende Styling direkt in einer Datei. Für diese Frontend-Version verzichten wir bewusst auf eine Aufteilung in mehrere Komponenten – das reduziert vorerst die Komplexität.

Im Projektverzeichnis befinden sich zahlreiche Dateien und Ordner, von denen für diese Aufgabe nur wenige eine Rolle spielen:

```
vier-gewinnt-frontend/
├── src/
│   ├── app.css
│   └── routes/
│       ├── +layout.svelte     ← gemeinsames Layout aller Seiten
│       ├── +layout.ts         ← legt fest, dass das Projekt prerenderbar ist
│       └── +page.svelte       ← Hauptdatei mit Spiellogik und UI
├── svelte.config.js
├── package.json
└── ...
```

Die Rolle von +layout.svelte

Die Datei `+layout.svelte` ist bei SvelteKit der Standardort für gemeinsame Layouts, die auf allen Seiten einer Route (hier: `/`) angezeigt werden sollen. In unserem Fall nutzen wir sie, um globale Styles (`app.css`) einzubinden und das allgemeine Layout der Seite zu definieren – z. B. zentrierte Inhalte, Container-Breiten oder Hintergrundfarben.

```
<script lang="ts">
	import '../app.css';
	let { children } = $props();
</script>

<div class="container mx-auto p-8">
	{@render children()}
</div>
```

Spiellogik

In der Datei (`src/routes/+page.svelte`) befindet sich unsere Haupt-Anwendung. Hier arbeiten wir mit mehreren zentralen Konzepten, die typisch für die Entwicklung mit **SvelteKit** sind. Anhand des Spiels lernen wir grundlegende Elemente kennen, mit denen sich interaktive Webanwendungen effizient umsetzen lassen.

Komponentenbasiertes Markup

Der HTML-Teil innerhalb der Datei ist direkt mit dem TypeScript-Code verknüpft. Das Markup beschreibt dabei nicht nur die Oberfläche, sondern ist **reaktiv** an die zugrunde liegende Logik gebunden – Änderungen im Zustand führen unmittelbar zu visuellen Updates.

Reaktive Zustände mit $state()

SvelteKit 5 bringt mit `$state()` ein vereinfachtes Modell für lokale Zustände innerhalb von Komponenten. In unserem Beispiel werden darüber folgende Variablen verwaltet:

- das aktuelle Spielfeld (grid)
- der aktive Spieler (currentPlayer)
- ein möglicher Gewinner (winner)

Wenn sich diese Werte ändern, wird das DOM automatisch aktualisiert – ganz ohne manuelle DOM-Manipulation oder explizite „set“-Funktionen.

each-Blöcke für dynamisches Rendering

Zur Darstellung des Spielfelds nutzen wir den each-Block von Svelte. Damit lassen sich Arrays direkt im Template durchlaufen. In unserem Fall verwenden wir verschachtelte each-Blöcke, um die zweidimensionale Grid-Struktur darzustellen:

```
{#each grid as row}
  {#each row as cell}
    <!-- einzelne Zelle -->
  {/each}
{/each}
```

Diese Technik ist in Svelte besonders elegant, da sie mit minimalem Code dynamische und komplexe Strukturen erlaubt.

Bedingte Anzeige mit if

Der if-Block ermöglicht es, Inhalte abhängig vom Zustand darzustellen. Beispielsweise zeigen wir den Gewinner nur an, wenn das Spiel beendet ist:

```
{#if winner}
  <p>Spieler {winner} hat gewonnen!</p>
{:else}
  <p>Am Zug: Spieler {currentPlayer}</p>
{/if}
```

Dateibasierte Routen

Ein weiterer wichtiger Aspekt von SvelteKit ist das **Dateisystem-basierte Routing**. Jede Datei im Ordner `src/routes` entspricht automatisch einer URL. In unserem Fall:

- src/routes/+page.svelte → /
- src/routes/+layout.svelte → gemeinsames Layout für alle Unterseiten

Dieses Routing-Prinzip reduziert die Komplexität im Vergleich zu klassischen Router-Konfigurationen erheblich.

Empfehlung: Code ansehen & selbst ausprobieren

Ich empfehle, den Code einmal vollständig durchzugehen oder lokal auszuführen, um diese Konzepte selbst auszuprobieren und besser zu verstehen.

□□ Zum vollständigen Code auf GitLab

Build & Deploy

Beim Einsatz des Static Adapters müssen wir SvelteKit explizit mitteilen, dass Seiten **statisch generiert** werden dürfen. Das geschieht über eine Datei `+layout.ts`.

```
vier-gewinnt-frontend/
├── src/
│   ├── app.css
│   └── routes/
│       ├── +layout.svelte
│       ├── +layout.ts        ← muss hier erstellt werden
│       └── +page.svelte
├── svelte.config.js
├── package.json
└── ...
```

In der in der `+layout.ts` müssen wir export `const prerender = true` setzen:

```
// src/routes/+layout.ts
export const prerender = true;
```

Ohne diese Angabe bleibt SvelteKit im „SSR-Modus“ und erzeugt keine statischen HTML-Dateien beim Build. Für die Frontend-only-Version ist das jedoch notwendig.

Jetzt kann das Projekt gebaut werden:

```
npm run build
```

Der HTML-Code befindet sich danach im Ordner `build` und kann direkt auf einen Webserver geladen werden.

Revision #4
Created 1 April 2025 06:58:43 by Marius Klein
Updated 23 April 2025 15:39:03 by Marius Klein