Svelte als Frontend-Technologie

Svelte ist kein klassisches Framework – es ist ein eleganter Kompiler für das Frontend. Statt virtuellen DOM zu jonglieren, schreibt Svelte reaktiven Code direkt in optimiertes JavaScript. Das ist schnell, leichtgewichtig und so intuitiv, dass man fast vergisst, wie komplex moderne Webentwicklung eigentlich ist.

In diesem Kapitel werden die Stärken und Schwächen von Svelte 5 aufgezeigt.

- Svelte in a Nutshell
- Projektsetup mit SvelteKit
- Grundkonzepte von Svelte 5
- Styling & UX mit Tailwind CSS
- Beispielaufgabe: Vier Gewinnt
- Exkurs: 3D-Anwendungen im Handumdrehen mit Threlte

Svelte in a Nutshell

Svelte ist ein modernes Frontend-Framework, das sich durch seinen innovativen, compilerbasierten Ansatz von anderen Bibliotheken wie React oder Vue abhebt. Es verspricht eine elegante, minimalistische und leistungsstarke Möglichkeit, Webanwendungen zu erstellen. Dabei gehen sowohl die Entwickler als auch die Community mit Begeisterung und kritischer Auseinandersetzung an die Sache heran.

Entwicklerperspektive und Community-Erfahrungen

Die Entwickler hinter Svelte haben bewusst einen Weg gewählt, der sich von etablierten Frameworks wie React oder Vue abhebt. Statt auf komplexe Laufzeit-Mechanismen zu setzen, erfolgt die Optimierung bereits im Kompilierungsprozess – ein Ansatz, der sich als besonders performant erweist. Viele Entwickler schätzen an Svelte vor allem:

- **Intuitive und elegante Syntax:**
  Die enge Verknüpfung von HTML, CSS und JavaScript in einer Komponente sorgt für übersichtlichen Code, der sich schnell lesen und warten lässt.
- **Minimale Boilerplate:**
  Die integrierte Reaktivität reduziert den Code auf das Wesentliche, was insbesondere bei der Entwicklung von komplexen Anwendungen ein echter Gewinn ist.
- **Exzellente Performance:**
  Durch die Voraboptimierung entfällt der typische Overhead, den man bei Frameworks mit Virtual DOM häufig vorfindet.

Ein einfaches Beispiel einer Svelte-Komponente illustriert dies:

```
<script>
  let count = $state(0);
</script>

<button onclick={() => count++}>
  Klicks: {count}
</button>
```

Hier zeigt sich, wie nahtlos Logik, Markup und Styling zusammengeführt werden können – ganz ohne das typische „boilerplate“ Gedöns, das man aus anderen Frameworks kennt.

Kritische Stimmen und Herausforderungen

Trotz aller Vorteile gibt es auch kritische Stimmen: Der Wegfall des Virtual DOM und der Paradigmenwechsel hin zu einer Kompilierungszeit-Optimierung können anfangs ungewohnt sein. Ein weiterer Nachteil, den ich persönlich erlebt habe, ist die mangelnde Vertrautheit von Sprachmodellen (wie etwa ChatGPT) mit der Svelte 5 Syntax.

Svelte vs. React/Vue – Der Unterschied

Während React und Vue weitgehend auf virtuelle DOM-Differenzierung und Laufzeitoptimierung setzen, geht Svelte einen anderen Weg. Die Unterschiede lassen sich wie folgt zusammenfassen:

- **Kompilierungsansatz**:
  Svelte kompiliert die Komponenten bereits zur Build-Zeit in hochoptimierten JavaScript-Code, wodurch der Overhead zur Laufzeit minimiert wird.
- **Reaktivität**:
  Anstatt eine eigene State-Management-Bibliothek oder umfangreiche API-Aufrufe zu benötigen, ist Reaktivität bei Svelte direkt in die Sprache integriert. Das macht den Code oft kürzer und verständlicher.
- **Syntax und Integration**:
  Svelte kombiniert HTML, CSS und JavaScript in einer einzigen Komponente, was den Entwicklungsprozess vereinfacht und die Wartung erleichtert. Bei React oder Vue sind diese Aspekte oft strikter voneinander getrennt.

Warum ich jedem einen Blick auf Svelte empfehle und für wen es richtig interessant ist

Svelte übt auf mich als begeisterten Webentwickler und Frontend-Enthusiasten, der selbst regelmäßig Websites baut, eine besondere Anziehungskraft aus. Es ist eine ausgezeichnete Wahl

für alle, die Freude an moderner Webentwicklung haben und nach einer eleganten, flexiblen Lösung suchen, die nicht an ein bestimmtes Framework-Ökosystem gebunden ist.

Flexibilität und Framework-Agnostik:

Svelte überzeugt insbesondere durch seine ausgeprägte Framework-Agnostik. Diese Eigenschaft macht es zum perfekten Tool, um Frontend-Bibliotheken zu bauen. Im Vergleich zu React und Vue, die eigene Lösungen anbieten, erweist sich Svelte in dieser Hinsicht als äußerst flexibel und leichtgewichtig.

Moderne Toolchain

In Kombination mit Tools wie Tailwind CSS und Shadcn-Svelte lassen sich elegante und funktionale Komponentenbibliotheken entwickeln. Diese Kombination aus modernem Styling und flexibler Komponentenarchitektur ermöglicht die Erstellung visuell ansprechender und gleichzeitig performanter Websites, Webapps und Frontend-Bibliotheken.

Neben Svelte selbst lohnen sich aus meiner Sicht auch angrenzende Systeme, die den Entwicklungsprozess bereichern:

- **Svelte Kit:** Das offizielle Meta-Framework für Svelte, das serverseitiges Rendering und Dateibasiertes Routing ermöglicht.
- **Tailwind CSS:** Ein modernes Utility-first CSS Framework, das es erlaubt, schnell und konsistent zu stylen.
- **Shadcn-Svelte:** Eine Komponentensammlung, die den Aufbau eigener, eleganter Komponentenbibliotheken erleichtert.
- **Threlte:** Für alle, die 3D-Visualisierungen lieben, bietet Threlte.js eine spannende Möglichkeit, Three.js in Svelte-Projekten zu nutzen.
- **Strapi (Headless CMS):** Ideal, wenn du ein flexibles Backend benötigst, das sich nahtlos in Svelte-basierte Projekte integrieren lässt.

Projektsetup mit SvelteKit

SvelteKit ist das offizielle Meta-Framework rund um Svelte. Es bringt alles mit, was moderne Webentwicklung braucht: Dateibasiertes Routing, SSR, API-Routen, Layouts, und eine cleane DX. In diesem Artikel wird Schritt für Schritt erklärt, wie man ein neues Projekt mit SvelteKit aufsetzt. Dabei werden die Optionen der SvelteKit CLI beleuchtet und sinnvolle Add-ons wie **TailwindCSS**, **Prettier**, **ESLint** sowie der passende **Adapter** für das Deployment ausgewählt.

Voraussetzungen

- Node.js (empfohlen: aktuelle LTS-Version)
- Ein Terminal (z. B. iTerm2, VS Code Terminal)
- Paketmanager, Vorzugsweise NPM

Projekt erstellen

Initialisiert wird das Projekt mit folgendem Befehl:

```
npx sv create my-sveltekit-project
```

`my-sveltekit-project` kann dabei durch einen beliebigen Projektnahmen ersetzt werden. `npx` muss ggf. je nach verwendetem Package-Manager angepasst werden.

Bei erstmaliger Ausführung wird das CLI-Paket automatisch installiert. Bestätigung erfolgt mit y.

```
Need to install the following packages:
sv@0.8.1
Ok to proceed? (y) y
```

SvelteKit CLI: Optionen im Überblick

Nun wird das Projekt interaktiv konfiguriert.

1. Template-Auswahl

```
Which template would you like?
❯ SvelteKit minimal (barebones scaffolding for your new app)
  SvelteKit demo
  Svelte library
```

- **SvelteKit minimal**: Reduziertes Grundgerüst für ein eigenes Projekt ohne Beispielcode.
- **SvelteKit demo**: Enthält Beispielseiten und eine Beispielnavigation.
- **Svelte library**: Template zur Entwicklung wiederverwendbarer Svelte-Komponenten.

Wir wählen für diese Demonstration SvelteKit minimal.

2. TypeScript-Unterstützung

```
Add type checking with TypeScript?
❯ Yes, using TypeScript syntax
  Yes, using JavaScript with JSDoc comments
  No
```

Empfohlen wird die Verwendung der TypeScript-Syntax, um statische Typüberprüfung und IntelliSense zu nutzen.

3. Erweiterungen im Projektsetup

Nach der Projekterstellung fragt das CLI nach zusätzlichen Tools und Bibliotheken, die dem Projekt hinzugefügt werden sollen. Die Auswahl erfolgt interaktiv über die Leertaste (Mehrfachauswahl möglich). Die Optionen sind:

```
What would you like to add to your project?
☐ prettier          (formatter – https://prettier.io)
☐ eslint            (linter – https://eslint.org)
☐ vitest            (unit testing – https://vitest.dev)
☐ playwright        (e2e testing – https://playwright.dev)
☐ tailwindcss       (CSS utility framework – https://tailwindcss.com)
☐ sveltekit-adapter (adapter for deployment – https://kit.svelte.dev/docs#adapters)
☐ drizzle           (ORM – https://orm.drizzle.team/)
☐ lucia             (auth – https://lucia-auth.com)
☐ mdsvex            (Markdown + Svelte – https://mdsvex.pngwn.io/)
```

❒ paraglide (i18n – https://paraglide.dev/)

❒ storybook (component explorer – https://storybook.js.org)

Erweiterung	Beschreibung	Einsatzzweck
Prettier	Automatisches Codeformatierungs-Tool. Definiert einheitlichen Stil für z. B. Einrückung, Semikolons.	Formatierung & Clean Code
ESLint	Linter zur statischen Analyse von JavaScript/TypeScript-Code. Findet potenzielle Fehler & Stilprobleme.	Codequalität und Fehlervorbeugung
Vitest	Schnelles Unit-Testing-Framework, optimiert für Vite und moderne Frontends.	Komponententests, Logiktests
Playwright	Framework für End-to-End-Tests. Ermöglicht UI-Tests mit echten Browserinstanzen.	Testen von Userflows & Accessibility
TailwindCSS	Utility-first CSS-Framework für schnelles und responsives Styling.	Styling über Utility-Klassen
SvelteKit Adapter	Bindeglied zwischen dem Framework und der Zielplattform (z. B. statisches HTML, SSR, Vercel etc.).	Deployment-Anpassung
Drizzle	TypeScript-ORM für SQL-Datenbanken mit gutem DX.	Datenbankzugriff (PostgreSQL etc.)
Lucia	Authentifizierungs-Framework mit Fokus auf Einfachheit und Sicherheit.	Login-Mechanismen, Zugriffskontrolle
mdsvex	Markdown-Präprozessor für Svelte-Komponenten. Kombination von Markdown und Svelte möglich.	Content Management, Dokusysteme
Paraglide	Internationalisierungs-Tool mit Compile-Time-Optimierung.	Mehrsprachigkeit (i18n)
Storybook	Tool zur Visualisierung und Dokumentation einzelner UI-Komponenten.	Komponentenkatalog & Dokumentation

In diesem Beispiel wählen wir: prettier, eslint, tailwindcss, sveltekit-adapter

4. TailwindCSS Plugins

Wir haben uns im Schritt zuvor entschieden, **TailwindCSS** in das Projekt einzubinden. TailwindCSS ist ein Utility-first CSS-Framework, das die Gestaltung von Benutzeroberflächen durch vorgefertigte

CSS-Klassen stark vereinfacht. Nach Auswahl von tailwindcss im Add-on-Menü wird abgefragt, welche **Plugins** integriert werden sollen:

```
tailwindcss: Which plugins would you like to add?
❐ typography (@tailwindcss/typography)
❐ forms      (@tailwindcss/forms)
```

Plugin	Beschreibung
Typography	Stellt sinnvolle Standard-Styles für typografische Inhalte bereit. Eignet sich besonders für Content-lastige Seiten, z. B. Dokumentation oder Blogbeiträge.
Forms	Vereinheitlicht das Styling von nativen HTML-Formular-Elementen wie input, select, textarea etc. Passt sich automatisch an das Tailwind-Designsystem an.

Wir wählen beide Plugins aus.

5. Auswahl des SvelteKit Adapters

SvelteKit ist ein Meta-Framework, das Applikationen sowohl statisch als auch serverseitig oder als Hybrid generieren kann. **Adapter** übernehmen dabei die Aufgabe, die Anwendung für eine bestimmte Zielplattform aufzubereiten (z. B. als statische HTML-Seiten oder als Server-Handler für Vercel, Netlify, Node.js etc.).

Die Auswahl erfolgt im CLI nach Aktivierung des Add-ons `sveltekit-adapter`:

```
sveltekit-adapter: Which SvelteKit adapter would you like to use?
  auto
  node
❯ static (@sveltejs/adapter-static)
  vercel
  cloudflare-pages
  netlify
```

Adapter	Beschreibung	Typ	Empfohlene Einsatzzwecke
auto	Automatische Auswahl anhand der Umgebung. Praktisch für Entwicklung, aber nicht für produktives Deployment empfohlen.	auto-detect	Nur lokale Nutzung, z. B. für Tests

Adapter	Beschreibung	Typ	Empfohlene Einsatzzwecke
node	Erstellt ein Node.js-Handler zur Laufzeit, z. B. für Express, Fastify oder Cloud-Server.	SSR	Eigener Serverbetrieb, z. B. VPS, Docker-Container
static	Exportiert alle Seiten als HTML/CSS/JS. Kein dynamisches Routing, aber sehr performant.	SSG (Static Site)	GitHub Pages, Netlify, klassische Webserver
vercel	Optimierung für das Vercel-Ökosystem. Deployment erfolgt über Vercel CLI oder GitHub-Integration.	Edge-/SSR-ready	Hosting über vercel.com
cloudflare-pages	Unterstützung für Cloudflare Pages, inkl. Worker-Skripte.	Edge SSR	Deployment auf pages.cloudflare.com
netlify	Adapter mit Funktionen wie Functions, Redirects, SSR via Netlify.	Hybrid	Deployment auf netlify.com

Für unser Beispiel wählen wir `static`.

6. Wahl des Paketmanagers

```
Which package manager do you want to install dependencies with?
❯ npm
  yarn
  pnpm
  bun
  deno
```

Die Auswahl erfolgt entsprechend der individuellen Präferenz. `npm` ist in den meisten Fällen ausreichend.

7. Nächste Schritte nach dem Setup

Nach erfolgreichem Setup wird vom CLI folgender Ablauf vorgeschlagen:

```
cd my-sveltekit-project
git init && git add -A && git commit -m "Initial commit"   # optional
```

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

Damit wird der lokale Entwicklungsserver gestartet. Der Zugriff erfolgt typischerweise unter http://localhost:5173.

Projektstruktur nach dem Setup

Nach erfolgreichem Setup mit der SvelteKit CLI liegt eine vorstrukturierte Projektbasis vor. Diese Struktur ermöglicht den sofortigen Einstieg in die Anwendungsentwicklung, getrennt nach Konfiguration, Komponenten, Routen und Styling.

Ein typisches Grundgerüst (vereinfacht dargestellt):

```
my-sveltekit-project/
├── src/
│   ├── lib/
│   │   └── components/     ← Wiederverwendbare UI-Komponenten
│   ├── routes/             ← Seitenstruktur (basierend auf Datei-Routing)
│   │   └── +page.svelte    ← Startseite (Route: "/")
│   └── app.css             ← TailwindCSS-Einstiegspunkt
├── static/                 ← Öffentliche Assets (z. B. Bilder, favicon)
├── svelte.config.js        ← Zentrale SvelteKit-Konfiguration
├── tailwind.config.cjs     ← TailwindCSS-Konfiguration
├── postcss.config.cjs      ← PostCSS-Integration für Tailwind
├── tsconfig.json           ← TypeScript-Projektdefinition
├── package.json            ← Abhängigkeiten und Skripte
└── vite.config.js          ← Build-Tool-Konfiguration (Vite)
```

Pfad	Beschreibung
src/routes/	Implementierung der Seitenstruktur der Anwendung. Jede Datei oder jeder Ordner stellt eine Route dar.
src/routes/+page.svelte	Einstiegspunkt der Anwendung (Startseite, Route /). Kann sofort bearbeitet werden.
src/lib/	Globale Hilfsmittel, z. B. Komponenten, Stores, Services. Nicht Routen-spezifisch.
src/app.css	Haupt-Stylesheet. Hier wird TailwindCSS eingebunden (@tailwind base, components, utilities).
static/	Enthält öffentlich zugängliche Dateien (z. B. robots.txt, favicon.ico, statische Bilder).
svelte.config.js	Konfiguriert Adapter, Preprocessing, Pfade etc.
tailwind.config.cjs	Definiert Tailwind-Themes, Plugins und Pfade zur Content-Erkennung.

Pfad	Beschreibung
tsconfig.json	Konfiguriert das Verhalten des TypeScript-Compilers.

Grundkonzepte von Svelte 5

Svelte 5 führt mit den sogenannten *Runes* eine neue, klare Art ein, mit Reaktivität umzugehen.

- $state() für reaktiven Zustand
- $derived() für abgeleitete Werte
- $effect() für Nebenwirkungen (z. B. DOM-Manipulation)
- $props() für Props-Zugriff
- $bindable() für zwei-Wege-Bindung

Beispiel:

```
<script>
  let count = $state(0);
  let doubled = $derived(count * 2);
</script>

<button onclick={() => count++}>
  Doppelt: {doubled}
</button>
```

> Keine useState-Calls, keine riesige API – nur schlankes HTML & JavaScript.

TODO:

- Dateibenennung
- Import/Export von Komponenten/Modulen
- *Runes* im Detail
- TypeScript-Support

Styling & UX mit Tailwind CSS

Tailwind CSS passt hervorragend zu Svelte. Es erlaubt schnelles Prototyping, konsistentes Design und volle Kontrolle direkt im Markup.

```
<button class="bg-red-500 hover:bg-red-600 text-white px-4 py-2 rounded">
  Klick mich
</button>
```

TODO:

- Verweis auf Installation mit SvelteKit
- Dev- vs. Build-Mode

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.

Exkurs: 3D-Anwendungen im Handumdrehen mit Threlte

```
npm install three @threlte/core @threlte/extras
```