Advanced Web Technologies – Wiki zum Mastermodul

Dieses Buch ist meine persönliche Dokumentation und Sammlung von Ergebnissen zum Modul Advanced Web Technologies im Master Angewandte Informatik an der Hochschule Mainz.

Es enthält:

- Eigene Recherchen & Notizen zu zentralen Themenfeldern
- Wiki-Artikel, Grafiken & externe Ressourcen
- Dokumentation zu Beispielaufgaben
- Dokumentation zur Entwicklung eines komplexen Webprojekts mit Laravel Filament und Svelte

Themenbereiche:

- Architekturen (MVC, MPPM, Thin/Fat Client etc.)
- Frontend vs Backend (Vue, Angular, Node, PHP, Python)
- Kommunikation zwischen Client & Server (REST, Sockets, Microservices, Message Queuing)
- Testing (z. B. mit Selenium)
- ☁ Cloud-Apps (AWS, GCP, Azure)

- Einleitung
- Svelte als Frontend-Technologie
    - 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
- Backend-Technologien
  - Mein Weg zu PayloadCMS: Backend-Vergleich aus der Praxis
  - PayloadCMS
  - Erweiterung des Beispielprojekts: Vier Gewinnt Remote
- Client-Server-Kommunikation
  - Web Services mit GraphQL (statt REST/SOAP)
  - Microservices
  - Sockets
  - Message Queueing
- Testing
  - Oberflächentests mit Playwright
- Cloud-Apps

Einleitung

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

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
node	Erstellt ein Node.js-Handler zur Laufzeit, z. B. für Express, Fastify oder Cloud-Server.	SSR	Eigener Serverbetrieb, z. B. VPS, Docker-Container

Adapter	Beschreibung	Typ	Empfohlene Einsatzzwecke
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.
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

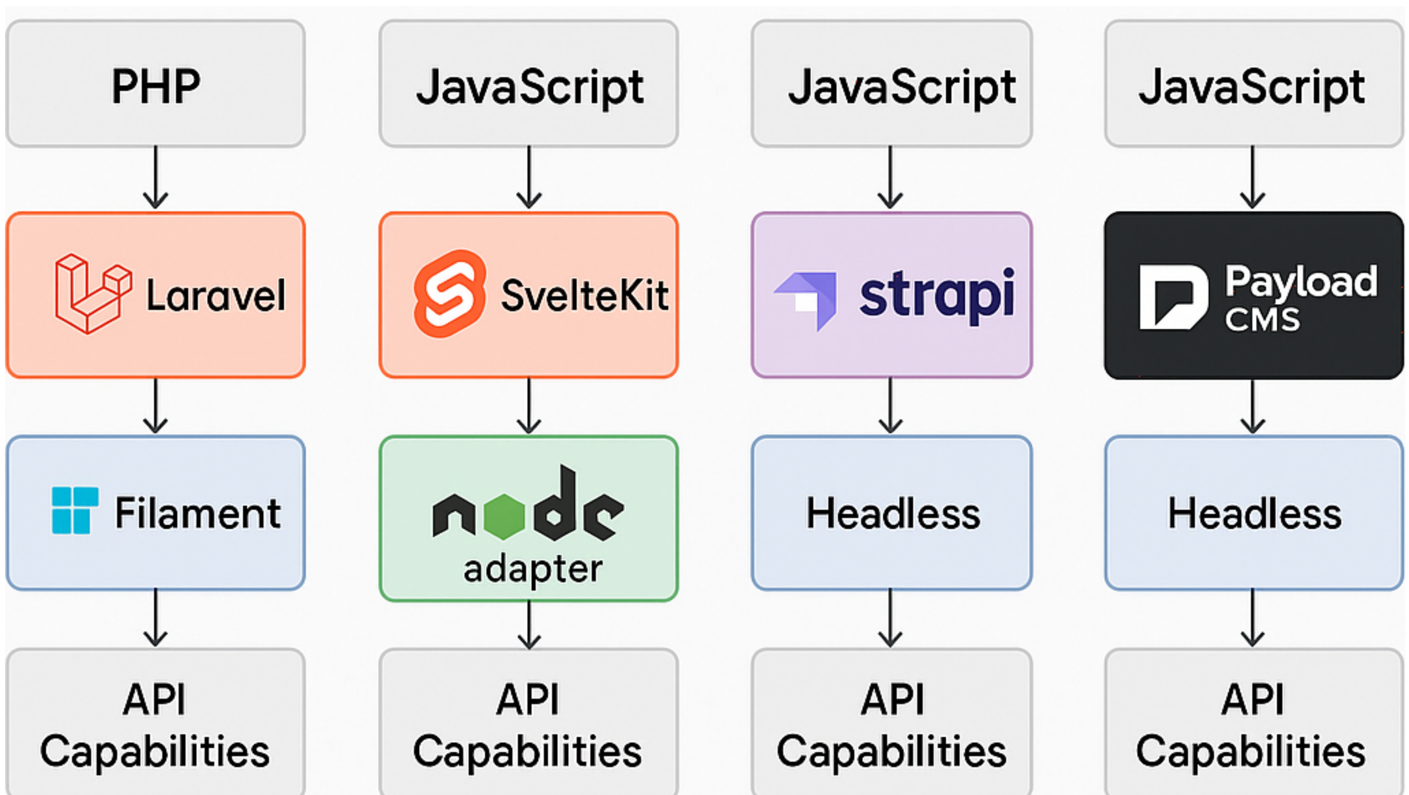
```
npm install three @threlte/core @threlte/extras
```

Backend-Technologien

Backend-Technologien

Mein Weg zu PayloadCMS: Backend-Vergleich aus der Praxis

In meinem Projekt stand ich vor der Aufgabe, ein modernes CMS aufzusetzen – mit möglichst viel Flexibilität, einem angenehmen Workflow und gutem API-Zugriff. Hier dokumentiere ich meine Evaluierung verschiedener Backend-Technologien und Frameworks – und wie ich schließlich bei **PayloadCMS** gelandet bin.



Laravel (PHP) – Stark, aber überdimensioniert und etwas altbacken?

Was mir gefallen hat: Laravel ist mächtig. Besonders das Eloquent ORM und das Ecosystem (Jobs, Queues, Policies etc.) bieten viele Features direkt „out of the box“. Mit **Filament** steht zudem ein sehr starkes Admin-Panel-Toolkit zur Verfügung.

Aber:

- Für ein CMS war mir das Setup oft zu schwergewichtig.
- PHP ist für viele Dinge okay – aber die Arbeit mit JSON, REST oder modernen Frontend-Workflows wirkt oft sperriger als bei node-basierten Lösungen.
- Auch das Hosting (z. B. auf Vercel oder Netlify) ist nicht ganz so smooth wie mit JavaScript-Stacks.

Fazit: Ein tolles Framework, aber für mein Ziel „Headless CMS mit modernem JS-Frontend“ war es nicht die erste Wahl.

SvelteKit – Elegantes Frontend, aber keine komplexen Backend-Lösungen

Was ich mochte: SvelteKit ist superschnell, modern und macht Spaß. Besonders die neue V5 ist sehr spannend für reaktive Frontends. Die Integration mit dem Node-Adapter ist einfach – und fürs **Frontend** sehe ich darin meine langfristige Lösung.

Was mir fehlte:

- Komplexere Backend-Funktionalität wie Rollen, dynamische Models, Auth oder Admin-Panels fehlen oder müssen selbst gebaut werden.
- Es gibt zwar ein paar Tools (z. B. Lucid, Prisma), aber der Ökosystem-Vergleich mit React zeigt: weniger Auswahl.
- Kein echtes CMS-Feeling.

Fazit: Perfekt für Frontends. Als CMS-Backend nicht geeignet.

Strapi – Gute API, aber UI nicht meins

Pluspunkte:

- Sehr reifes Headless CMS auf Basis von Node.js.
- Gute REST- und GraphQL-Schnittstellen.
- Rollen-/Rechtemanagement, dynamische Content-Types, einfache Auth.

Aber:

- Das Admin-Panel wirkt auf mich visuell und UX-technisch nicht ansprechend.
- Die Konfiguration ist teilweise uneinheitlich (Code vs. UI).
- Etwas schwergewichtig für kleinere Projekte.

Fazit: Technisch solide – aber ich habe mich im Admin nicht wohlgefühlt.

PayloadCMS

Warum ich geblieben bin:

- Komplett in TypeScript.
- Das Admin-Panel ist schnell, intuitiv und sehr anpassbar.
- Content-Modeling über Code (statt Click UI).
- Lokale Entwicklung, gute Dokumentation, offene Architektur.
- Built-in Auth, Access Control, Dateiupload, Versioning etc.
- Ideal für Entwickler:innen.

Fazit: PayloadCMS ist genau das, was ich gesucht habe: Entwicklerfreundlich, modern, headless und API-zentriert. Es passt gut zu meinem Setup mit SvelteKit als Frontend – und ist damit mein Favorit für das Projekt.

Beispiel-Systemarchitektur

```
+------------------------------+
|      Frontend (SvelteKit)  |
|------------------------------|
|  - SvelteKit App            |
|  - Node Adapter             |
|    (für API-Proxy)          |
+-------------+----------------+
              |
              | REST / GraphQL
              v
+-----------------------------+
|     Backend (PayloadCMS)    |
```

```
|------------------------------|
|  - Payload Server (Next.js)  |
|  - Access Control & Auth     |
|  - Custom Hooks / Logic      |
|  - ORM / Content Models      |
|  - File Uploads / Admin UI   |
+-------------+----------------+
              |
              v
+------------------------------+
|          MongoDB             |
|------------------------------|
|  - Persistente Speicherung   |
|    von Inhalten & Usern      |
+------------------------------+
```

Weiterführende Ressourcen

- PayloadCMS Docs
- SvelteKit V5 Docs
- Filament für Laravel

PayloadCMS

PayloadCMS ist ein modernes, Headless CMS, das vollständig auf **Node.js** basiert und speziell für Entwickler:innen konzipiert ist. Es kombiniert ein leistungsfähiges Admin-Panel mit einem *code-first* -Ansatz, wodurch Inhalte, Strukturen und Logiken direkt im Code definiert werden können.

Überblick

Eigenschaft	Beschreibung
CMS-Typ	Headless CMS
Backend	Node.js + Express
Sprache	TypeScript (auch JavaScript möglich)
API-Schnittstellen	REST und GraphQL
Admin-Oberfläche	Automatisch generiert aus dem Code
Authentifizierung	Integriert (JWT, Sessions, Role-based Access)
Datenbanken	MongoDB (Standard), **PostgreSQL / SQLite** via Kysely (experimentell)
ORM/Query Builder	Mongoose (MongoDB) / Kysely (SQL-DBs wie SQLite)

Code-First Schema

Alle Collections (Inhaltstypen) werden im Code als JavaScript/TypeScript-Objekte definiert.

```
import { CollectionConfig } from 'payload/types';


const Posts: CollectionConfig = {
  slug: 'posts',
  fields: [
    { name: 'title', type: 'text', required: true },
    { name: 'content', type: 'richText' },
```

```
  ],
};

export default Posts;
```

Authentifizierung & Rollen

- Integrierte Benutzerverwaltung
- Rollenbasierte Zugriffskontrolle (Access Control Policies)
- Auth-Collection konfigurierbar

Dateiuploads & Medien

- Unterstützung für File-Uploads (lokal oder via Cloud)
- Optimierung und Vorschau automatisch im Admin-Panel

Hooks & Middleware

- Asynchrone Hooks vor/nach Aktionen
- Business-Logik z. B. bei beforeChange, afterDelete, etc.

Mehrsprachigkeit

- Unterstützung für i18n (lokalisierte Inhalte)

Datenbankunterstützung

Datenbank	Standard?	ORM / Query Layer	Hinweise
MongoDB	□	Mongoose	Reif & empfohlen
SQLite	□□ (ab 1.12+)	Kysely	Gut für lokale Dev
PostgreSQL	□□ (ab 1.12+)	Kysely	Für produktive SQL-Setups

Backend-Technologien

Erweiterung des Beispielprojekts: Vier Gewinnt Remote

In diesem Artikel erweitern wir unser Beispielprojekt um ein Backend, damit Spieler auch online gegeneinander spielen können.

Client-Server-Kommunikation

Web Services mit GraphQL (statt REST/SOAP)

GraphQL ist ein modernes API-Paradigma, das sich zunehmend als Alternative zu klassischen Web-Service-Modellen wie REST oder SOAP etabliert. Im Vergleich zu diesen Modellen erlaubt GraphQL eine flexible, clientgesteuerte Datenabfrage über eine einheitliche Schnittstelle. Seine typisierte Struktur, die Möglichkeit zur Introspektion und die präzise Steuerung von Abfrageinhalten machen GraphQL besonders attraktiv für dynamische Webanwendungen und KI-basierte Systeme.

Einordnung: Was sind Web Services?

Web Services stellen eine standardisierte Methode dar, um Daten zwischen Client und Server auszutauschen – meist über das HTTP-Protokoll. Sie ermöglichen plattformunabhängige Kommunikation, eine lose Kopplung zwischen Systemkomponenten und den Zugriff auf zentrale Datenquellen oder Geschäftslogik.

Etablierte Architekturen im Vergleich

Im Laufe der Zeit haben sich drei grundlegende Web-Service-Modelle etabliert: **SOAP**, **REST** und **GraphQL**. Die folgende Tabelle bietet eine strukturierte Gegenüberstellung:

Kriterium	SOAP	REST	GraphQL
Architekturtyp	Protokollbasiert	Architekturstil	Abfragesprache & Laufzeitumgebung
Transportprotokoll	HTTP, SMTP, TCP	HTTP	HTTP (meist POST)
Datenformat	XML	JSON, XML	JSON
Endpunkte	Mehrere, pro Operation	Mehrere, pro Ressource	Ein einziger Endpunkt
Abfrageflexibilität	Gering – festgelegte Operationen	Mittel – durch verschiedene Endpunkte	Hoch – clientseitig definierte Abfragen
Versionierung	Über WSDL-Dateien	Häufig über URL-Versionierung (z. B. /v1/)	Nicht erforderlich – Schema kann erweitert werden

Kriterium	SOAP	REST	GraphQL
Caching	Komplex, selten genutzt	Gut unterstützt durch HTTP-Caching	Eingeschränkt – abhängig von Abfragekomplexität
Fehlerbehandlung	Standardisierte Fehlercodes in XML	HTTP-Statuscodes + optionale Fehlermeldungen	Fehlerobjekte im JSON-Format
Sicherheitsmechanismen	WS-Security (z. B. XML-Signaturen)	TLS/HTTPS, OAuth, API-Keys	TLS/HTTPS, OAuth, API-Keys
Einsatzgebiete	Unternehmensanwendungen, Legacy-Systeme	Web-APIs, Microservices, mobile Anwendungen	Moderne SPAs, mobile Apps, datenintensive Anwendungen
Komplexität	Hoch – umfangreiche Spezifikationen	Mittel – abhängig vom Design	Hoch – insbesondere bei komplexen Schemas

GraphQL im Detail

GraphQL ist ein modernes API-Design-Paradigma, das 2015 von Facebook veröffentlicht wurde. Es verfolgt einen abfragegetriebenen Ansatz, bei dem der Client exakt definiert, welche Daten benötigt werden. Dies unterscheidet sich grundlegend von REST, wo die Struktur der Antwort durch den Server vorgegeben wird.

Ein GraphQL-Endpunkt akzeptiert zwei Arten von Operationen:

- **Queries**: Daten vom Server lesen
- **Mutations**: Daten auf dem Server verändern

Beispiel: Datenabfrage mit einer Query

```
query {
  books {
    title
    author
  }
}
```

Diese Abfrage fordert vom Server alle Bücher und gibt pro Buch nur die Felder `title` und `author` zurück – weitere Felder wie `id`, `createdAt` etc. werden ignoriert, sofern sie nicht explizit abgefragt werden.

Beispiel: Datenerzeugung mit einer Mutation

```
mutation {
  addBook(title: "Clean Code", author: "Robert C. Martin") {
    id
    title
    author
  }
}
```

Mutationen ähneln POST-Requests in REST. Sie erlauben das Anlegen, Verändern oder Löschen von Datenobjekten.

Aufbau eines GraphQL-Backends

Ein GraphQL-Dienst basiert auf einem Schema, das die verfügbaren Typen, Queries und Mutationen beschreibt. Die eigentliche Logik liegt in sogenannten **Resolvern**, die die Daten bereitstellen oder verändern.

Ein einfaches Setup umfasst:

1. Definition des Schemas (Typen & Operationen)
2. Implementierung der Resolver
3. Bereitstellung über einen GraphQL-Server

Apollo Server

Für Node.js-Projekte ist **Apollo Server** eine der bekanntesten und am besten dokumentierten Lösungen zur Umsetzung eines GraphQL-Endpunkts. Apollo Server stellt Werkzeuge bereit, um:

- ein Typschema zu definieren
- Resolver-Funktionen zu implementieren
- Middleware-Logik für Authentifizierung, Logging oder Caching zu integrieren
- einen interaktiven Playground für Queries bereitzustellen

Apollo lässt sich leicht mit Express oder Fastify kombinieren und ist durch seine Modularität für einfache wie komplexe Projekte gleichermaßen geeignet.

Weitere verwandte Tools im Apollo-Ökosystem sind:

- **Apollo Client**: für die Anbindung im Frontend (z. B. in React oder Vue)
- **Apollo Federation**: für das Zusammenführen mehrerer GraphQL-Services (Microservices)

GraphQL und KI

Ein zunehmend relevanter Anwendungsbereich für GraphQL ist die Kombination mit künstlicher Intelligenz (KI). Dabei ergeben sich mehrere Synergieeffekte:

Introspektive APIs für maschinelles Verständnis

GraphQL-APIs sind introspektiv – das heißt, sie geben auf Anfrage strukturiert Auskunft darüber, welche Operationen erlaubt sind, welche Parameter benötigt werden und welche Typen verfügbar sind. Das ist besonders im KI-Kontext von Vorteil:

- Eine KI kann über eine Introspection-Query selbst herausfinden, wie die API funktioniert.
- Auf dieser Basis lassen sich automatisch passende Abfragen generieren (z. B. aus natürlicher Sprache).
- Die Typsicherheit reduziert Fehlerraten und erleichtert die automatische Validierung.

Beispielhafte Introspection-Query

```
{
  __schema {
    queryType {
      fields {
        name
        args {
          name
          type {
            name
          }
        }
      }
    }
  }
}
```

Damit können LLMs (wie ChatGPT) oder andere KI-Systeme eigenständig die Struktur einer API analysieren – ohne auf externe Dokumentation angewiesen zu sein. REST-basierte APIs erfordern dafür meist zusätzliche Spezifikationen (z. B. OpenAPI).

Weitere Einsatzfelder

- **Datenbereitstellung für ML-Pipelines**
- **API-Zugriff durch natürliche Sprache** (Text-zu-Query)
- **Automatisiertes API-Monitoring durch KI-gestützte Analyse**

Typische Einsatzszenarien

GraphQL eignet sich besonders für Anwendungen mit variablen oder dynamisch zusammensetzbaren Datenansichten, z. B.:

- Single Page Applications (SPAs) mit Frontend-Frameworks wie Svelte, React oder Vue
- Mobile Apps mit geringem Datenbudget
- Headless CMS-Lösungen mit flexiblen Content-Views

Im CMS-Kontext ist GraphQL besonders dann interessant, wenn verschiedene Frontends unterschiedliche Datenansichten benötigen – etwa eine Vorschau in der Admin-UI und eine kompakte Darstellung im öffentlichen Blog.

Zusammenfassung

GraphQL ist ein leistungsfähiges und zugleich flexibel einsetzbares Werkzeug für moderne Webarchitekturen. Es bietet eine strukturierte und typisierte Alternative zu REST und ermöglicht effiziente, clientgesteuerte Datenabfragen. Die Einführung in ein Projekt lohnt sich vor allem dann, wenn unterschiedliche Clients auf dieselbe API zugreifen oder Daten gezielt gefiltert werden sollen.

Ein vollständiges Beispiel zur Umsetzung eines GraphQL-Servers mit Apollo und TypeScript findest du in unserem Repository: https://gitlab.rlp.net/marius.klein2/awt-marius-klein/-/tree/main/beispielaufgaben/server-client-kommunikation/1-graphql-book-service

Microservices

Microservices sind ein Architekturparadigma, das sich in den letzten zehn Jahren stark verbreitet hat. Der Begriff steht für eine Herangehensweise, bei der Software nicht mehr als eine große, monolithische Anwendung entwickelt und betrieben wird, sondern als Sammlung kleiner, voneinander unabhängiger Dienste. Jeder dieser Dienste erfüllt eine klar abgegrenzte Aufgabe innerhalb des Gesamtsystems und kommuniziert mit anderen Diensten über wohldefinierte Schnittstellen.

Ursprünge und Motivation

Die Idee der Microservices entwickelte sich im Kontext wachsender monolithischer Systeme, die mit zunehmendem Umfang schwer wart- und testbar wurden. Die Beobachtung: Je größer der Code und je mehr Teams beteiligt sind, desto größer wird die Reibung beim gemeinsamen Arbeiten an einer gemeinsamen Codebasis. Änderungen in einem Bereich ziehen häufig Änderungen in anderen Bereichen nach sich, und das Deployment eines Features kann durch die Abhängigkeit von nicht fertiggestellten Teilbereichen blockiert sein.

Microservices sind eine Antwort auf diese Skalierungsprobleme. Sie setzen auf:

- **Lose Kopplung**: Jeder Dienst ist weitgehend unabhängig von den anderen.
- **Hohe Kohäsion**: Jeder Dienst konzentriert sich auf eine spezifische Funktion oder Domäne.
- **Unabhängige Deployments**: Jeder Dienst kann einzeln aktualisiert werden.
- **Technologievielfalt**: Dienste können mit verschiedenen Sprachen und Frameworks implementiert werden.

Technisches Grundprinzip

Ein Microservice-System besteht aus mehreren eigenständigen Prozessen, die über Netzwerkprotokolle miteinander kommunizieren – in der Regel über HTTP (REST oder GraphQL), gRPC oder asynchrone Messaging-Systeme wie Kafka oder RabbitMQ.

Jeder Dienst bringt idealerweise seine gesamte technische Infrastruktur mit:

- eigenes Repository

- eigene Build- und Deployment-Pipeline
- eigene Datenbank oder zumindest isolierten Zugriff auf persistente Daten

Diese vollständige Kapselung wird in der Praxis jedoch nicht immer konsequent umgesetzt. Besonders im Bereich der Datenpersistenz kommt es oft zu Überschneidungen, etwa wenn mehrere Dienste auf dieselbe Datenbank oder Tabelle zugreifen müssen. Das steht im Spannungsfeld zum Prinzip der vollständigen Isolation.

Paradigmen und Interpretationen

Es gibt verschiedene Interpretationen und Ableitungen des Microservices-Gedankens:

- **Domain-Driven Design (DDD)**: Dienste orientieren sich an fachlichen Domänen („Bounded Contexts") und spiegeln die Struktur der Geschäftslogik wider.
- **Self-Contained Systems (SCS)**: Jeder Dienst enthält Backend, Businesslogik und sogar das zugehörige UI.
- **Modular Monolith**: Die Anwendung bleibt ein Prozess, ist aber intern stark modularisiert und potenziell in Microservices aufteilbar.

Diese Vielfalt führt dazu, dass unter dem Begriff „Microservices" oft unterschiedliche Dinge verstanden werden. Eine klare und konsistente Definition fehlt bis heute.

Vorteile von Microservices

Vorteil	Beschreibung
Skalierbarkeit	Einzelne Services lassen sich unabhängig skalieren (z. B. CPU-hungrige Module).
Flexibilität	Technologieentscheidungen können pro Dienst individuell getroffen werden.
Deployment-Freiheit	Teams können unabhängig voneinander releasen.
Fehlertoleranz	Ein Fehler in einem Dienst betrifft nicht notwendigerweise das Gesamtsystem.
Teamautonomie	Kleinere Teams können Verantwortung für „ihren" Dienst übernehmen.

Herausforderungen und Kritik

Nachteil / Herausforderung	Beschreibung
Systemkomplexität	Die Gesamtarchitektur wird komplexer, insbesondere hinsichtlich Kommunikation und Datenfluss.
Testing-Aufwand	Integrationstests und End-to-End-Tests werden aufwendiger.
Verteilte Transaktionen	ACID-Eigenschaften sind über mehrere Dienste schwer zu garantieren.
Fehlende Übersicht	Es kann schwierig sein, einen systemweiten Überblick zu behalten.
Tooling & Infrastruktur	Microservices benötigen reifes CI/CD, Observability, Logging, Monitoring.

Entwicklung und Trendwende

Microservices galten über Jahre hinweg als das Ideal moderner Softwarearchitektur. Viele Unternehmen haben ihre Systeme unter großem Aufwand von Monolithen auf Microservices umgestellt. Mittlerweile mehren sich jedoch die Stimmen, die auf die Nachteile und Überforderungen durch zu viele verteilte Komponenten hinweisen.

In der Praxis zeigt sich: Nicht jedes Team ist darauf vorbereitet, eine derart feingranulare Architektur sinnvoll zu betreiben. Auch große Unternehmen wie Amazon oder Uber haben Teile ihrer Architektur wieder konsolidiert oder stark modularisierte Monolithen eingeführt.

Derzeit entstehen vermehrt Architekturen, die eine Balance zwischen Modularität und Einfachheit suchen:

- **Modulare Monolithen** mit klar abgegrenzten Domänen und interner API-Struktur
- **„Micro-Frontends“** als Antwort auf verteilte Zuständigkeiten im UI-Bereich
- **Backend-for-Frontend (BFF)**-Muster zur Trennung von domänenspezifischer Darstellung

Wann sind Microservices sinnvoll?

Microservices lohnen sich besonders, wenn folgende Bedingungen erfüllt sind:

- Das System ist fachlich sehr komplex und wächst weiter.
- Es gibt mehrere Entwicklungsteams, die unabhängig voneinander arbeiten sollen.
- Die Anwendung muss stark skalieren oder hohe Verfügbarkeit garantieren.
- Es besteht ein Bedarf an Technologievielfalt oder Dienstgrenzen entlang von Domänen.

Für kleinere Projekte oder Teams kann ein gut strukturierter Monolith hingegen **deutlich effektiver** und wartbarer sein.

Infrastruktur und Orchestrierung

Microservices entfalten ihr volles Potenzial erst mit der passenden Infrastruktur. Zwei Schlüsseltechnologien, die den Betrieb verteilter Systeme ermöglichen, sind:

- **Docker**: Eine Container-Technologie, mit der einzelne Microservices isoliert, reproduzierbar und unabhängig voneinander paketiert werden können. Jeder Dienst läuft in seinem eigenen Container mit definierter Laufzeitumgebung.
- **Kubernetes**: Eine Open-Source-Plattform zur Orchestrierung von Containern (wie Docker). Kubernetes verwaltet die Verteilung, Skalierung, Wiederherstellung und Kommunikation von Microservices über einen Cluster aus Maschinen hinweg. Es ist damit das Rückgrat vieler Cloud-nativer Architekturen.

Ein „Cluster" ist in diesem Zusammenhang eine Gruppe vernetzter physischer oder virtueller Server, auf denen Kubernetes die Microservices verteilt und steuert.

Der Aufbau eines produktionsreifen Microservice-Systems setzt daher Kenntnisse in Containerisierung und Orchestrierung voraus – ein Grund, warum der Infrastrukturaufwand im Vergleich zu monolithischen Architekturen deutlich höher ist.

Glossar

Begriff	Bedeutung
API-Gateway	Zentrale Anlaufstelle für externe Anfragen an ein Microservice-System.
Cluster	Gruppe aus mehreren Servern, die gemeinsam eine verteilte Umgebung bilden.
Container	Leichtgewichtige Umgebung zur isolierten Ausführung von Software-Komponenten.
DDD	Domain-Driven Design – domänenzentrierter Ansatz zur Softwaremodellierung.
Docker	Plattform zur Containerisierung und zum Deployment verteilter Anwendungen.
Kubernetes	System zur automatisierten Verwaltung, Skalierung und Orchestrierung von Containern.

Begriff	Bedeutung
Monolith	Architektur, bei der die gesamte Anwendung als eine Einheit betrieben wird.
Self-contained System	Architekturansatz, bei dem jeder Dienst auch UI, Logik und Persistenz umfasst.
Service Discovery	Verfahren, mit dem Microservices einander automatisch auffinden können.

Sockets

Das WebSocket-Protokoll ermöglicht eine bidirektionale, persistente Kommunikation zwischen Client und Server über eine einzelne TCP-Verbindung. Im Gegensatz zum traditionellen HTTP-Protokoll, das auf einem Anfrage-Antwort-Modell basiert, erlaubt WebSocket eine kontinuierliche Datenübertragung in beide Richtungen, ohne dass der Client ständig neue Anfragen stellen muss. Dies ist besonders nützlich für Anwendungen, die Echtzeitdaten erfordern, wie Chat-Anwendungen, Online-Spiele oder Live-Dashboards.

Historischer Kontext

Vor der Einführung von WebSockets waren Entwickler auf Techniken wie Polling oder Long Polling angewiesen, um Echtzeitkommunikation zu simulieren. Diese Methoden waren jedoch ineffizient und belasteten sowohl Server als auch Netzwerkressourcen. Mit der Standardisierung des WebSocket-Protokolls durch die IETF im Jahr 2011 (RFC 6455) wurde eine effizientere Lösung geschaffen, die echte bidirektionale Kommunikation ermöglicht.

Funktionsweise von WebSockets

Verbindungsaufbau (Handshake)

Der Verbindungsaufbau beginnt mit einem HTTP-Request des Clients, der ein Upgrade auf das WebSocket-Protokoll anfordert. Wenn der Server zustimmt, antwortet er mit einem 101 Switching Protocols-Statuscode, und die Verbindung wird auf WebSocket umgestellt. Ab diesem Punkt bleibt die Verbindung offen und ermöglicht den kontinuierlichen Datenaustausch.

Datenübertragung

Nach dem erfolgreichen Handshake können sowohl Client als auch Server jederzeit Nachrichten senden. Die Kommunikation erfolgt über Frames, die entweder Text- oder Binärdaten enthalten können. Diese Frames sind leichtgewichtig und verursachen minimalen Overhead, was zu einer effizienten Datenübertragung führt.

Vorteile von WebSockets

- **Bidirektionale Kommunikation**: Sowohl Client als auch Server können jederzeit Daten senden und empfangen.
- **Persistente Verbindung**: Die Verbindung bleibt offen, wodurch wiederholte Handshakes vermieden werden.
- **Geringer Overhead**: Im Vergleich zu HTTP sind die Header kleiner, was die Bandbreite schont.
- **Echtzeitfähigkeit**: Ideal für Anwendungen, die sofortige Datenaktualisierungen benötigen.

Nachteile und Herausforderungen

- **Firewall- und Proxy-Kompatibilität**: Einige Netzwerke blockieren WebSocket-Verbindungen, was zusätzliche Konfiguration erfordern kann.
- **Sicherheitsaspekte**: Da die Verbindung offen bleibt, müssen Mechanismen implementiert werden, um unbefugten Zugriff zu verhindern.
- **Komplexität**: Die Implementierung kann komplexer sein als bei traditionellen HTTP-Anwendungen.

Anwendungsfälle

- **Chat-Anwendungen**: Echtzeitkommunikation zwischen Benutzern.
- **Online-Spiele**: Synchronisation von Spielzuständen in Echtzeit.
- **Finanz- und Börsenanwendungen**: Live-Aktualisierungen von Kursen und Marktdaten.
- **Kollaborative Tools**: Gemeinsames Bearbeiten von Dokumenten oder Whiteboards.

Implementierung im Backend mit Node.js und ws

Die ws-Bibliothek ist eine schlanke und performante Lösung zur Implementierung von WebSocket-Servern in Node.js. Sie ermöglicht die einfache Einrichtung eines Servers, der bidirektionale

Kommunikation mit Clients unterstützt.

Installation

Zunächst wird ein neues Node.js-Projekt erstellt und die ws-Bibliothek installiert:

```
mkdir websocket-server
cd websocket-server
npm init -y
npm install ws
```

Einfacher WebSocket-Server

Im Anschluss kann ein einfacher WebSocket-Server wie folgt implementiert werden:

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  console.log('Neuer Client verbunden');

  ws.on('message', (message) => {
    console.log(`Empfangen: ${message}`);
    ws.send(`Server: ${message}`);
  });

  ws.on('close', () => {
    console.log('Client hat die Verbindung geschlossen');
  });
});

console.log('WebSocket-Server läuft auf ws://localhost:8080');
```

Dieser Server lauscht auf Port 8080 und ermöglicht es Clients, Nachrichten zu senden und Antworten zu empfangen.

Erweiterte Funktionen

Die ws-Bibliothek bietet zusätzliche Funktionen, wie z.B.:

- **Broadcasting**: Versenden von Nachrichten an alle verbundenen Clients.
- **Ping/Pong-Mechanismus**: Überprüfung der Verbindungslatenz und -stabilität.
- **Integration mit HTTP-Servern**: Kombination von WebSocket- und HTTP-Servern auf demselben Port.

Ein Beispiel für Broadcasting:

```
wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    // Nachricht an alle Clients senden
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  });
});
```

Implementierung im Frontend

Die WebSocket-API ist in modernen Browsern nativ verfügbar und ermöglicht eine einfache Integration:

```
const socket = new WebSocket('ws://example.com/socket');

socket.onopen = () => {
  console.log('Verbindung geöffnet');
  socket.send('Hallo Server!');
};

socket.onmessage = (event) => {
  console.log('Nachricht vom Server:', event.data);
};
```

```
socket.onclose = () => {
  console.log('Verbindung geschlossen');
};

socket.onerror = (error) => {
  console.error('Fehler:', error);
};
```

In diesem Beispiel wird eine Verbindung zum Server hergestellt, eine Nachricht gesendet und eingehende Nachrichten sowie Verbindungsereignisse behandelt.

Zusammenfassung

WebSockets bieten eine leistungsfähige Lösung für Anwendungen, die Echtzeitkommunikation erfordern. Durch die bidirektionale, persistente Verbindung können Daten effizient und mit minimaler Latenz übertragen werden. Trotz einiger Herausforderungen, wie Sicherheitsaspekte und Netzwerkkompatibilität, sind WebSockets ein unverzichtbares Werkzeug für moderne Webanwendungen.

> Für weitere Informationen und tiefergehende technische Details empfiehlt sich die offiziellen Spezifikation RFC 6455 sowie die Dokumentation auf MDN Web Docs.

Client-Server-Kommunikation

Message Queueing

Hier ist ein umfassender Wiki-Artikel zum Thema **Message Queuing**, mit besonderem Fokus auf **RabbitMQ** als Praxisbeispiel:

□□ Message Queuing – Grundlagen und Praxis mit RabbitMQ

1. Einführung: Was ist Message Queuing?

Message Queuing (MQ) ist ein Kommunikationsparadigma, das es ermöglicht, Nachrichten asynchron zwischen verschiedenen Komponenten eines Systems auszutauschen. Dabei werden

Nachrichten in einer Warteschlange (Queue) zwischengespeichert, bis sie von einem Empfänger (Consumer) verarbeitet werden. Dieses Modell fördert die Entkopplung von Systemkomponenten und erhöht die Skalierbarkeit und Fehlertoleranz von Anwendungen.

2. Vorteile von Message Queuing

- **Asynchrone Kommunikation**: Sender und Empfänger müssen nicht gleichzeitig aktiv sein.
- **Entkopplung**: Komponenten können unabhängig voneinander entwickelt und betrieben werden.
- **Lastverteilung**: Nachrichten können auf mehrere Empfänger verteilt werden, um die Verarbeitungslast zu verteilen.
- **Fehlertoleranz**: Nachrichten bleiben in der Queue, bis sie erfolgreich verarbeitet wurden, was die Zuverlässigkeit erhöht.
- **Skalierbarkeit**: Einfaches Hinzufügen weiterer Empfänger zur Verarbeitung steigender Nachrichtenmengen.

Anwendungsfälle

- **Auftragsverarbeitung**: Bestellungen werden in einer Queue gespeichert und von einem Backend-System verarbeitet.
- **E-Mail-Versand**: E-Mails werden als Nachrichten in eine Queue gestellt und von einem separaten Dienst versendet.
- **Log-Verarbeitung**: Anwendungslogs werden gesammelt und asynchron analysiert.
- **Microservices-Kommunikation**: Services kommunizieren über Nachrichten, um lose gekoppelt zu bleiben.

RabbitMQ – Ein Praxisbeispiel

RabbitMQ ist ein weit verbreiteter, quelloffener Message Broker, der das **Advanced Message Queuing Protocol (AMQP)** implementiert. Es ermöglicht das Senden, Empfangen und Weiterleiten von Nachrichten zwischen Anwendungen oder Diensten.

Grundkonzepte

- **Producer**: Erzeugt und sendet Nachrichten.
- **Exchange**: Empfängt Nachrichten vom Producer und leitet sie gemäß bestimmter Regeln weiter.
- **Queue**: Speichert Nachrichten, bis sie vom Consumer abgeholt werden.
- **Consumer**: Empfängt und verarbeitet Nachrichten aus der Queue.

Exchange-Typen

- **Direct**: Leitet Nachrichten basierend auf einer exakten Routing-Key-Übereinstimmung weiter.
- **Fanout**: Leitet Nachrichten an alle gebundenen Queues weiter, unabhängig vom Routing Key.
- **Topic**: Leitet Nachrichten basierend auf Musterabgleich des Routing Keys weiter.
- **Headers**: Leitet Nachrichten basierend auf Header-Attributen weiter.

Implementierung mit RabbitMQ

Installation

RabbitMQ kann lokal installiert oder über Docker bereitgestellt werden. Eine einfache Möglichkeit ist die Verwendung des offiziellen Docker-Images:

```
docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

Dies startet RabbitMQ mit dem Management-Plugin, das über http://localhost:15672 erreichbar ist.

Beispiel: Nachricht senden und empfangen mit Python

Verwendung der pika-Bibliothek:

Producer (Sender):

```
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')
channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

Consumer (Empfänger):

```
import pika

def callback(ch, method, properties, body):
    print(f" [x] Received {body}")

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')
channel.basic_consume(queue='hello', on_message_callback=callback, auto_ack=True)
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Diese einfachen Beispiele zeigen, wie Nachrichten in eine Queue gesendet und von dort empfangen werden können.

Zusammenfassung

Message Queuing ist ein leistungsfähiges Muster zur asynchronen Kommunikation in verteilten Systemen. RabbitMQ bietet eine robuste und flexible Implementierung dieses Musters und ist in vielen Szenarien einsetzbar, von einfachen Anwendungen bis hin zu komplexen Microservices-Architekturen.

Weiterführende Ressourcen:

- RabbitMQ Tutorials
- RabbitMQ Dokumentation
- AMQP 0-9-1 Referenz

Testing

Oberflächentests mit Playwright

Einleitung

Oberflächentests (UI-Tests) stellen sicher, dass Webanwendungen sich so verhalten, wie es Benutzer erwarten – unabhängig vom internen Code. Mit **Playwright** steht ein modernes, mächtiges Test-Framework zur Verfügung, das speziell für dynamische Web-UIs entwickelt wurde. Dieser Artikel stellt Playwright vor, zeigt seinen Einsatz in einem typischen Projekt mit SvelteKit und vergleicht es mit Alternativen wie Cypress und Selenium. Darüber hinaus werden praktische Strategien zur Teststruktur, Fehlervermeidung und Debugging vermittelt.

Was ist Playwright?

Playwright ist ein von Microsoft entwickeltes Open-Source-Tool für **UI-Tests** von Webanwendungen. Tests werden in echten Browserinstanzen (Chromium, Firefox, WebKit) ausgeführt und über DevTools-Protokolle gesteuert. Besonders hervorzuheben sind:

- Unterstützung mehrerer Browser
- Automatische Warte-Mechanismen
- Netzwerk-Interception, Screenshot, Tracing, Debugging

Typischer Playwright-Workflow

Playwright erkennt standardmäßig alle Testdateien mit den Endungen `.spec.ts` oder `.test.ts`. Es ist also üblich, Testdateien nach dem Schema `xyz.spec.ts` zu benennen. Dies ist keine Pflicht, aber eine empfohlene Konvention, da sie vom Test-Runner automatisch erkannt wird.

Ein vollständiger Testdurchlauf mit Playwright besteht in der Regel aus drei Schritten:

1. **Test schreiben**
    - Der Test wird als JavaScript- oder TypeScript-Datei im `e2e/`-Verzeichnis abgelegt.
    - Beispiel:

```
test('zeigt Loginformular', async ({ page }) => {
  await page.goto('/login');
  await expect(page.locator('form')).toBeVisible();
});
```

2. **Test ausführen**
   - Im Terminal ausführen:
     ```
     npx playwright test
     ```
   - Optional: Nur bestimmte Tests oder Dateien ausführen:
     ```
     npx playwright test login.spec.ts
     ```
3. **Ergebnisse analysieren**
   - Playwright gibt im Terminal an, ob Tests bestanden oder fehlgeschlagen sind.
   - Bei Fehlern:
     - Trace-Dateien oder Screenshots analysieren
     - Optional: `--debug` oder `--trace on` verwenden

     Beispiel:
     ```
     npx playwright show-trace trace.zip
     ```

Dieser Workflow ist die Grundlage für einfache lokale Tests, funktioniert aber auch identisch in CI/CD-Umgebungen (z. B. GitHub Actions).

Automatische Warte-Mechanismen

Playwright wartet **intelligent und automatisch** auf Ereignisse im DOM. Das bedeutet: Wenn du z. B. ein Element anklickst oder eine Seite neu lädst, wartet Playwright so lange, bis die Seite bereit ist (Ladezustand, Sichtbarkeit, Interaktivität), **ohne dass du explizit `wait`-Befehle schreiben musst**. Das reduziert Fehler durch Timing-Probleme erheblich und macht Tests stabiler als bei älteren Tools wie Selenium.

Beispiel:

```
await page.click('text=Speichern'); // wartet intern, bis der Button sichtbar & klickbar ist
```

Netzwerk-Interception

Mit Playwright kannst du **HTTP-Anfragen abfangen und manipulieren**, bevor sie den Server erreichen oder nachdem sie empfangen wurden. Das ermöglicht z. B.:

- das **Mocken von Backend-Antworten**
- das **Blockieren von Requests** (z. B. zu externen APIs)

- das **Simulieren von Fehlerzuständen** (z. B. 500er-Fehler)

Beispiel:

```
await page.route('**/api/user', route => {
  route.fulfill({ status: 200, body: JSON.stringify({ name: 'Testuser' }) });
});
```

Screenshot

Playwright kann jederzeit einen Screenshot vom aktuellen Zustand der Seite aufnehmen – z. B. zur **Fehlerdokumentation**, zur visuellen Regression oder einfach als Debugging-Hilfe. Das ist besonders nützlich bei CI-Testläufen, wenn ein Test fehlschlägt.

Beispiel:

```
await page.screenshot({ path: 'screenshot.png', fullPage: true });
```

Tracing

Das **Tracing-System von Playwright** erstellt eine vollständige Aufzeichnung eines Testlaufs. Die generierte Datei (`trace.zip`) kann mit dem Trace Viewer analysiert werden. Dieser zeigt:

- DOM-Zustände und Screenshots nach jeder Aktion
- Klickpfade und Zeitverlauf
- Netzwerkanfragen und -antworten
- Konsolenausgaben und Fehler

Der Trace Viewer eignet sich ideal zum Debuggen, wenn Tests unerwartet fehlschlagen oder das Timing von UI-Interaktionen unklar ist.

Verwendung:

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Das Tool ist interaktiv und CI-fähig – es gehört zu den zentralen Debugging-Funktionen von Playwright.

HTML-Report

Playwright enthält standardmäßig einen HTML-Reporter, der nach einem Testlauf einen interaktiven Bericht erzeugen kann.

Verwendung:

```
npx playwright test --reporter=html
npx playwright show-report
```

Dabei entsteht ein Verzeichnis `playwright-report/`, in dem die Testergebnisse als durchklickbarer HTML-Bericht gespeichert werden. Der Report enthält:

- Status jedes Tests (✔ / ✖)
- Detaillierte Fehlerbeschreibung (Stacktrace)
- Screenshots bei Fehlern
- Trace-Links, sofern aktiviert

Hinweis: Reporter lassen sich auch dauerhaft in `playwright.config.ts` setzen:

```
reporter: ['list', 'html']
```

Der HTML-Report eignet sich besonders gut für Reviews, Testprotokolle oder zur Weitergabe an Stakeholder.

Das **Tracing-System von Playwright** erstellt eine vollständige Aufzeichnung eines Testlaufs. Die generierte Datei (`trace.zip`) kann mit dem Trace Viewer analysiert werden. Dieser zeigt:

- DOM-Zustände und Screenshots nach jeder Aktion
- Klickpfade und Zeitverlauf
- Netzwerkanfragen und -antworten
- Konsolenausgaben und Fehler

Der Trace Viewer eignet sich ideal zum Debuggen, wenn Tests unerwartet fehlschlagen oder das Timing von UI-Interaktionen unklar ist.

Verwendung:

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Das Tool ist interaktiv und CI-fähig – es gehört zu den zentralen Debugging-Funktionen von Playwright.

Das **Tracing-System von Playwright** erstellt einen kompletten „Film“ eines fehlgeschlagenen Testlaufs: Es enthält Informationen über:

- alle DOM-Änderungen
- Netzwerk-Anfragen
- Konsolenmeldungen
- Screenshots und Zeitleisten

Diese Daten können im **Playwright Trace Viewer** visuell nachvollzogen werden – ideal zum Debuggen komplexer Fehler in CI/CD.

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Debugging

Zusätzlich zum Tracing bietet Playwright u. a.:

- einen **Inspector** (GUI zum „Step-by-Step-Debuggen" im Browser)
- `codegen` zur **Aufzeichnung von Tests**
- **Pause-Funktion** für interaktives Testen (`page.pause()`)

Beispiel:

```
await page.pause(); // öffnet Debugger-Fenster mit DOM-Explorer
```

Integration in moderne Frameworks

Ein großer Pluspunkt ist die **nahtlose Integration in moderne Toolchains**. Bei der Installation eines SvelteKit-Projekts wird Playwright beispielsweise direkt mit angeboten. Es ist kein manueller Konfigurationsaufwand nötig.

Typische Teststruktur

Playwright kann Tests in mehreren Browsern ausführen – etwa in Chromium, Firefox oder WebKit. Wird in der `playwright.config.ts` jedoch kein `projects`-Abschnitt definiert, wie z. B. im SvelteKit-Starter, verwendet Playwright **standardmäßig nur Chromium**. Das reicht für viele Szenarien – schränkt aber die Cross-Browser-Testbarkeit ein.

Möchte man Tests zusätzlich in Firefox und WebKit durchführen, lässt sich die Konfiguration um `projects` erweitern:

```
projects: [
  { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
  { name: 'firefox',  use: { ...devices['Desktop Firefox'] } },
  { name: 'webkit',   use: { ...devices['Desktop Safari'] } },
]
```

Playwright führt die Tests dann automatisch für alle angegebenen Browser durch – entweder nacheinander oder parallel, je nach Konfiguration. So lassen sich UI-Komponenten und Workflows direkt in allen wichtigen Rendering-Engines verifizieren.

Playwright führt die Tests dann automatisch für alle angegebenen Browser durch – entweder nacheinander oder parallel, je nach Konfiguration. So lassen sich UI-Komponenten und Workflows direkt in allen wichtigen Rendering-Engines verifizieren.

Playwright-Konfigurationen werden in der Datei `playwright.config.ts` definiert. Besonders praktisch ist dabei die Möglichkeit, den lokalen Server automatisch zu starten, bevor die Tests ausgeführt werden:

```
import { defineConfig } from '@playwright/test';


export default defineConfig({
  webServer: {
    command: 'npm run build && npm run preview',
    port: 4173
  },
  testDir: 'e2e'
});
```

- **`command`** startet den lokalen Server, z. B. den SvelteKit-Preview-Modus.
- **`port`** legt fest, wo der Test-Runner auf die Anwendung wartet.
- **`testDir`** gibt das Verzeichnis an, in dem die Tests gespeichert sind.

Diese automatische Startlogik ist besonders hilfreich für CI/CD, da kein separater Serverlauf erforderlich ist. Wer möchte, kann hier auch eigene Kommandos oder Ports verwenden – etwa um eine andere Umgebung zu testen oder eine API-Instanz parallel zu starten.

Playwright sucht standardmäßig in dem Ordner nach Tests, in dem sich deine `playwright.config.ts` befindet. Zudem erkennt es von sich aus alle Dateien mit den Endungen `.spec.ts`, `.test.ts`, `.spec.js`, `.test.js`.

Viele Framework-Vorlagen – z. B. SvelteKit – legen hierfür das Verzeichnis **`e2e/`** an. Dies ist aber keine Vorgabe seitens Playwright, sondern eine Empfehlung, um **Frontend- und End-to-End-Tests (E2E)** strukturiert zu trennen.

Du kannst stattdessen auch beliebige andere Verzeichnisse nutzen (z. B. `tests/`, `ui-tests/`), solange sie sich im selben Verzeichnis wie `playwright.config.ts` befinden – oder du `testDir` in der Konfiguration entsprechend anpasst.

Nach dem Setup legt Playwright ein `e2e/`-Verzeichnis an (z. B. durch SvelteKit). Ein einfacher Test könnte so aussehen:

```
test('zeigt ein leeres Board mit 6x7 Zellen', async ({ page }) => {
  await page.goto('/');
  const cells = await page.locator('[data-test^=cell-]');
  await expect(cells).toHaveCount(6 * 7);
});
```

Häufig verwendete Befehle:

- `page.click(selector)` – Klick auf ein Element
- `page.fill(selector, text)` – Eingabe simulieren
- `expect(locator).toHaveText()` – Zustand prüfen
- `page.screenshot()` – Fehler sichtbar machen
- `test.beforeEach()` – Setup vor jedem Test

Nach dem Setup legt Playwright ein `e2e/`-Verzeichnis an. Ein einfacher Test könnte so aussehen:


```
test('zeigt ein leeres Board mit 6x7 Zellen', async ({ page }) => {
  await page.goto('/');
  const cells = await page.locator('[data-test^=cell-]');
  await expect(cells).toHaveCount(6 * 7);
});
```

Häufig verwendete Befehle:

- `page.click(selector)` – Klick auf ein Element
- `page.fill(selector, text)` – Eingabe simulieren
- `expect(locator).toHaveText()` – Zustand prüfen
- `page.screenshot()` – Fehler sichtbar machen
- `test.beforeEach()` – Setup vor jedem Test

Vergleich: Playwright, Cypress, Selenium

Merkmal	Playwright	Cypress	Selenium
Ausführungskontext	Außerhalb des Browsers über native DevTools-Protokolle	Im Browser	Extern via WebDriver
Sprachen	JS/TS, Python, C#, Java	Nur JS/TS	Viele (Java, Python, etc.)
Browser-Support	Chromium, Firefox, WebKit	Chromium-basiert, Firefox (beta)	Alle, inkl. IE
Multi-Tab, Multi-Window	Voll unterstützt	Eingeschränkt	Möglich, aber komplex
Netzwerk-Interception	Einfach via `page.route()`	Eingeschränkt	Aufwendig

Merkmal	Playwright	Cypress	Selenium
Debugging	Trace Viewer, Screenshots, CLI	Zeitreise im Browser, GUI	Schwerfällig
Setup-Aufwand	Minimal (bei modernen Frameworks)	Mittel	Hoch

Geeignete Einsatzzwecke

Playwright eignet sich besonders für:

- SPAs und dynamische Benutzeroberflächen
- Tests in mehreren Browsern (inkl. WebKit)
- CI/CD-Systeme mit paralleler Ausführung
- Komplexe User-Flows mit mehreren Tabs oder Netzwerkaktionen

Weniger geeignet ist Playwright für klassische Unit-Tests oder Alt-Systeme mit Internet Explorer.

Erweiterte Konzepte & Best Practices

1. Locator-Strategien

Robuste Selektoren sind entscheidend:

- `data-test="..."` als Attribut-Selektoren
- `getByRole('button', { name: 'Senden' })` für semantisch korrekte Auswahl
- CSS-Selektoren vermeiden

2. Teststruktur

- **Isolierte Tests:** Jeder Test startet mit `beforeEach()` in einer eigenen Umgebung.
- **Page Object Pattern:** UI-Interaktionen auslagern, um die Tests lesbar und wartbar zu halten.
- **Fixtures:** Wiederverwendbare Setup-Schritte in `test.extend()` kapseln.

3. Parallelisierung & Stabilität

Playwright führt Tests standardmäßig parallel aus, um die Laufzeit zu optimieren – lokal ebenso wie in CI/CD. Das bringt große Vorteile, erfordert aber, dass die Tests entsprechend gestaltet sind.

Grundprinzip:

> *Schreibe Tests so, als würden sie gleichzeitig laufen.*

Das bedeutet:

- Jeder Test sollte **unabhängig** von anderen Tests funktionieren.
- Es darf keine geteilten Daten, Zustände oder Seiteneffekte zwischen Tests geben.
- Testdaten sollten innerhalb des Tests erzeugt und genutzt werden.
- Falls Setup nötig ist: mit `beforeEach()` arbeiten, nicht mit geteilten Variablen.

Typisches Problem: Zwei Tests nutzen denselben Nutzer oder Datensatz – je nach Reihenfolge oder Timing schlägt einer davon fehl. Das wirkt dann wie ein „flaky test“ – ist aber ein Architekturproblem.

Hilfreiche Optionen in der Konfiguration oder beim Aufruf:

- `--shard=1/3` – teilt die Tests in gleichmäßige Gruppen für parallele Ausführung
- `--retries=2` – Wiederholung fehlgeschlagener Tests (z. B. bei Netzwerkaussetzern)
- `--workers=4` – legt explizit die Anzahl paralleler Worker fest

Fazit: Parallele Tests sparen Zeit – aber nur, wenn sie wirklich unabhängig voneinander sind. Genau das sollte beim Schreiben konsequent beachtet werden.

- `--shard`, `--retries`, `--workers` nutzen für schnelle, stabile CI-Runs.
- Traces (`--trace on`) für Fehleranalyse aktivieren.

4. Netzwerkinterception & Mocking

- Mit `page.route()` lassen sich APIs mocken oder blockieren.
- Reduziert Abhängigkeiten zu externen Diensten und beschleunigt die Tests.

Weiterführende Links

- Playwright Dokumentation
- Trace Viewer
- Playwright vs Cypress
- SvelteKit Testing Guide

Cloud-Apps