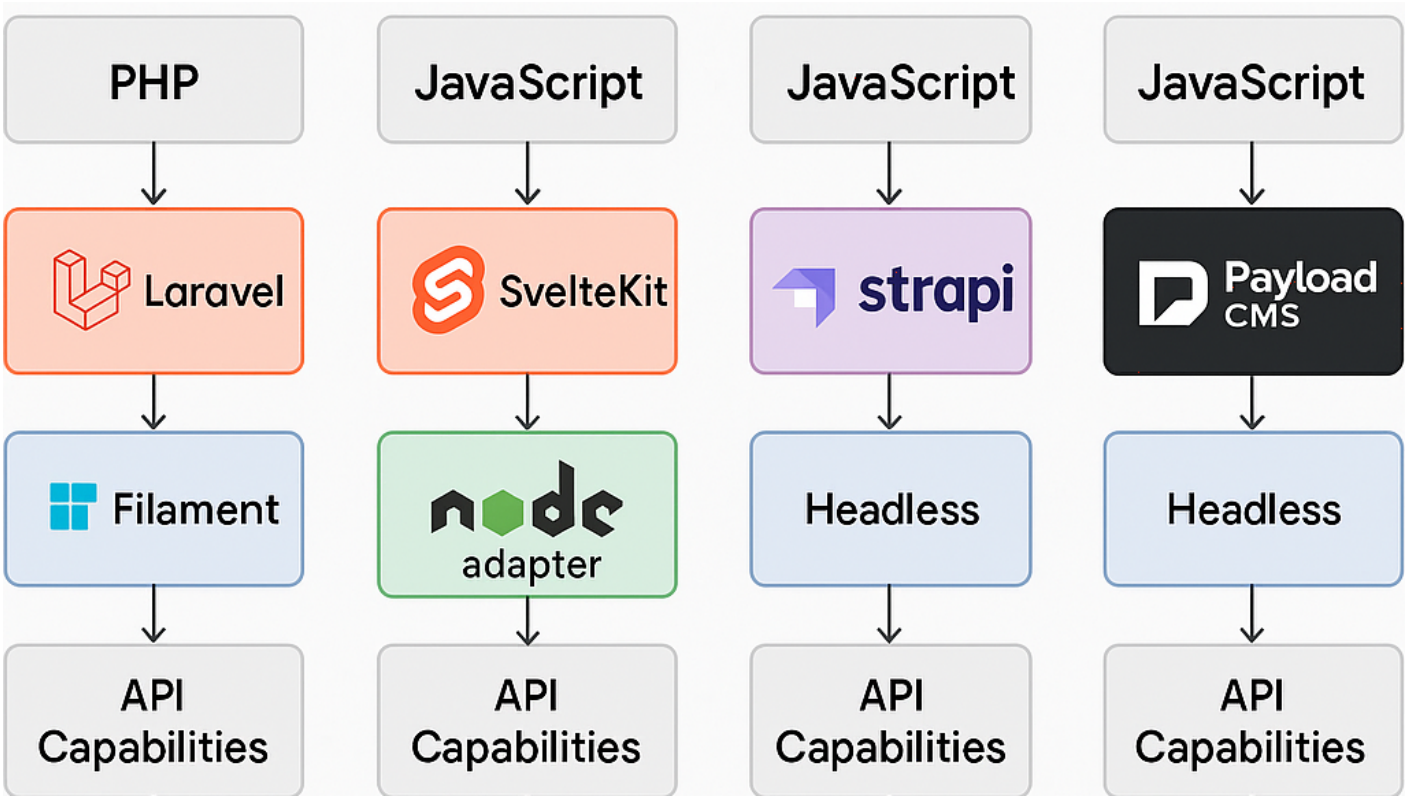
Backend-Technologien

- Mein Weg zu PayloadCMS: Backend-Vergleich aus der Praxis
- PayloadCMS
- Erweiterung des Beispielprojekts: Vier Gewinnt Remote

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

```
|     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

Datenbank	Standard?	ORM / Query Layer	Hinweise
PostgreSQL	□□ (ab 1.12+)	Kysely	Für produktive SQL-Setups

Erweiterung des Beispielprojekts: Vier Gewinnt Remote

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