Framework- und Technologie-Bewertung (Stand: März 2026)

Stack-Bewertung

Backend: PHP (plain, kein Framework)

Aktualität: PHP selbst ist mit Version 8.3/8.4 aktuell und performant — die Sprache hat sich stark verbessert (Fibers, Enums, readonly Properties, Match-Expressions). **Das Problem ist nicht PHP, sondern der fehlende Rahmen.**

Was fehlt:

- Kein Dependency Injection
- Kein Routing-System (nur switch-case in `api/index.php`)
- Kein ORM — handgeschriebene SQL-Queries überall
- Keine Typsicherheit auf Servicegrenzen
- `technical_data_service.php` mit 3500+ Zeilen ist ein klassischer God-Object-Antipattern
- Kein Autoloading (nur `require_once` Ketten)

Mit Laravel oder Symfony wäre dasselbe in strukturierter, testbarer, wartbarer Form möglich — gleiche Sprache, deutlich besser.

Frontend: Vanilla JS + globale Variablen

Das ist der kritischste schwache Punkt. Symptome:

- Globaler State (`selectedItemId`, `selectedStructureId` auf `window`)
- Kein Komponenten-Modell — UI-Logik verstreut über ~10 JS-Dateien
- Kein Bundler → kein Tree-Shaking, kein Caching, kein TypeScript
- Bootstrap 5 Modals als "SPA-Routing" — fragil bei Complexity-Wachstum

Moderner Ersatz: React, Vue 3, oder Svelte mit TypeScript und Vite. Der Unterschied bei einer komplexen Domäne (TD-Vererbung, Space-Kombinationen, Variant-Validierung) wäre enorm — zustandsbehaftete UI-Logik wäre nachvollziehbar, testbar, refaktorierbar.

Datenbank: MySQL

MySQL ist solide und für dieses Projekt völlig ausreichend. Dennoch:

Alternative	Vorteil hier
PostgreSQL	Besserer JSON-Support, echte Arrays, CTE-Rekursion für Hierarchien ist eleganter, strikter Typmodus, bessere Constraints
SQLite	Für Single-User/Embedded sinnvoll, aber hier nicht passend

Alternative	Vorteil hier
Relationale DB bleibt richtig	Das Datenmodell (Items, Strukturen, TD-Definitionen, Constraints) ist hochgradig relational

PostgreSQL wäre die natürliche Upgrade-Wahl — bessere Hierarchie-Queries (mit `WITH RECURSIVE`), JSONB für flexible TD-Werte, und strengere Datenintegrität.

Empfehlung: Moderner Ziel-Stack

Wenn man heute neu anfangen würde:

```
Backend:   Laravel 12 (PHP) oder FastAPI (Python) oder NestJS (TypeScript)
Frontend:  React 19 + TypeScript + Vite  oder  Vue 3 + TypeScript
Datenbank: PostgreSQL
ORM:       Eloquent (Laravel), SQLAlchemy, oder Prisma
```

Warum Laravel als realistischster Weg?

- **PHP bleibt** — kein kompletter Sprachenwechsel
- Eloquent ORM löst die SQL-Spaghetti
- Laravel hat eingebautes Routing, Middleware, Validation, Queue, Jobs
- Migrations-System ist professionell
- Sehr aktive Community, LTS-Support

Warum TypeScript/NestJS als beste Option insgesamt?

- **Ein Stack für Frontend und Backend** — kein Kontext-Switch
- NestJS bringt Angular-ähnliche Struktur (Module, Services, Guards, DTOs)
- Starke Typisierung durchgängig von DB (Prisma) bis Browser
- Perfekt für komplexe Domänenlogik wie TD-Vererbung

Zusammenfassung

Bereich	Jetzt	Empfehlung
Backend-Sprache	PHP 8.x (ok)	PHP (Laravel) oder TypeScript (NestJS)
Backend-Struktur	Kein Framework	Laravel / NestJS / FastAPI
Frontend	Vanilla JS, global state	React/Vue + TypeScript + Vite
Datenbank	MySQL	PostgreSQL
ORM	Kein ORM	Eloquent / Prisma / SQLAlchemy
Tests	Manuell	PHPUnit / Vitest / Pytest

Der größte Hebel wäre das Frontend. Die globale-Variable-Architektur wird bei wachsender Domänenkomplexität zum echten Wartungsproblem. Das Backend mit Laravel zu strukturieren wäre Schritt zwei. Die Datenbank zu PostgreSQL zu migrieren wäre Schritt drei und hat den geringsten Return on Investment im Verhältnis zum Aufwand.

Framework-Vergleich am konkreten Beispiel: `GET /items` und `POST /items`

Der folgende Vergleich zeigt denselben Endpunkt aus `api/controllers/items.php` — einmal im aktuellen Plain-PHP-Zustand, dann in allen drei Kandidaten-Frameworks.

Was der Endpunkt tut

- `GET /items?item_type=component&limit=20&offset=0` — gefilterte, paginierte Liste
- `POST /items` — neues Item anlegen, Pflichtfelder validieren, 201 zurückgeben

---

0. Aktueller Stand: Plain PHP

```
// api/index.php — Router
switch ($segments[0]) {
    case 'items':
        handleItems($db, $method, array_slice($segments, 1));
        break;
    // ... 15 weitere cases
}

// api/controllers/items.php
function handleItems(PDO $db, string $method, array $segments) {
    if (!$segments) {
        if ($method === 'GET') {
            $itemType    = $_GET['item_type'] ?? null;
            $itemGroupId = isset($_GET['item_group_id']) ?
(int)$_GET['item_group_id'] : null;
            $limit       = isset($_GET['limit']) ? min((int)$_GET['limit'], 500) :
null;
            $offset      = isset($_GET['offset']) ? max((int)$_GET['offset'], 0) :
0;

            $sql    = "SELECT i.*, ig.name AS item_group_name FROM items i LEFT
JOIN item_groups ig ON ig.id = i.item_group_id WHERE 1=1";
            $params = [];
            if ($itemType)    { $sql .= " AND i.item_type = :item_type";
$params[':item_type']    = $itemType; }
            if ($itemGroupId) { $sql .= " AND i.item_group_id = :item_group_id";
$params[':item_group_id'] = $itemGroupId; }
            $sql .= " ORDER BY i.item_type, ig.name, i.type";

            if ($limit !== null) {
                // separate COUNT-Query für Paginierung — duplizierter WHERE-Block
```

```
                $countStmt = $db->prepare("SELECT COUNT(*) FROM items i WHERE 1=1"
.
                    ($itemType    ? " AND i.item_type = :item_type"       : "") .
                    ($itemGroupId ? " AND i.item_group_id = :item_group_id" :
""));
                $countStmt->execute($params);
                $total = (int)$countStmt->fetchColumn();
                $sql  .= " LIMIT :limit OFFSET :offset";
                $stmt  = $db->prepare($sql);
                foreach ($params as $k => $v) $stmt->bindValue($k, $v);
                $stmt->bindValue(':limit',  $limit,  PDO::PARAM_INT);
                $stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
                $stmt->execute();
                echo json_encode(["data" => $stmt->fetchAll() ?: [], "total" =>
$total]);
            } else {
                $stmt = $db->prepare($sql);
                $stmt->execute($params);
                echo json_encode($stmt->fetchAll() ?: []);
            }
            return;
        }
        if ($method === 'POST') {
            $data = readJsonBody();                              //
helpers/http.php
            if (empty($data['item_type']) || empty($data['type'])) {
                badRequest("item_type and type are required"); //
helpers/http.php, exit()
            }
            $stmt = $db->prepare("INSERT INTO items (item_type, item_group_id,
type, class, description,
                configuration_required, parent_item_id, created_at, updated_at)
                VALUES (:item_type, :item_group_id, :type, :class, :description,
                :configuration_required, :parent_item_id, NOW(), NOW())");
            $stmt->execute([/* ... */]);
            http_response_code(201);
            echo json_encode(["id" => $db->lastInsertId()]);
            return;
        }
        methodNotAllowed();
    }
    $id = (int)$segments[0];
    // ... GET /{id}, PUT /{id}, DELETE /{id} ...
    // ... Sub-Ressourcen: /technicalData, /options, /space,
/aggregatedConstraints ...
}
```

Probleme auf einen Blick:

- Router ist ein einzelner `switch` mit 15 Cases — jede neue Route verlängert diese Datei
- SQL wird per String-Konkatenation dynamisch gebaut — der `WHERE`-Block wird für die COUNT-Query **copy-paste dupliziert**

- Validation = manuelle `if (empty(...))` + `exit()`
- HTTP-Status, Headers, JSON-Serialisierung — alles manuell, überall anders
- Die Funktion wächst mit jeder Sub-Ressource: aktuell 390 Zeilen in einer einzigen Funktion
- Kein Typ-System: `$data` kommt als `array|null` aus `readJsonBody()`, PHP weiß nicht was drin ist

---

1. Laravel 12 (PHP)

Laravel bleibt PHP — der Wechsel ist evolutionär, nicht revolutionär.

```
// routes/api.php — deklaratives Routing, keine Switch-Cases mehr
Route::apiResource('items', ItemController::class);
// Erzeugt automatisch: GET /, POST /, GET /{id}, PUT /{id}, DELETE /{id}

// app/Http/Requests/StoreItemRequest.php — Validation als eigene Klasse
class StoreItemRequest extends FormRequest {
    public function rules(): array {
        return [
            'item_type' => ['required', 'string', Rule::in(['product',
'component', 'sub_component'])],
            'type'      => ['required', 'string', 'max:255'],
            'class'     => ['nullable', 'string'],
            'description'           => ['nullable', 'string'],
            'configuration_required'=> ['boolean'],
            'item_group_id'         => ['nullable', 'integer',
'exists:item_groups,id'],
            'parent_item_id'        => ['nullable', 'integer', 'exists:items,id'],
        ];
    }
}

// app/Http/Controllers/ItemController.php
class ItemController extends Controller {

    public function index(Request $request): JsonResponse {
        $query = Item::with('itemGroup')                // Eloquent Relation, kein
JOIN von Hand
            ->when($request->item_type,    fn($q, $v) => $q->where('item_type',
$v))
            ->when($request->item_group_id,fn($q, $v) => $q-
>where('item_group_id', $v))
            ->orderBy('item_type')->orderBy('type');

        // Paginierung: ?limit=20 → paginate(20), sonst alle
        if ($request->filled('limit')) {
            return response()->json($query->paginate($request->limit));
            // Gibt automatisch: { data: [...], total: N, per_page: 20,
current_page: 1, ... }
        }
        return response()->json($query->get());
    }
```

```
    public function store(StoreItemRequest $request): JsonResponse {
        // $request->validated() enthält NUR die Felder aus rules() — kein
Overposting möglich
        $item = Item::create($request->validated());
        return response()->json(['id' => $item->id], 201);
    }

    public function show(Item $item): JsonResponse {
        // Route Model Binding: Laravel lädt Item::find($id) automatisch, gibt 404
wenn nicht gefunden
        return response()->json($item->load('itemGroup'));
    }

    public function update(StoreItemRequest $request, Item $item): JsonResponse {
        $item->update($request->validated());
        return response()->json(['status' => 'ok']);
    }

    public function destroy(Item $item): JsonResponse {
        try {
            $item->delete();
            return response()->json(['status' => 'ok']);
        } catch (QueryException $e) {
            if ($e->getCode() === '23000') {
                return response()->json(['error' => 'Item wird noch verwendet'],
409);
            }
            throw $e;
        }
    }
}

// app/Models/Item.php
class Item extends Model {
    protected $fillable = ['item_type', 'item_group_id', 'type', 'class',
                           'description', 'configuration_required',
'parent_item_id'];
    protected $casts    = ['configuration_required' => 'boolean', 'created_at' =>
'datetime'];

    public function itemGroup(): BelongsTo {
        return $this->belongsTo(ItemGroup::class);
    }
    public function technicalData(): HasMany {
        return $this->hasMany(ItemTechnicalData::class);
    }
}
```

Was Laravel hier löst:

Problem (Plain PHP)	Laravel-Lösung

Problem (Plain PHP)	Laravel-Lösung
Duplizierter SQL-WHERE für COUNT	`paginate()` macht beides intern in einer Methode
String-Konkatenation SQL	Eloquent Query Builder — typsicher, komponierbar
Manuelle Validation mit `exit()`	`FormRequest` — läuft vor dem Controller, wirft automatisch 422
390-Zeilen-Funktion	5 kleine Methoden, je 5–15 Zeilen
Route-Discovery per Switch	`apiResource()` — eine Zeile
Kein 404-Handling	Route Model Binding — automatisch

2. FastAPI (Python)

FastAPI nutzt Python-Typen als Contract — was du deklarierst, wird automatisch validiert **und** als OpenAPI-Doku generiert.

```
# schemas/item.py — Pydantic-Modell = Validation + Serialisierung + Dokumentation
in einem
from pydantic import BaseModel
from typing import Optional
from enum import Enum

class ItemType(str, Enum):
    product       = "product"
    component     = "component"
    sub_component = "sub_component"

class ItemCreate(BaseModel):
    item_type:               ItemType           # automatisch validiert gegen Enum
    type:                    str
    class_:                  Optional[str] = None   # Field-Alias für reserviertes
Wort
    description:             Optional[str] = None
    configuration_required:  bool = False
    item_group_id:           Optional[int] = None
    parent_item_id:          Optional[int] = None

    model_config = {"populate_by_name": True}

class ItemResponse(ItemCreate):
    id:              int
    item_group_name: Optional[str] = None

    model_config = {"from_attributes": True}   # erlaubt ORM-Objekte als Input

# routers/items.py
from fastapi import APIRouter, Depends, HTTPException, Query
from sqlalchemy.orm import Session
from typing import Optional
```

```python
router = APIRouter(prefix="/items", tags=["Items"])

@router.get("/", response_model=list[ItemResponse] |
PaginatedResponse[ItemResponse])
def list_items(
    item_type:    Optional[ItemType] = Query(None),   # Query-Parameter mit Typ
    item_group_id:Optional[int]      = Query(None),
    limit:        Optional[int]       = Query(None, le=500),  # le=500: max 500
    offset:       int                 = Query(0,    ge=0),    # ge=0: min 0
    db:           Session             = Depends(get_db),      # Dependency
Injection
):
    query = db.query(Item).join(ItemGroup, isouter=True)

    if item_type:     query = query.filter(Item.item_type == item_type)
    if item_group_id: query = query.filter(Item.item_group_id == item_group_id)

    query = query.order_by(Item.item_type, Item.type)

    if limit is not None:
        total = query.count()
        items = query.offset(offset).limit(limit).all()
        return {"data": items, "total": total, "limit": limit, "offset": offset}

    return query.all()

@router.post("/", response_model=ItemResponse, status_code=201)
def create_item(payload: ItemCreate, db: Session = Depends(get_db)):
    # payload ist bereits validiert – wenn item_type kein gültiger Enum-Wert ist,
    # hat FastAPI bereits 422 zurückgegeben, bevor diese Zeile erreicht wird
    item = Item(**payload.model_dump(by_alias=True))
    db.add(item)
    db.commit()
    db.refresh(item)
    return item

@router.get("/{item_id}", response_model=ItemResponse)
def get_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if not item:
        raise HTTPException(status_code=404, detail="Item nicht gefunden")
    return item

@router.delete("/{item_id}", status_code=204)
def delete_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if not item:
        raise HTTPException(status_code=404, detail="Item nicht gefunden")
    try:
        db.delete(item)
        db.commit()
    except IntegrityError:
        raise HTTPException(status_code=409, detail="Item wird noch verwendet")
```

Was FastAPI zusätzlich bringt:

Feature	Beschreibung
Automatische API-Doku	`/docs` (Swagger UI) und `/redoc` werden aus den Pydantic-Typen generiert — ohne Zusatzaufwand
Async-Support	`async def list_items(...)` — nicht-blockierende DB-Queries möglich (mit asyncpg)
Typen als einzige Quelle der Wahrheit	`ItemCreate` definiert Validation, Serialisierung und Doku gleichzeitig
`Query(None, le=500)`	Constraints direkt im Funktions-Signature — kein separates Validierungs-IF mehr
Dependency Injection	`Depends(get_db)` — DB-Session wird injiziert, testbar durch Überschreiben

Nachteil gegenüber Laravel: Python ist eine andere Sprache — kein Wiederverwendung von PHP-Kenntnissen. Mehr Konfigurationsaufwand für Deployment (kein XAMPP, braucht ASGI-Server wie uvicorn/gunicorn).

---

3. NestJS (TypeScript)

NestJS ist die strukturierteste Option. Alles ist explizit getypt — von der HTTP-Schicht bis zur Datenbank.

```
// items/dto/create-item.dto.ts — Data Transfer Object
import {
  IsEnum,
  IsString,
  IsOptional,
  IsBoolean,
  IsInt,
} from "class-validator";
import { Transform } from "class-transformer";

export enum ItemType {
  Product = "product",
  Component = "component",
  SubComponent = "sub_component",
}

export class CreateItemDto {
  @IsEnum(ItemType)
  item_type: ItemType;

  @IsString()
  type: string;

  @IsOptional()
```

```
  @IsString()
  class?: string;

  @IsOptional()
  @IsString()
  description?: string;

  @IsOptional()
  @IsBoolean()
  configuration_required?: boolean;

  @IsOptional()
  @IsInt()
  item_group_id?: number;

  @IsOptional()
  @IsInt()
  parent_item_id?: number;
}

// items/items.controller.ts
@Controller("items")
export class ItemsController {
  constructor(private readonly itemsService: ItemsService) {} // Dependency
Injection

  @Get()
  findAll(
    @Query("item_type") itemType?: ItemType,
    @Query("item_group_id", new ParseIntPipe({ optional: true }))
    itemGroupId?: number,
    @Query("limit", new ParseIntPipe({ optional: true })) limit?: number,
    @Query("offset", new ParseIntPipe({ optional: true })) offset?: number,
  ) {
    return this.itemsService.findAll({ itemType, itemGroupId, limit, offset });
  }

  @Post()
  @HttpCode(201)
  create(@Body() dto: CreateItemDto) {
    // NestJS hat ValidationPipe global registriert → dto ist garantiert valid
    // Wenn item_type kein gültiger Enum-Wert → 400 automatisch, bevor diese Zeile
läuft
    return this.itemsService.create(dto);
  }

  @Get(":id")
  findOne(@Param("id", ParseIntPipe) id: number) {
    return this.itemsService.findOne(id); // Service wirft NotFoundException wenn
nicht gefunden
  }

  @Put(":id")
  update(@Param("id", ParseIntPipe) id: number, @Body() dto: CreateItemDto) {
```

```
    return this.itemsService.update(id, dto);
  }

  @Delete(":id")
  @HttpCode(204)
  remove(@Param("id", ParseIntPipe) id: number) {
    return this.itemsService.remove(id);
  }
}

// items/items.service.ts
@Injectable()
export class ItemsService {
  constructor(
    @InjectRepository(Item) private itemRepo: Repository<Item>, // TypeORM
  ) {}

  async findAll(filters: ItemFilters) {
    const qb = this.itemRepo
      .createQueryBuilder("i")
      .leftJoinAndSelect("i.itemGroup", "ig")
      .orderBy("i.item_type")
      .addOrderBy("i.type");

    if (filters.itemType)
      qb.andWhere("i.item_type = :t", { t: filters.itemType });
    if (filters.itemGroupId)
      qb.andWhere("i.item_group_id = :g", { g: filters.itemGroupId });

    if (filters.limit != null) {
      const [data, total] = await qb
        .skip(filters.offset ?? 0)
        .take(filters.limit)
        .getManyAndCount();
      return { data, total, limit: filters.limit, offset: filters.offset ?? 0 };
    }
    return qb.getMany();
  }

  async create(dto: CreateItemDto): Promise<Item> {
    const item = this.itemRepo.create(dto);
    return this.itemRepo.save(item);
  }

  async findOne(id: number): Promise<Item> {
    const item = await this.itemRepo.findOne({
      where: { id },
      relations: ["itemGroup"],
    });
    if (!item) throw new NotFoundException(`Item ${id} nicht gefunden`);
    return item;
  }
}
```

```
// items/item.entity.ts — Datenbank-Schema als TypeScript-Klasse
@Entity("items")
export class Item {
  @PrimaryGeneratedColumn() id: number;
  @Column() item_type: ItemType;
  @Column({ nullable: true }) class: string | null;
  @Column() type: string;
  @Column({ nullable: true }) description: string | null;
  @Column({ default: false }) configuration_required: boolean;
  @CreateDateColumn() created_at: Date;
  @UpdateDateColumn() updated_at: Date;

  @ManyToOne(() => ItemGroup, { nullable: true })
  @JoinColumn({ name: "item_group_id" })
  itemGroup: ItemGroup | null;

  @OneToMany(() => ItemTechnicalData, (td) => td.item)
  technicalData: ItemTechnicalData[];
}
```

Was NestJS zusätzlich ermöglicht:

Feature	Beschreibung
Ein Typsystem für alles	`ItemType` Enum ist in DTO, Entity und Frontend-Code (shared types) identisch
`ParseIntPipe`	Query-Parameter werden automatisch in `number` konvertiert — kein `(int)$_GET[...]` mehr
`ValidationPipe` global	Jede Route ist durch das DTO abgesichert — kein einzelnes `if (empty(...))` mehr
Guards & Interceptors	Auth, Logging, Rate-Limiting als deklarative Decorator — kein Copy-Paste in jedem Controller
Migration von MySQL zu PostgreSQL	Nur die TypeORM-Datasource-Config ändern — alle Queries bleiben gleich

Direkter Vergleich der drei Frameworks

Kriterium	Laravel 12	FastAPI	NestJS
Lernkurve	Gering (PHP bleibt)	Mittel (neue Sprache)	Mittel-Hoch (neue Sprache + Struktur)
Typsicherheit	Mittel (PHP-Types + Larastan)	Hoch (Pydantic + mypy)	Sehr hoch (TypeScript end-to-end)
Automatische Doku	Nein (Scramble-Package nötig)	Ja (eingebaut, OpenAPI)	Ja (Swagger-Plugin)

Kriterium	Laravel 12	FastAPI	NestJS
ORM-Qualität	Sehr gut (Eloquent)	Gut (SQLAlchemy)	Gut (TypeORM/Prisma)
Frontend-Synergien	Keine	Keine	Hoch (shared TypeScript-Types)
Deployment auf XAMPP	Ja (mit Anpassungen)	Nein (Python-Server nötig)	Nein (Node.js-Server nötig)
Realistische Migration	Schrittweise möglich	Kompletter Neustart	Kompletter Neustart
Community/Docs	Sehr groß	Groß, wächst schnell	Groß

Empfehlung für EPIC

Kurzfristig (evolutionär): Laravel 12 — PHP bleibt, Code kann schrittweise portiert werden, XAMPP-Deployment mit geringen Anpassungen.

Langfristig (grüne Wiese): NestJS + React/Vue — ein Typsystem für gesamten Stack, kein Kontext-Switch zwischen Backend und Frontend, das komplexe Domänenmodell (TD-Vererbung, Space-Kombinationen, Constraint-Validierung) profitiert maximal von durchgängiger Typsicherheit.

Was ist ein ORM?

ORM = Object-Relational Mapper

Ein ORM ist eine Bibliothek, die Datenbank-Tabellen auf Klassen im Code abbildet — damit man nicht selbst SQL schreiben muss, sondern mit Objekten arbeitet.

Konkretes Beispiel aus EPIC

Ohne ORM (aktueller Stand in `api/controllers/items.php`):

```
$sql = "SELECT i.*, ig.name AS item_group_name
        FROM items i
        LEFT JOIN item_groups ig ON ig.id = i.item_group_id
        WHERE i.item_type = :item_type";
$stmt = $db->prepare($sql);
$stmt->execute([':item_type' => $itemType]);
$rows = $stmt->fetchAll();
// $rows ist ein rohes Array — PHP weiß nicht, was die Felder bedeuten
```

Mit ORM (Laravel Eloquent):

```
$items = Item::with('itemGroup')
    ->where('item_type', $itemType)
    ->get();
```

```
// $items ist eine Collection von Item-Objekten
// $item->itemGroup->name funktioniert direkt – der JOIN passiert automatisch
```

Was ein ORM konkret abnimmt

Aufgabe	Ohne ORM	Mit ORM
Datensatz laden	`SELECT` + `prepare` + `execute` + `fetch`	`Item::find($id)`
Relation laden	manueller `LEFT JOIN`	`$item->itemGroup` (lazy) oder `with('itemGroup')` (eager)
Einfügen	`INSERT INTO ...` mit allen Feldern	`Item::create($data)`
`updated_at` setzen	`updated_at = NOW()` in jedem UPDATE manuell	automatisch durch ORM
404-Handling	`if (!$row) notFound()` in jeder Funktion	automatisch via Route Model Binding

Die Kehrseite

Ein ORM versteckt SQL — das ist Stärke und Schwäche zugleich. Für einfache Queries ist es deutlich sauberer. Für komplexe Queries wie die aggregierten Constraints in EPIC (mit `GROUP_CONCAT`, rekursiven CTEs, verschachtelten Subqueries) schreibt man trotzdem Raw-SQL — aber nur dort wo es wirklich nötig ist, statt überall.

Sprachvergleich ohne Einschränkung: Rust, Go, .NET und die persönliche Empfehlung

Rust (Axum / Actix-web)

Rust ist die performanteste Option — und die ungeeignetste für dieses Projekt.

Stärken:

- Maximale Laufzeit-Performance, kein Garbage Collector
- Speichersicherheit zur Compile-Zeit garantiert
- Exzellent für System-Software, Embedded, hochfrequente APIs

Schwächen für EPIC:

- Kein ausgereiftes ORM: `sqlx` ist mächtig aber low-level, `Diesel` hat steile Lernkurve
- Borrow-Checker: für ein CRUD-lastiges Datenmodell wie EPIC ist der kognitive Overhead unverhältnismäßig
- Lernkurve ist die steilste aller Optionen — Monate, nicht Wochen
- Community für Web-Backends existiert, ist aber deutlich kleiner als Laravel/NestJS

Fazit: Rust ist die richtige Wahl für Datenbank-Engines, Betriebssysteme und Performance-kritische Services. Für eine Verwaltungsanwendung mit komplexem Datenmodell ist der Aufwand nicht gerechtfertigt.

Go (Gin / Echo / Fiber)

Go ist der "vernünftige" Performancekompromiss — einfacher als Rust, schneller als Node/PHP.

Stärken:

- Sehr schnell, kleine Binaries, einfaches Deployment (ein einziges Binary)
- Klare, lesbare Sprache — wenig Magic, wenig Abstraktion
- Sehr gute Concurrency (Goroutines) — gut für parallele TD-Aggregation
- `sqlc` generiert typsicheren Go-Code direkt aus SQL — interessanter Ansatz

```
// router/items.go — Go mit Gin
func (h *ItemHandler) ListItems(c *gin.Context) {
    itemType    := c.Query("item_type")
    itemGroupID := c.Query("item_group_id")

    items, err := h.service.FindAll(c.Request.Context(), ItemFilter{
        ItemType:    itemType,
        ItemGroupID: itemGroupID,
    })
    if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }
    c.JSON(200, items)
}

// service/items.go — explizite Fehlerbehandlung, kein Magic
func (s *ItemService) FindAll(ctx context.Context, f ItemFilter) ([]Item, error) {
    query := `SELECT i.*, ig.name AS item_group_name
              FROM items i LEFT JOIN item_groups ig ON ig.id = i.item_group_id
              WHERE 1=1`
    args := []any{}

    if f.ItemType != "" {
        query += " AND i.item_type = $1"
        args = append(args, f.ItemType)
    }
    // ... PostgreSQL-Parameter-Nummerierung ($1, $2, ...) statt :name
    rows, err := s.db.QueryContext(ctx, query, args...)
    if err != nil {
        return nil, fmt.Errorf("FindAll: %w", err)
    }
    // ...
}
```

Schwächen für EPIC:

- Kein vollwertiges ORM — man schreibt mehr SQL-Boilerplate als in Laravel oder NestJS
- Kein Ökosystem für Admin-UIs, Form-Handling, Validation wie Laravel es bietet
- Kein Frontend-Synergie (andere Sprache als TypeScript)
- Fehlerbehandlung ist explizit und ausführlich — `if err != nil` nach jeder Operation

Fazit: Go wäre eine solide Wahl wenn Performance ein Thema ist und man eine saubere, pragmatische API-Schicht ohne viel Magic will. Für EPIC mit seinem komplexen Datenmodell bringt Go wenig Vorteile gegenüber NestJS, dafür mehr Boilerplate.

.NET (ASP.NET Core / Minimal API)

.NET ist die Enterprise-Option — ausgereift, vollständig, opinionated.

Stärken:

- Entity Framework Core ist das ausgefeilteste ORM in diesem Vergleich
- Starke Typsicherheit durch C# — vergleichbar mit TypeScript, aber mit mehr Laufzeit-Garantien
- Exzellente Tooling-Integration (Visual Studio, Rider)
- LINQ macht komplexe Queries ausdrucksstark:

```
// ItemsController.cs
[HttpGet]
public async Task<IActionResult> GetItems(
    [FromQuery] string? itemType,
    [FromQuery] int? itemGroupId,
    [FromQuery] int? limit,
    [FromQuery] int offset = 0)
{
    var query = _db.Items
        .Include(i => i.ItemGroup)
        .AsQueryable();

    if (itemType    != null) query = query.Where(i => i.ItemType    == itemType);
    if (itemGroupId != null) query = query.Where(i => i.ItemGroupId ==
itemGroupId);

    query = query.OrderBy(i => i.ItemType).ThenBy(i => i.Type);

    if (limit.HasValue) {
        var total = await query.CountAsync();
        var data  = await query.Skip(offset).Take(limit.Value).ToListAsync();
        return Ok(new { data, total, limit, offset });
    }
    return Ok(await query.ToListAsync());
}

[HttpPost]
public async Task<IActionResult> CreateItem([FromBody] CreateItemDto dto)
{
    // ModelState-Validation automatisch durch [ApiController]-Attribut
    var item = _mapper.Map<Item>(dto);
```

```
    _db.Items.Add(item);
    await _db.SaveChangesAsync();
    return CreatedAtAction(nameof(GetItem), new { id = item.Id }, new { id =
item.Id });
}
```

- `WITH RECURSIVE` für die Hierarchie-Queries in EPIC wäre in EF Core elegant lösbar
- Swagger/OpenAPI ist eingebaut

Schwächen für EPIC:

- Windows-affin, aber Cross-Platform (Linux/Docker) ist inzwischen vollwertig unterstützt
- Schwergewichtig — für eine interne Verwaltungsanwendung viel Framework-Overhead
- Lizenz-Modell (obwohl .NET selbst Open Source ist, ist das Microsoft-Ökosystem eng)
- Kein Frontend-Synergie (C# ≠ TypeScript, außer bei Blazor — aber das ist ein eigenes Thema)

Fazit: .NET ist die stärkste Enterprise-Option, vor allem wenn das Team bereits C#-Erfahrung hat. Für EPIC ohne C#-Background unnötig schwer.

Persönliche Empfehlung (Sprache egal, grüne Wiese)

Wahl: NestJS (TypeScript) + React + PostgreSQL

Begründung speziell für EPIC:

1. Das Domänenmodell ist komplex — Typsicherheit zahlt sich direkt aus. TD-Vererbung, Space-Kombinationen, Constraint-Validierung — das sind Algorithmen mit vielen Verzweigungen und Datentypen. In TypeScript schlägt der Compiler jeden Fehler an, der in Plain PHP erst zur Laufzeit auftaucht. Das ist bei einem Datenmodell dieser Komplexität kein Luxus, sondern Notwendigkeit.

2. Frontend und Backend teilen denselben Stack. `ItemType`, `ConstraintScope`, `SpaceCombination` — diese Typen existieren heute doppelt: einmal als PHP-String, einmal als JS-Kommentar. In einem TypeScript-Monorepo sind sie einmal definiert und in beiden Schichten automatisch synchron.

3. NestJS ist strukturell am ähnlichsten zum aktuellen Code. Controller → Service → Repository entspricht genau der aktuellen Aufteilung (controllers/ → services/ → PDO). Die Migration wäre konzeptuell verständlich, nicht ein kompletter Paradigmenwechsel.

4. Prisma als ORM ist ideal für dieses Schema. Prisma generiert den gesamten TypeScript-Client aus dem Datenbankschema — inklusive typisierter Queries für die komplexen Relationen (Items → Structures → TechnicalData → Definitions → Units).

```
Empfohlener Ziel-Stack:
├── Backend:    NestJS 11 + Prisma ORM
├── Frontend:   React 19 + TypeScript + Vite + TanStack Query
├── Datenbank:  PostgreSQL 17
├── Shared:     TypeScript-Typen als eigenes Package (monorepo)
└── Tooling:    pnpm workspaces, Vitest, Playwright (E2E)
```

Warum nicht Rust oder Go? Beide sind für EPIC überqualifiziert in Richtung Performance und unterqualifiziert in Richtung Produktivität. EPIC ist kein High-Traffic-System — es ist eine komplexe Domänenlogik-Anwendung. Da zählt Ausdrucksstärke und Typsicherheit mehr als Nanosekunden.

Warum nicht .NET? .NET wäre die zweitbeste Wahl, insbesondere wenn C#-Kenntnisse vorhanden wären. Entity Framework Core ist das ausgefeilteste ORM, LINQ ist für Hierarchie-Queries elegant. Aber ohne bestehende C#-Basis fehlt der entscheidende Vorteil gegenüber NestJS.

Ideale Deployment-Kette

Aktueller Stand: XAMPP / manuell

```
Entwickler bearbeitet Datei in Notepad++
         ↓
Datei liegt direkt in D:\xampp\htdocs\EPIC\
         ↓
Apache liest die PHP-Datei bei jedem Request neu
         ↓
Fertig — kein Build, kein Deploy, kein Prozess
```

Vorteil: Sofort sichtbar, kein Overhead. **Nachteil:** Kein Versionskontrolle-Workflow, kein Staging, kein Rollback, Produktion = Entwicklungsmaschine.

Moderne Deployment-Kette (allgemeines Prinzip)

```
Code schreiben (lokale Entwicklungsumgebung)
         ↓
git commit + git push → GitHub / GitLab
         ↓
CI-Pipeline startet automatisch (GitHub Actions / GitLab CI)
    ├── Tests laufen (Unit + Integration)
    ├── Linter / Typecheck
    └── Build (nur bei Bedarf: JS, Docker-Image)
         ↓
Wenn alles grün: automatisches Deploy auf Staging
         ↓
Manuelle Freigabe → Deploy auf Produktion
         ↓
Monitoring / Logging (Fehler werden gemeldet)
```

Laravel 12 — Deployment-Kette

Lokal:

```
# Abhängigkeiten installieren
composer install

# Umgebungsvariablen
cp .env.example .env
php artisan key:generate

# Datenbank vorbereiten
php artisan migrate

# Entwicklungsserver starten
php artisan serve       # http://localhost:8000
```

CI/CD (GitHub Actions):

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]

jobs:
  test-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: PHP-Abhängigkeiten installieren
        run: composer install --no-dev --optimize-autoloader

      - name: Tests ausführen
        run: php artisan test

      - name: Deploy via SSH
        run: |
          ssh user@server "cd /var/www/epic && git pull && composer install --no-
dev && php artisan migrate --force && php artisan config:cache"
```

Produktion (klassischer Webserver):

```
Webserver:     Apache oder Nginx (kein XAMPP nötig)
PHP:           PHP-FPM (schneller als mod_php)
Datenbank:     PostgreSQL auf demselben oder separatem Server
Prozessmanager: systemd oder Supervisor (für Queue-Worker)
```

```
Anfrage → Nginx → PHP-FPM → Laravel → PostgreSQL
```

Vorteil gegenüber XAMPP: Laravel kann auf jedem Linux-Server deployed werden — inkl. günstiger VPS wie Hetzner (3–5 €/Monat). Kein Windows-Server nötig. Laravel Forge automatisiert die Server-Einrichtung komplett.

FastAPI — Deployment-Kette

Lokal:

```
# Virtuelle Umgebung
python -m venv .venv
source .venv/bin/activate          # Linux/Mac
.venv\Scripts\activate             # Windows

# Abhängigkeiten
pip install -r requirements.txt

# Entwicklungsserver (mit Auto-Reload)
uvicorn app.main:app --reload      # http://localhost:8000
# Automatische API-Doku: http://localhost:8000/docs
```

Docker (bevorzugte Produktionsform für Python):

```
# Dockerfile
FROM python:3.13-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["gunicorn", "app.main:app", "-k", "uvicorn.workers.UvicornWorker", \
     "--bind", "0.0.0.0:8000", "--workers", "4"]
```

CI/CD:

```
jobs:
  deploy:
    steps:
      - name: Docker-Image bauen
        run: docker build -t epic-api:${{ github.sha }} .

      - name: Image zu Registry pushen
        run: docker push registry.example.com/epic-api:${{ github.sha }}

      - name: Auf Server deployen
```

```
        run: |
          ssh user@server "docker pull registry.example.com/epic-api:${{
github.sha }} && \
                           docker stop epic-api && \
                           docker run -d --name epic-api -p 8000:8000 epic-api:${{
github.sha }}"
```

Produktions-Stack:

```
Anfrage → Nginx (Reverse Proxy) → Gunicorn (4 Worker) → FastAPI → PostgreSQL
                                          ↓
                                   Docker Container
```

Besonderheit FastAPI: Die automatisch generierte Swagger-Doku (`/docs`) ist sofort im Browser nutzbar — kein Zusatzaufwand, kein Postman nötig. Ideal für interne APIs.

NestJS — Deployment-Kette

Lokal:

```
# Abhängigkeiten
npm install

# Entwicklungsserver (mit Hot-Reload)
npm run start:dev      # http://localhost:3000

# Datenbank-Migrations (Prisma)
npx prisma migrate dev
npx prisma studio      # Datenbank-Browser im Browser
```

Build:

```
npm run build          # TypeScript → JavaScript in /dist
npm run start:prod     # Startet den kompilierten Code
```

CI/CD (GitHub Actions):

```
jobs:
  build-and-deploy:
    steps:
      - run: npm ci
      - run: npm run lint
      - run: npm run test
      - run: npm run build
```

```
      - name: Deploy
        run: |
          rsync -av dist/ user@server:/var/www/epic/dist/
          ssh user@server "cd /var/www/epic && npm ci --omit=dev && \
                           npx prisma migrate deploy && \
                           pm2 restart epic-api"
```

Produktions-Stack:

```
Anfrage → Nginx → Node.js (via PM2) → NestJS → PostgreSQL
              ↑
         Reverse Proxy
         (SSL-Terminierung,
          statische Dateien)
```

PM2 (Process Manager für Node.js):

```
pm2 start dist/main.js --name epic-api --instances 4
pm2 save       # Startet automatisch nach Reboot
pm2 monit      # Live-Monitoring im Terminal
```

Monorepo mit Frontend (Vite + React):

```
├── apps/
│   ├── api/          ← NestJS Backend
│   └── web/          ← React Frontend (Vite)
├── packages/
│   └── types/        ← Geteilte TypeScript-Typen
└── package.json      ← pnpm workspaces
```

```
# Alles auf einmal starten (lokal)
pnpm run dev          # startet API + Frontend parallel

# Frontend bauen (statische Dateien → Nginx)
pnpm run build:web    # dist/web/ → Nginx bedient direkt
```

Vergleich der Deployment-Komplexität

	XAMPP (jetzt)	Laravel	FastAPI	NestJS

	XAMPP (jetzt)	Laravel	FastAPI	NestJS
Lokale Einrichtung	Sofort (Datei kopieren)	5 Minuten	5 Minuten	5 Minuten
Build-Schritt nötig	Nein	Nein (PHP ist interpretiert)	Nein	Ja (`tsc`)
Produktions-Server	Windows + XAMPP	Linux + Apache/Nginx + PHP-FPM	Linux + Docker	Linux + Node.js + PM2
CI/CD einrichten	Nicht vorhanden	1–2 Stunden	1–2 Stunden	1–2 Stunden
Zero-Downtime-Deploy	Nein (Datei-Überschreiben)	Ja (mit Deployer/Forge)	Ja (Docker rolling update)	Ja (PM2 cluster reload)
Rollback	Nein (kein Versionsstand)	`git checkout` + re-deploy	Docker-Image-Tag wechseln	`git checkout` + rebuild
Umgebungsvariablen	php.ini / hart kodiert	`.env` Datei (Laravel standard)	`.env` + Pydantic-Settings	`.env` + ConfigModule
Datenbankmigrationen	Manuelle SQL-Skripte	`php artisan migrate`	Alembic	`npx prisma migrate deploy`
Monitoring	Keins	Laravel Telescope / Sentry	Sentry + Prometheus	NestJS Logger / Sentry

Empfohlene Minimal-Kette für EPIC (realistisch)

Unabhängig vom Framework-Wechsel könnte der aktuelle Stand sofort verbessert werden:

```
1. Git-Workflow einführen
   git push → GitHub → automatischer Deploy via GitHub Actions

2. Staging-Umgebung (zweite XAMPP-Instanz oder kleiner VPS)
   Kein Deploy direkt auf Produktion mehr

3. Automatische Datenbankmigrationen
   migration_*.sql-Dateien → versioniert + automatisch ausgeführt

4. Environment-Variablen
   Datenbankpasswort + Konfiguration aus dem Code herauslösen (.env)

5. Fehler-Logging
   Sentry (kostenloser Plan) — Fehler werden gemeldet statt stillschweigend
geschluckt
```

Diese fünf Schritte sind **unabhängig vom Framework** und würden die Entwicklungs-Qualität sofort messbar verbessern — auch wenn EPIC weiterhin Plain PHP bleibt.