



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 06

SOFTWARE-ARCHITEKTUR — TEIL 1

Eure Sprints sind schnell... Aber kann euer System skalieren?

Ihr habt Agile und Scrum gelernt:

- Veränderungen durch kurze Iterationszyklen annehmen
- Funktionierende Software alle 1-4 Wochen liefern
- Arbeit mit GitHub Issues und Projects verfolgen

Euer Road Profile Viewer funktioniert super! CI ist grün. Product Owner ist zufrieden.

Dann fragt der Stakeholder:

"Können wir das für 1.000 gleichzeitige Nutzer in drei europäischen Büros bereitstellen?"

Das Problem: Architektur vs. Agilität

Plötzlich wird euch klar:

- Eure Dash-App läuft auf einem einzigen Laptop
- SQLite kann keine gleichzeitigen Schreibzugriffe von mehreren Standorten verarbeiten
- Die gesamte Anwendung ist ein Python-Prozess
- Es gibt keine Möglichkeit, Komponenten unabhängig zu skalieren

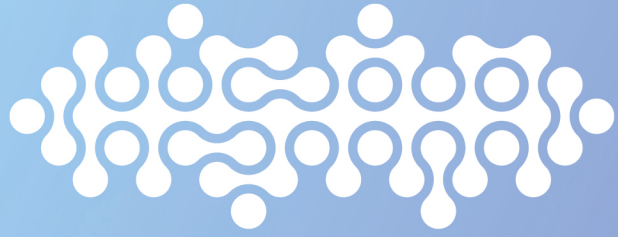
Der agile Prozess funktioniert. Die Architektur nicht.

Praxisbeispiel: Twitters Fail Whale

Twitter 2008-2010: Die monolithische Ruby on Rails-Anwendung konnte das explosive Nutzerwachstum nicht bewältigen.

Der "Fail Whale" wurde legendär — ein Symbol dafür, was passiert, wenn die Architektur den Erfolg nicht unterstützen kann.

Lektion: Auch erfolgreiche Produkte scheitern ohne skalierbare Architektur.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Grundlagen der Software- Architektur

1

Was ist Software-Architektur?

Software-Architektur beschreibt, wie ein System als Menge kommunizierender Komponenten organisiert ist und wie sich diese Komponenten verhalten und interagieren.

Architektur beantwortet:

- Wie werden Verantwortlichkeiten auf Komponenten verteilt?
- Wie kommunizieren Komponenten miteinander?
- Wie wird das System auf Hardware verteilt?
- Was passiert, wenn eine Komponente ausfällt?

Die Brücke: Agile trifft Architektur

Agile sagt uns:

- *Wie man Arbeit organisiert*
- Kurze Iterationszyklen
- Auf Veränderungen reagieren

Architektur sagt uns:

- *Wie man das System organisiert*
- Komponentenstruktur
- Kommunikationsmuster

Beides muss zusammenarbeiten. Agile Architektur *entwickelt sich* durch iterative Verfeinerung — genug Entscheidungen treffen um zu starten, dann verfeinern während man lernt.

Lernziele

Am Ende dieser Vorlesung (Teil 1) werdet ihr in der Lage sein:

1. **Motivation und Ziele** für Architekturentwurf zu benennen — warum Architektur wichtig ist und was ein Architekturmodell ermöglicht
2. **Die Abhängigkeit** zwischen Systemstruktur und nicht-funktionalen Anforderungen zu erklären — wie Performance, Sicherheit und Wartbarkeit Entscheidungen beeinflussen
3. **Das 4+1-Sichtenmodell** anzuwenden um ein System aus mehreren Perspektiven zu beschreiben — Logisch, Prozess, Entwicklung, Physisch und Szenarien

Architektur als kreativer Prozess

Anders als Algorithmen mit beweisbar optimalen Lösungen beinhaltet Architektur **Trade-offs**, die abhängen von:

- **Systemtyp:** Echtzeit-Embedded $\neq$ Webanwendung
- **Architekt-Erfahrung:** Was bei ähnlichen Systemen funktioniert hat (und gescheitert ist)
- **Spezifische Anforderungen:** Die einzigartigen Einschränkungen eures Projekts

*"Es gibt keinen formelhaften Architekturentwurfsprozess."
— Sommerville, Software Engineering*

Denkt an Architekturentwurf als eine Reihe von zu treffenden Entscheidungen.

Zentrale Entwurfsfragen

1 Gibt es eine generische Architekturvorlage?

2 Wie wird es auf Hardware verteilt?

3 Welche Architekturmuster könnten verwendet werden?

4 Was ist der grundlegende Strukturierungsansatz?

5 Wie werden Komponenten zerlegt?

6 Welche Strategie steuert den Komponentenbetrieb?

7 Welche Organisation liefert NFRs am besten?

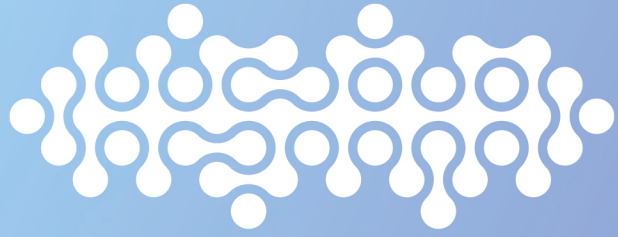
8 Wie soll die Architektur dokumentiert werden?

Think-Pair-Share: Road Profile Viewer

Frage	Road Profile Viewer Antwort
Vorlage?	Web-Dashboard-Anwendungen
Verteilung?	Aktuell Einzelrechner
Muster?	MVC für UI, Schichtenarchitektur
Zerlegung?	Trennung von Visualisierung, Datenzugriff, Geschäftslogik
Steuerung?	Request-Response für Web, Callbacks für Dash
NFR-Priorität?	Wartbarkeit vor Performance (Studentenprojekt)

Hinweis: MVC und Schichtenarchitektur werden in Teil 2 ausführlich erklärt.

Diskutiert mit eurem Nachbarn: Was würde sich bei 1.000 Nutzern ändern?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Nicht-funktionale Anforderungen

2

Nicht-funktionale Anforderungen bestimmen die Architektur

Architekturentscheidungen hängen stark von nicht-funktionalen Anforderungen ab und beeinflussen umgekehrt die nicht-funktionalen Eigenschaften des Systems.

NFR	Architektonische Implikation
Performance	Kritische Operationen lokalisieren; Netzwerk-Hops minimieren
Sicherheit	Schichtenarchitektur mit kritischen Assets im Innersten
Verfügbarkeit	Redundante Komponenten; Hot-Swapping ermöglichen
Wartbarkeit	Feinkörnige, eigenständige Komponenten

Performance-orientierte Architektur

Wenn Performance das Hauptanliegen ist:

Kernprinzipien:

1. Kritische Operationen lokalisieren
2. Netzwerk-Hops minimieren
3. Daten und Berechnung zusammen platzieren
4. Größere, monolithische Komponenten verwenden

Warum?

- Netzwerkaufruf: 1-100ms
- Lokaler Funktionsaufruf: Nanosekunden
- Weniger Komponenten = weniger Overhead

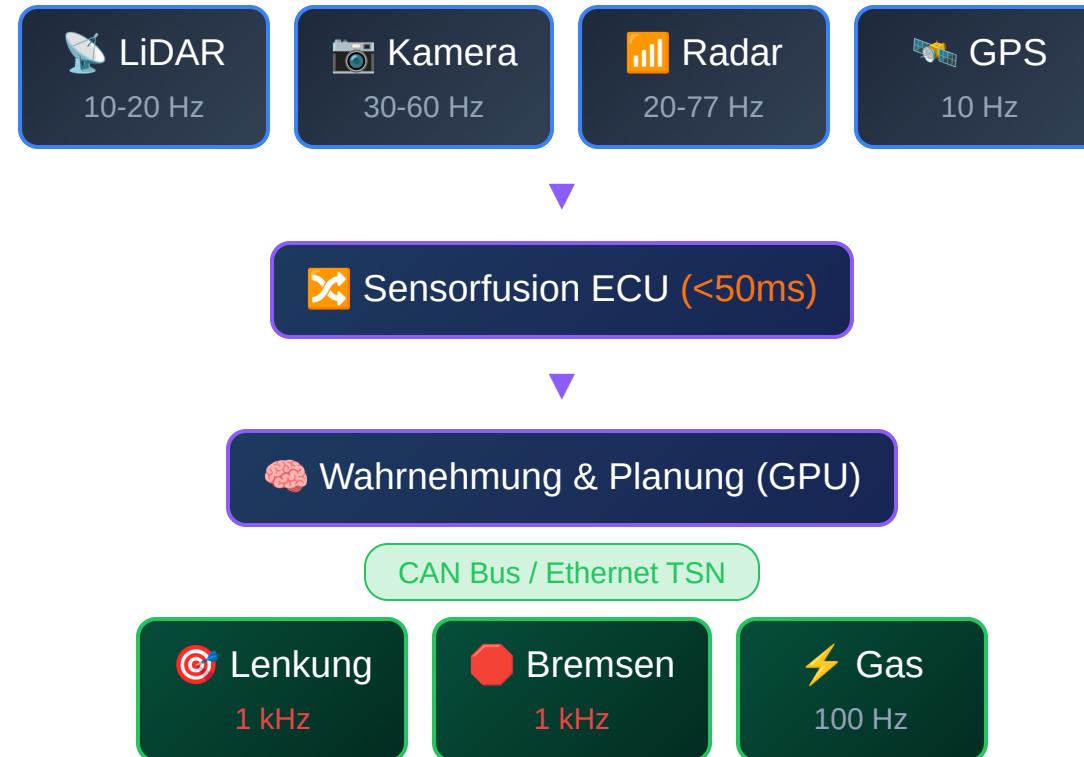
Trade-off: Performance-optimierte Architekturen opfern Modularität und Testbarkeit.

Echtzeitsysteme Beispiel

Systemtyp	Zeitanforderung	Architektonische Implikation
Harte Echtzeit	Deadlines dürfen <i>nie</i> verfehlt werden	Vorhersagbare Ausführung, kein GC, dedizierte Hardware
Weiche Echtzeit	Gelegentliche Verfehlungen akzeptabel	Prioritätsplanung, begrenzte Warteschlangen
Nahe Echtzeit	Schnelle Antwort (< 100ms)	Caching, asynchrone Verarbeitung, Load Balancing

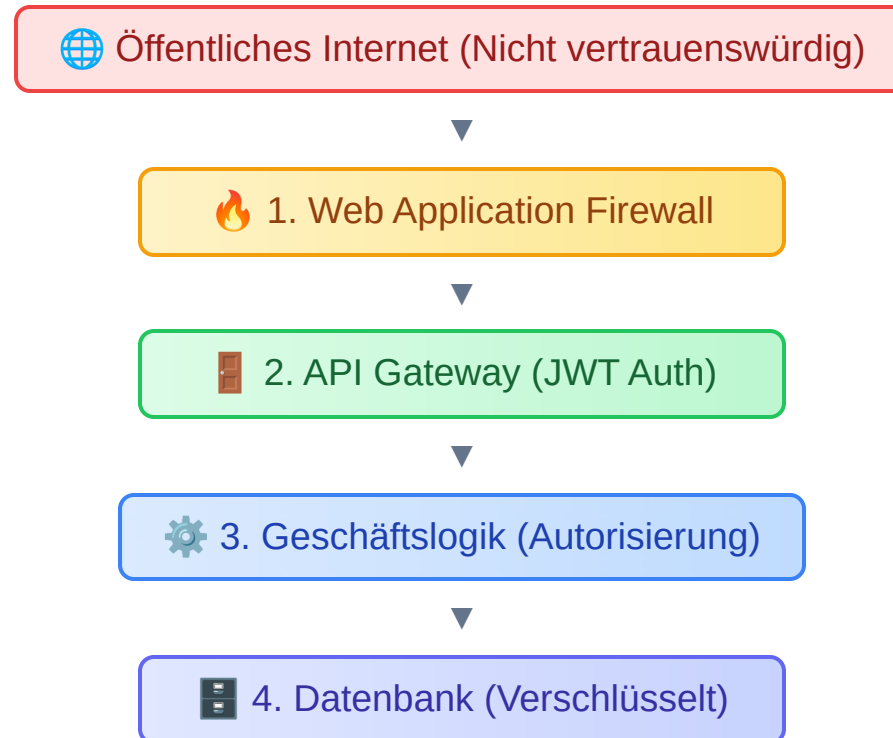
Beispiel: Autonomes Fahrzeug — Bremsen muss in < 1ms reagieren. HTTP-Microservices wären fatal!

Architektur autonomer Fahrzeuge



Sicherheitsorientierte Architektur

Defense in Depth: Mehrere Schutzschichten. Ein Angreifer muss ALLE Schichten durchbrechen.



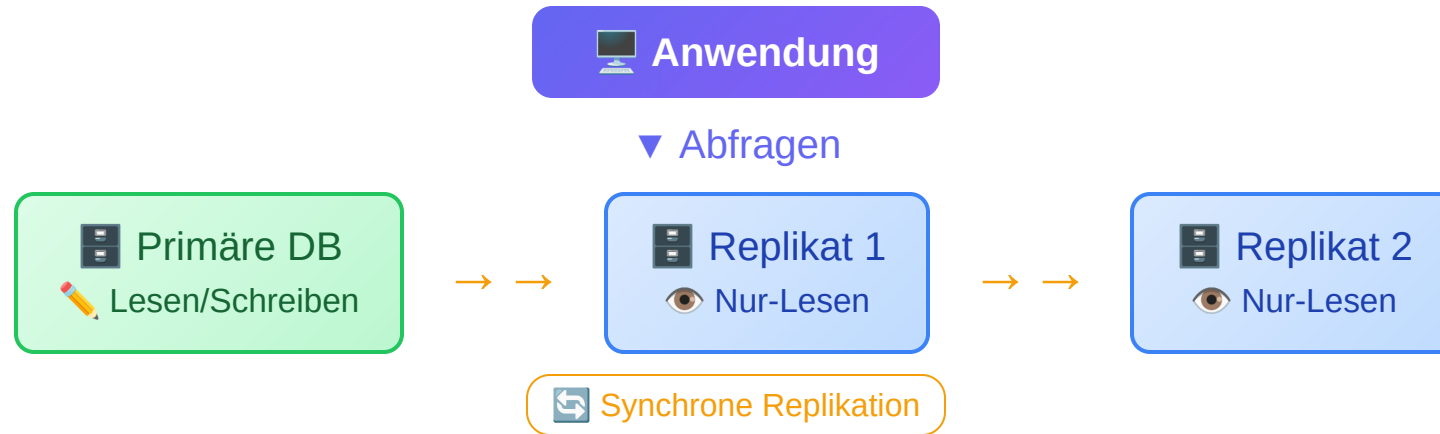
Verfügbarkeitsorientierte Architektur

Verfügbarkeit = Wie oft euer System betriebsbereit ist.

Ziel	Max. Ausfallzeit/Jahr	Anforderungen
99% ("zwei Neunen")	3,65 Tage	Grundlegendes Monitoring
99,9% ("drei Neunen")	8,76 Stunden	Redundante Komponenten, Auto-Failover
99,99% ("vier Neunen")	52,6 Minuten	Mehrere Rechenzentren
99,999% ("fünf Neunen")	5,26 Minuten	Aktiv-Aktiv-Replikation

Kernprinzip: Redundanz. Kein Single Point of Failure.

Datenbank-Failover-Muster



Bei Ausfall der Primären → Replikat kann befördert werden!

Netflix: Chaos Engineering

Netflix erfordert 99,99% Verfügbarkeit — nur 52 Minuten Ausfallzeit/Jahr.

Ihr Ansatz:

- **Mehrere AWS-Regionen:** Aktiv-Aktiv in US-East + US-West
- **Chaos Monkey:** Beendet zufällig Produktionsinstanzen
- **Chaos Gorilla:** Legt ganze Availability Zones lahm
- **Chaos Kong:** Simuliert kompletten Regionsausfall

"Wir konnten unsere Teams auf Infrastruktur-Resilienz ausrichten, indem wir Probleme isolierten und bis ins Extreme trieben."

Trade-off: Hohe Verfügbarkeit = teuer (Redundanz kostet Geld).

Wartbarkeitsorientierte Architektur

80% der Softwarekosten sind Wartung, nicht Erstentwicklung.

Kernprinzip: Separation of Concerns

Zeichen für SCHLECHTE Wartbarkeit:

- "God Class" mit 500+ Zeilen
- DB-Änderung erfordert UI-Anpassung
- Testen ohne Server-Start nicht möglich

Zeichen für GUTE Wartbarkeit:

- Jedes Modul hat einen Zweck
- Änderungen sind lokalisiert
- Komponenten einzeln testbar

Single Responsibility Principle (SRP)

```
# ✓ GUT: Getrennte Verantwortlichkeiten
class ProfileRepository:    # Nur: Datenzugriff
    def get(self, id): ...
    def save(self, profile): ...

class ProfileService:      # Nur: Geschäftslogik
    def validate_and_save(self, profile):
        self._validate(profile)
        self.repository.save(profile)

def create_profile_chart(profile): # Nur: Visualisierung
    fig, ax = plt.subplots()
    ax.plot(profile.distances, profile.elevations)
    return fig
```

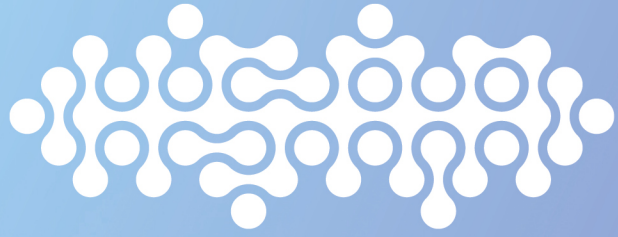
Jedes Modul hat genau EINEN Grund sich zu ändern.

Dependency Injection: Schlüssel zur Testbarkeit

```
# ✗ Schwer zu testen: Service erstellt eigene Abhängigkeiten
class ProfileService:
    def __init__(self):
        self.repository = ProfileRepository("production.db") # Hardcoded!

# ✓ Einfach zu testen: Abhängigkeiten werden injiziert
class ProfileService:
    def __init__(self, repository: ProfileRepository):
        self.repository = repository # Aufrufer entscheidet!

# In Tests:
mock_repo = MockProfileRepository() # Fake, keine echte DB
service = ProfileService(mock_repo)
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Architektur-Trade-offs

3

Trade-offs: Man kann nicht alles optimieren

Architekturziele stehen oft im Konflikt:

Ziel A	Konfligiert mit	Warum
Performance	Wartbarkeit	Große Komponenten schnell aber schwer änderbar
Sicherheit	Benutzerfreundlichkeit	Mehr Schichten = mehr Reibung
Verfügbarkeit	Kosten	Redundante Systeme sind teuer
Flexibilität	Einfachheit	Mehr Konfiguration = mehr Bugs

Umgang mit Trade-offs

- 1. Priorisieren:** Welche NFRs sind für EUER System am wichtigsten?
- 2. Partitionieren:** Verschiedene Architekturen für verschiedene Teile verwenden
 - Performance-kritischer Kern + wartbare Plugins
- 3. Dokumentieren:** Festhalten, WARUM ihr jeden Trade-off gemacht habt

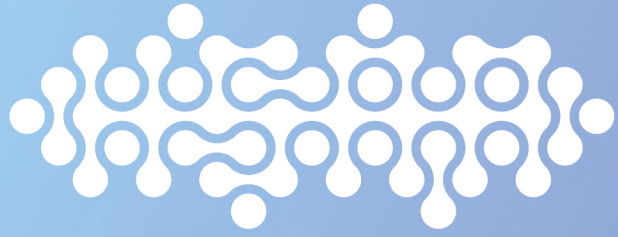
Für Road Profile Viewer: Wartbarkeit und Einfachheit > Performance-Optimierung.
Performance kann warten, bis ihr tatsächlich 1.000 Nutzer habt!

Kurze Umfrage: Eure Prioritäten

Welche NFR ist für eure Studentenprojekte am WICHTIGSTEN?

- A) **Performance** — muss schnell sein
- B) **Sicherheit** — muss Daten schützen
- C) **Verfügbarkeit** — muss immer erreichbar sein
- D) **Wartbarkeit** — muss einfach zu ändern sein

Diskutiert: Warum könnte die "richtige" Antwort für ein Startup vs. eine Bank vs. ein Spiel unterschiedlich sein?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Das 4+1-Sichtenmodell

4

Warum ein Diagramm nie ausreicht

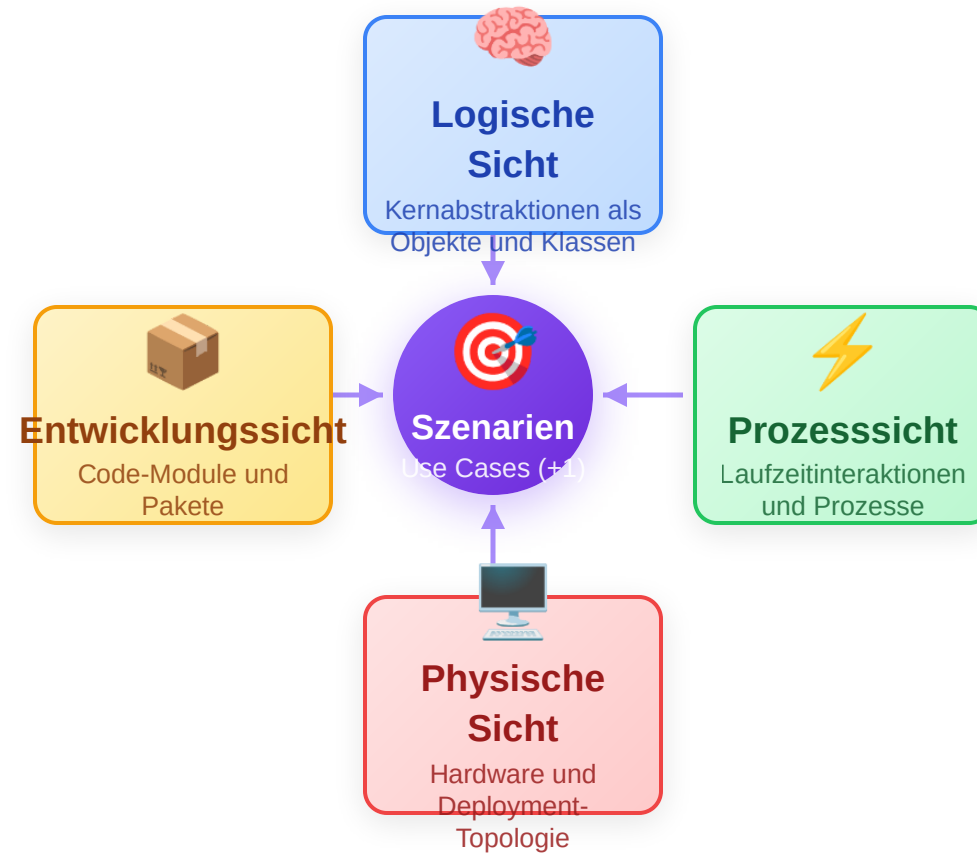
Verschiedene Stakeholder brauchen verschiedene Informationen:

- **Entwickler:** Welche Klassen existieren? In welchen Paketen sind sie?
- **Operations:** Welche Server betreiben welche Dienste? Wie kommunizieren sie?
- **Projektmanager:** Welche Teams verantworten welche Komponenten?
- **Sicherheitsprüfer:** Wohin fließen sensible Daten?

"Es ist unmöglich, alle relevanten Informationen über die Architektur eines Systems in einem einzigen Diagramm darzustellen."

— Sommerville

Das 4+1-Sichtenmodell



4+1 Sichten Übersicht

Sicht	Zeigt	Schlüsselfrage
Logisch	Objekte und Klassen	"Was sind die Hauptkonzepte?"
Prozess	Laufzeitinteraktionen	"Was läuft wann?"
Entwicklung	Code-Module	"Wie ist der Code organisiert?"
Physisch	Hardware-Deployment	"Was läuft wo?"
Szenarien (+1)	Use Cases	"Wie fließt eine Anfrage durch?"

Logische Sicht

Zeigt die Kernabstraktionen des Systems — die Domänenkonzepte.

Road Profile Viewer - Logische Sicht:

RoadProfile ← ProfileRepository

↑

└── GeometryCalculator

↑

└── ChartBuilder

Kernkonzepte: RoadProfile (Daten), Repository (Speicherung), Calculator (Mathematik), Builder (Visualisierung)

Prozesssicht

Zeigt was zur Laufzeit passiert — Prozesse, Threads, Interaktionen.

Benutzer → DashApp → Callback → Datenbank



Chart generieren



Aktualisierte Figur zurück



Anzeige für Benutzer

Hilft bei der Analyse: "Jede Benutzerinteraktion löst eine DB-Abfrage aus — ist das ein Engpass?"

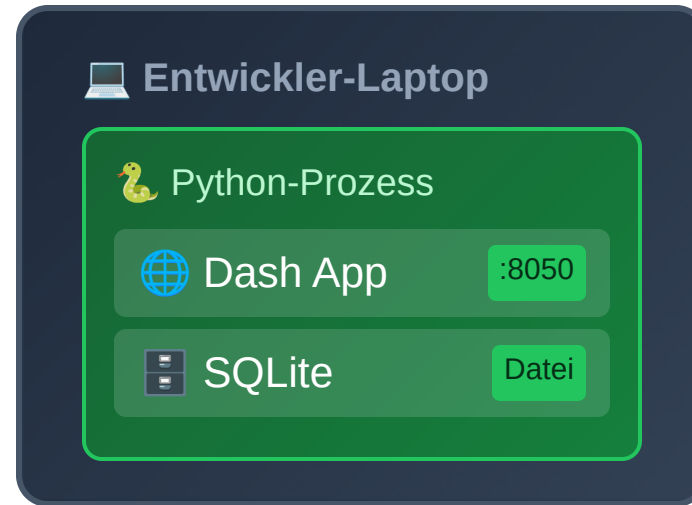
Entwicklungssicht

Zeigt wie Code in Pakete und Module organisiert ist.

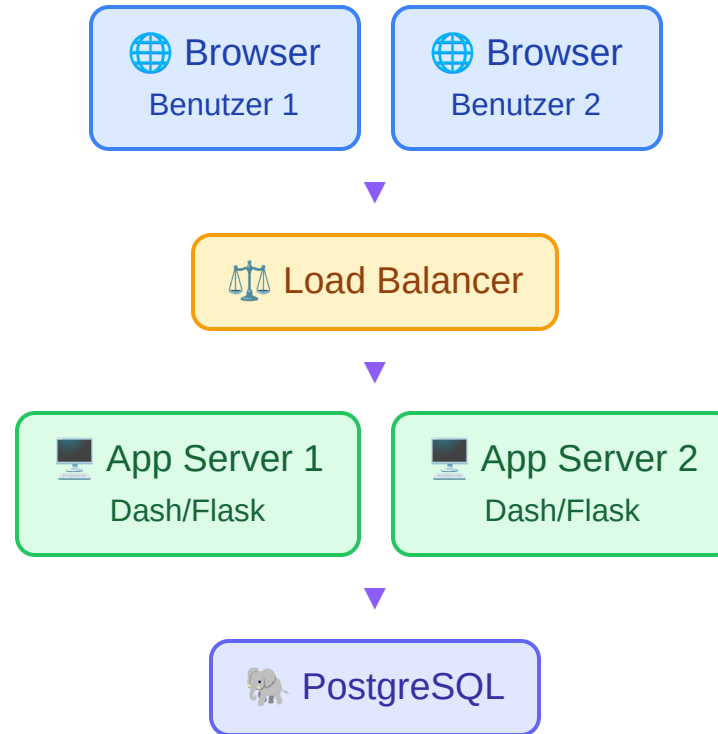
```
road-profile-viewer/  
├─ src/  
│   ├── models/          # Domänenmodelle  
│   ├── repositories/    # Datenzugriff  
│   ├── services/        # Geschäftslogik  
│   └─ presentation/     # Dash UI  
├─ tests/  
└─ pyproject.toml
```

Hilft bei der Koordination: "Anna arbeitet an `services/`, Ben arbeitet an `presentation/`."

Physische Sicht — Aktuell (Einfach)



Physische Sicht — Skaliert (Zukunft)



Szenarien: Das +1

Szenarien verbinden alle vier Sichten, indem sie eine Benutzeraktion nachverfolgen.

Szenario: Benutzer lädt ein neues Straßenprofil hoch

1. **Physisch:** Anfrage kommt am Load Balancer an → App Server 1
2. **Prozess:** Upload-Handler validiert, parst, ruft ProfileService auf
3. **Logisch:** ProfileService erstellt RoadProfile, validiert mit Pydantic
4. **Entwicklung:** Code in `services/` ruft `repositories/` auf
5. **Physisch:** Datenbankschreiben geht an PostgreSQL
6. **Prozess:** Antwort fließt zurück zum Browser des Benutzers

Schnelle Architekturdokumentation (Markdown)

Ihr braucht kein formales UML für ein Studentenprojekt!

ROAD PROFILE VIEWER ARCHITEKTUR

LOGISCHE SICHT: RoadProfile, ProfileRepository, GeometryCalculator

ENTWICKLUNGSSICHT: src/models/, src/services/, src/presentation/

PROZESSSICHT: Single-threaded Dash, Callbacks, synchrone DB-Abfragen

PHYSISCHE SICHT: Einzelrechner, SQLite, Port 8050

SCHLÜSSELSZENARIO: Profil anzeigen

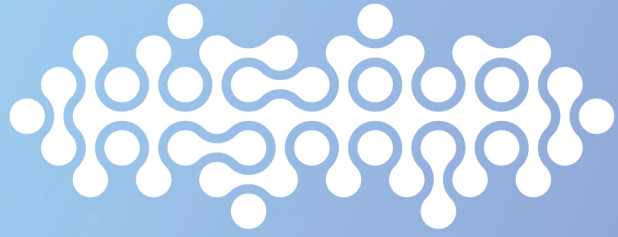
Benutzer wählt → Callback → DB-Abfrage → Chart → UI-Update

UML: Formale Notation

UML (Unified Modeling Language) — standardisierte visuelle Notation.

UML-Diagrammtyp	4+1 Sicht	Zeigt
Klassendiagramm	Logisch	Klassen, Attribute, Beziehungen
Sequenzdiagramm	Prozess	Objektinteraktionen über Zeit
Paketdiagramm	Entwicklung	Code-Organisation
Deployment-Diagramm	Physisch	Hardware-Knoten
Use-Case-Diagramm	Szenarien	Akteur-System-Interaktionen

Für jetzt: Einfaches Markdown + Mermaid-Diagramme sind ausreichend.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Zusammenfassung & Reflexion

5

Zusammenfassung

Konzept	Kernpunkt
Software-Architektur	Grundlegende Organisation als kommunizierende Komponenten
Architekturentscheidungen	Kreativer Prozess, getrieben von NFRs
Nicht-funktionale Anforderungen	Performance, Sicherheit, Verfügbarkeit, Wartbarkeit
4+1 Sichten	Logisch, Prozess, Entwicklung, Physisch + Szenarien
Trade-offs	Kann nicht alle Qualitäten optimieren; muss priorisieren

Wichtigste Erkenntnisse

1. **Architektur ist von Tag eins wichtig** — auch kleine Projekte profitieren
2. **NFRs formen die Architektur** — Performance, Sicherheit, Wartbarkeit treiben Entscheidungen
3. **Mehrere Sichten sind notwendig** — kein einzelnes Diagramm erfasst alles
4. **Trade-offs sind unvermeidlich** — priorisiert basierend auf den Bedürfnissen eures Systems
5. **Architektur ermöglicht Kommunikation** — Diagramme helfen Stakeholdern zu verstehen

Reflexionsfragen

1. **Aktuelle Architektur:** Wie würdet ihr die Architektur eures Road Profile Viewers beschreiben? Welche Sichten könnt ihr identifizieren?
2. **NFR-Prioritäten:** Was sind die drei wichtigsten NFRs für euer Projekt? Wie beeinflussen diese eure Entscheidungen?
3. **Trade-off-Szenario:** Einfacher zu warten ODER schneller auszuführen — was würdet ihr wählen? Warum?
4. **4+1 Übung:** Zeichnet ein einfaches Diagramm für jede der vier Sichten eures Projekts.

Was kommt als Nächstes: Teil 2

Demnächst: Architekturmuster

- **MVC (Model-View-Controller)** — Trennung von Concerns in interaktiven Apps
- **Schichtenarchitektur** — Präsentations-, Geschäfts-, Datenschichten
- **Verteilte Muster** — Client-Server, Microservices
- **Muster auf Road Profile Viewer anwenden** — euer Projekt weiterentwickeln

Ihr habt gelernt **WARUM** Architektur wichtig ist.
Jetzt lernt **WIE** man bewährte Muster anwendet.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

TEIL 1 ABGESCHLOSSEN