



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 06

SOFTWARE-ARCHITEKTUR — TEIL 2

Rückblick: Was wir in Teil 1 gelernt haben

In Teil 1 haben wir erkundet:

- **Warum Architektur wichtig ist** — die Brücke zwischen agiler Entwicklung und nachhaltigen Systemen
- **Das 4+1 Sichtenmodell** — Logisch, Prozess, Entwicklung, Physisch und Szenarien
- **Nichtfunktionale Anforderungen** formen architektonische Entscheidungen

Heute: Vom *Verstehen* zum *Anwenden* von Architektur mit bewährten **Mustern**.

Das Versprechen der Muster

Wenn ein Entwickler sagt "**wir verwenden MVC**", verstehen andere Entwickler sofort:

- Wie Verantwortlichkeiten aufgeteilt sind
- Wo Geschäftslogik vs. UI-Code zu finden ist
- Wie neue Features hinzugefügt werden können, ohne bestehende zu zerstören

*Muster sind nicht nur Code-Strukturen — sie sind **gemeinsames Vokabular**, das effiziente Kommunikation ermöglicht.*

Lernziele

Am Ende dieser Vorlesung werden Sie in der Lage sein:

1. **Architekturmuster zu erklären** und warum sie als bewährte Lösungen Mehrwert bieten
2. **MVC und Schichtenarchitektur anzuwenden**, um interaktive Anwendungen zu strukturieren
3. **Zu bewerten, wann verteilte Muster eingesetzt werden** wie Client-Server und Microservices
4. **Eine weiterentwickelte Architektur zu entwerfen** für den Road Profile Viewer

Was ist ein Architekturmuster?

***Ein Architekturmuster** ist die Beschreibung einer Systemorganisation, die erfolgreich in verschiedenen Softwaresystemen eingesetzt wurde.*

Muster sind Entdeckungen, keine Erfindungen.

Warum Muster verwenden?

1. **Bewährt:** Tausende Systeme haben diese Ansätze validiert
2. **Kommunizierbar:** "Wir verwenden MVC" vermittelt sofort Struktur
3. **Analysierbar:** Bekannte Muster haben bekannte Kompromisse
4. **Wiederverwendbar:** Framework-Unterstützung, Dokumentation, Community-Wissen

Drei Kategorien von Architekturmustern

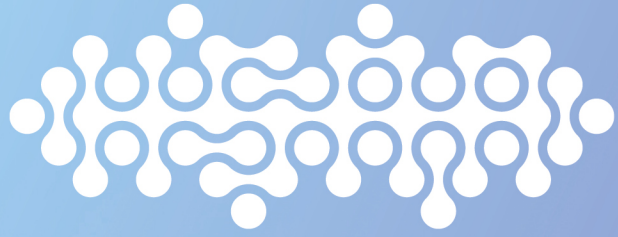
Kategorie	Zweck	Beispiele
Interaktive Systeme	Mensch-Computer-Interaktionen organisieren	MVC, MVP, MVVM
Systemstruktur	Komponenten und Objekte organisieren	Schichten, Pipes/Filter, Repository
Verteilte Systeme	Verteilte Ressourcen ermöglichen	Client-Server, Microservices, Event-Driven

Heute erkunden wir ein Muster aus jeder Kategorie.

Schnellumfrage: Welche Kategorie interessiert Sie am meisten?

- A)** Interaktive Systeme (MVC, MVP, MVVM)
- B)** Systemstruktur (Schichtenarchitektur)
- C)** Verteilte Systeme (Microservices)
- D)** Alle gleichermaßen!

Hand heben oder den Umfrage-Link nutzen!

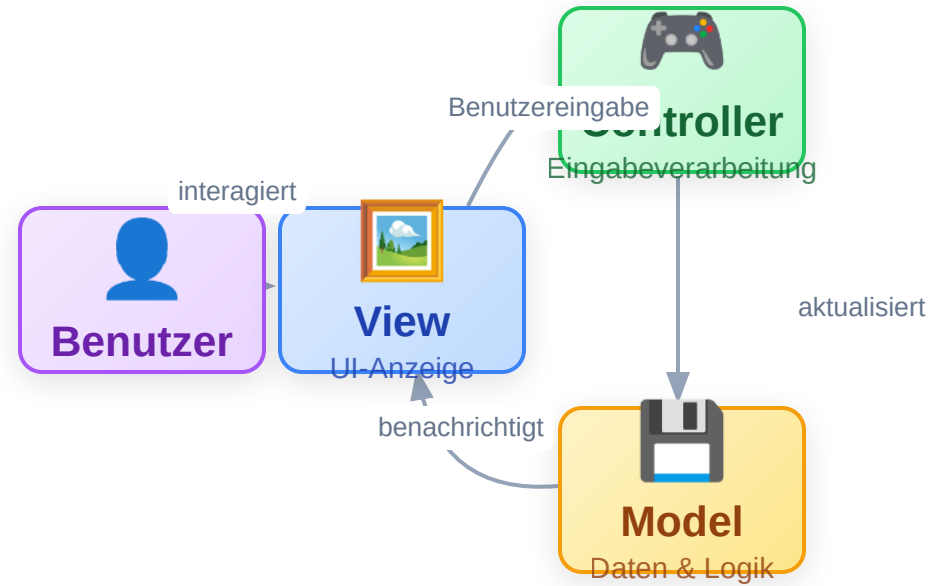


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Interaktive Systeme: MVC

1

Model-View-Controller (MVC)



MVC-Komponenten-Verantwortlichkeiten

Komponente	Verantwortlichkeit	Road Profile Viewer
Model	Daten und Geschäftslogik; weiß nichts über UI	<code>RoadProfile</code> -Klasse, <code>GeometryCalculator</code>
View	Zeigt Daten an; empfängt Benutzereingaben	Dash-Layout, Plotly-Charts
Controller	Verarbeitet Benutzereingaben; aktualisiert Model und View	Dash-Callbacks

Warum MVC?

Vorteile:

- **Trennung der Belange:** UI-Designer arbeiten an Views, Entwickler an Models
- **Testbarkeit:** Models können ohne UI getestet werden
- **Flexibilität:** Gleiches Model kann mehrere Views haben (Web, Mobile, CLI)

Kompromisse:

- Mehr Dateien zu verwalten
- Lernkurve für neue Entwickler
- Überdimensioniert für sehr einfache Apps

MVC in Django (Python)

Hinweis: Django verwendet **MTV** (Model-Template-View) Terminologie, nicht MVC. Djangos "View" fungiert als Controller!

```
# models.py – Model (wie bei MVC)
class RoadProfile(models.Model):
    name = models.CharField(max_length=100)
    data = models.JSONField()

# views.py – Django "View" = MVC Controller (verarbeitet Anfragen)
def profile_detail(request, profile_id):
    profile = RoadProfile.objects.get(id=profile_id)
    return render(request, 'profile.html', {'profile': profile})

# templates/profile.html – Django "Template" = MVC View (Darstellung)
<h1>{{ profile.name }}</h1>
```

MVC in Flask (Python)

```
# models.py (Model)
@dataclass
class RoadProfile:
    id: str
    name: str
    coordinates: list[tuple[float, float]]

# routes.py (Controller)
@app.route('/profiles/<profile_id>')
def show_profile(profile_id):
    profile = ProfileRepository.get(profile_id)
    return render_template('profile.html', profile=profile)

# templates/profile.html (View) - Jinja2-Template
```

Think-Pair-Share: MVC in Ihrer Erfahrung

2 Minuten: Denken Sie an Webanwendungen, die Sie genutzt haben.

2 Minuten: Diskutieren Sie mit einem Nachbarn.

Teilen: Welche Muster erkennen Sie?

Diskussionsfragen:

1. Können Sie Model, View, Controller in Apps identifizieren, die Sie gebaut haben?
2. Haben Sie Apps gesehen, in denen Belange vermischt waren? Welche Probleme entstanden?

Road Profile Viewer: MVC anwenden

Model (`models/profile.py`):

```
class RoadProfile(BaseModel):  
    """Domain-Model - weiß nichts über UI oder Speicherung."""  
    id: str  
    name: str  
    x_coordinates: list[float]  
    y_coordinates: list[float]  
  
    def max_slope(self) -> float:  
        """Geschäftslogik lebt im Model."""  
        ...
```

Road Profile Viewer: View-Komponente

View (`presentation/charts.py`):

```
def create_profile_figure(profile: RoadProfile) -> go.Figure:
    """Reine Visualisierung - keine Geschäftslogik."""
    return go.Figure(
        data=go.Scatter(
            x=profile.x_coordinates,
            y=profile.y_coordinates,
            mode='lines', name=profile.name
        ),
        layout=go.Layout(title=f"Straßenprofil: {profile.name}")
    )
```

Road Profile Viewer: Controller-Komponente

Controller (`presentation/callbacks.py`):

```
@callback(
    Output('profile-chart', 'figure'),
    Input('profile-dropdown', 'value')
)
def update_chart(profile_id: str):
    """Controller: koordiniert Model und View."""
    if not profile_id:
        return go.Figure() # Leere Figur

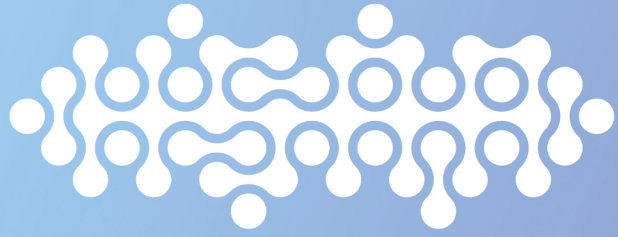
    profile = repository.get(profile_id) # Model holen
    return create_profile_figure(profile) # View erstellen
```

Erreichte MVC-Vorteile

Mit sauberer MVC-Trennung:

- `RoadProfile` kann **ohne Dash** getestet werden
- Charts können in anderen Kontexten wiederverwendet werden (PDF-Export, etc.)
- Callbacks sind **schlank** — nur Koordination, keine Geschäftslogik

*Das Model weiß nichts über die View.
Die View weiß nichts über den Controller.
Alle sprechen mit dem Model.*



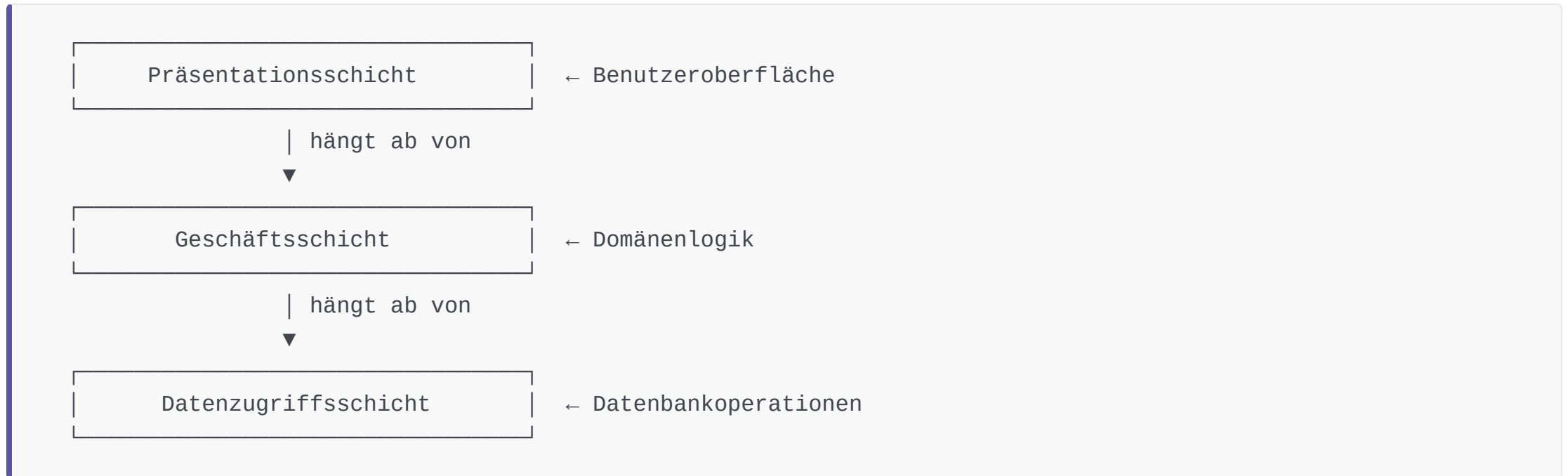
Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Systemstruktur: Schichtenarchitektur

2

Schichtenarchitektur

Schichtenarchitektur organisiert Code in horizontale Schichten, wobei jede Schicht nur von der darunterliegenden Schicht abhängt.



Schichtenarchitektur: Schlüsselregeln

Die drei goldenen Regeln:

1. **Nur Abwärts-Abhängigkeiten:** Präsentation → Geschäft → Daten
2. **Keine Schichten überspringen:** Präsentation sollte NICHT direkt Daten aufrufen
3. **Schichten sind kohäsiv:** Aller UI-Code in Präsentation, alle Domänenlogik in Geschäft

Warum? Änderungen in einer Schicht kaskadieren nicht durch das gesamte System.

Schicht-Verantwortlichkeiten

Schicht	Verantwortlichkeit	Typische Komponenten
Präsentation	Benutzerinteraktion, Daten anzeigen	Webseiten, API-Endpunkte, CLI
Geschäft	Domänenlogik, Validierung, Berechnungen	Services, Domänenmodelle, Validatoren
Datenzugriff	Daten persistieren und abrufen	Repositories, ORM-Models, Cache

Netflix: Ein Praxisbeispiel

Netflix demonstriert Schichtenprinzipien im großen Maßstab:

PRÄSENTATION: Web-UI, Mobile Apps, TV Apps, API Gateway

|

GESCHÄFT: User Service, Content Service, Recommendation

|

DATEN: Cassandra, MySQL, ElasticSearch, S3

Netflix: Präsentationsschicht

Komponente	Beschreibung
Web UI	React Single Page Application
Mobile Apps	iOS & Android native Apps
TV Apps	Smart TVs, Roku, PlayStation
API Gateway (Zuul)	Leitet Anfragen an Services

Jeder Präsentations-Client kann unabhängig aktualisiert werden!

Netflix: Geschäftsschicht (Microservices)

Service	Verantwortlichkeit
User Service	Authentifizierung & Profile
Content Service	Filme & Serien Metadaten
Recommendation	ML-gestützte Vorschläge
Playback Service	Streaming-Logik

Jeder Service kann **unabhängig skaliert** werden basierend auf Bedarf!

Netflix: Datenschicht

Datenbank	Anwendungsfall
Cassandra	Benutzerdaten & Wiedergabeverlauf
MySQL	Abrechnung & Konten
ElasticSearch	Suchindizierung
S3	Videodateien (Petabytes!)

Wichtige Erkenntnis: Verschiedene Datenspeicher für verschiedene Bedürfnisse!

Schichtenarchitektur: Wichtige Vorteile

Unabhängige Skalierung:

- Mehr Empfehlungen nötig?
- Nur diesen Service skalieren
- Andere Schichten unverändert

Technologie-Flexibilität:

- SQLite gegen PostgreSQL tauschen
- Nur Datenzugriffsschicht ändern
- Geschäftslogik unberührt

Jede Schicht kann sich unabhängig weiterentwickeln.

Road Profile Viewer: Aktueller Zustand (Vermischte Belange)

```
@app.callback(...)
def update_chart(profile_id):
    if not profile_id:
        return {} # UI-Logik

    conn = sqlite3.connect('profiles.db') # Datenzugriff
    cursor = conn.execute('SELECT * FROM profiles WHERE id = ?', (profile_id,))
    row = cursor.fetchone()

    x_coords = json.loads(row[1]) # Geschäftslogik
    y_coords = json.loads(row[2])
    max_slope = calculate_max_slope(x_coords, y_coords)

    return create_figure(x_coords, y_coords, max_slope) # Wieder UI
```

Problem: Alle Schichten in einer Funktion vermischt!

Road Profile Viewer: Datenschicht

`data_access/repositories.py` :

```
class ProfileRepository:
    def __init__(self, db_path: str = 'profiles.db'):
        self.db_path = db_path

    def get(self, profile_id: str) -> Profile | None:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.execute(
            'SELECT * FROM profiles WHERE id = ?', (profile_id,)
        )
        row = cursor.fetchone()
        return Profile.from_row(row) if row else None
```

Road Profile Viewer: Geschäftsschicht

business/services.py :

```
class ProfileService:
    def __init__(self, repository: ProfileRepository):
        self.repository = repository

    def get_profile_with_analysis(self, profile_id: str) -> dict:
        profile = self.repository.get(profile_id)
        if not profile:
            raise ProfileNotFoundError(profile_id)

        return {
            'profile': profile,
            'max_slope': self._calculate_max_slope(profile),
        }
```

Road Profile Viewer: Präsentationsschicht

presentation/callbacks.py :

```
service = ProfileService(ProfileRepository())

@app.callback(...)
def update_chart(profile_id):
    if not profile_id:
        return empty_figure()

    try:
        data = service.get_profile_with_analysis(profile_id)
        return create_figure(data)
    except ProfileNotFoundError:
        return error_figure("Profil nicht gefunden")
```

Sauber! Jede Schicht hat eine Verantwortlichkeit.

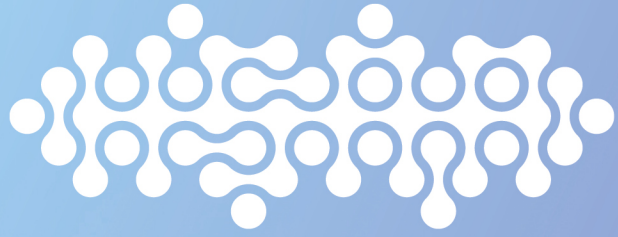
Think-Pair-Share: Wo würden Sie Caching hinzufügen?

Szenario: Profil-Berechnungen sind langsam. Sie möchten Ergebnisse cachen.

Diskutieren Sie:

1. In welcher Schicht sollte der Cache leben?
2. Sollte die Geschäftsschicht vom Cache wissen?
3. Wie würden Sie es implementieren, ohne andere Schichten zu ändern?

2 Minuten nachdenken, 2 Minuten zu zweit, dann teilen!

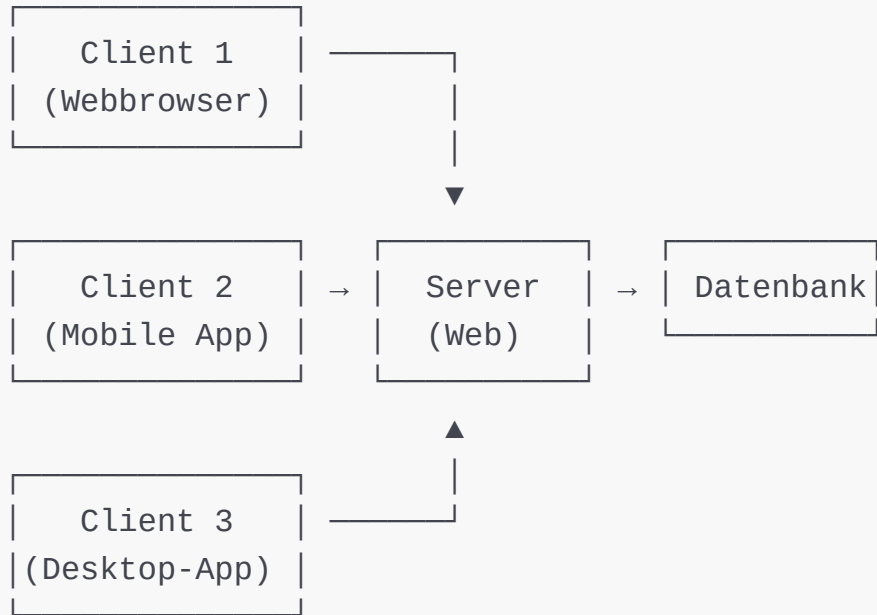


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Verteilte Systeme

3

Client-Server-Architektur



Client-Server-Eigenschaften

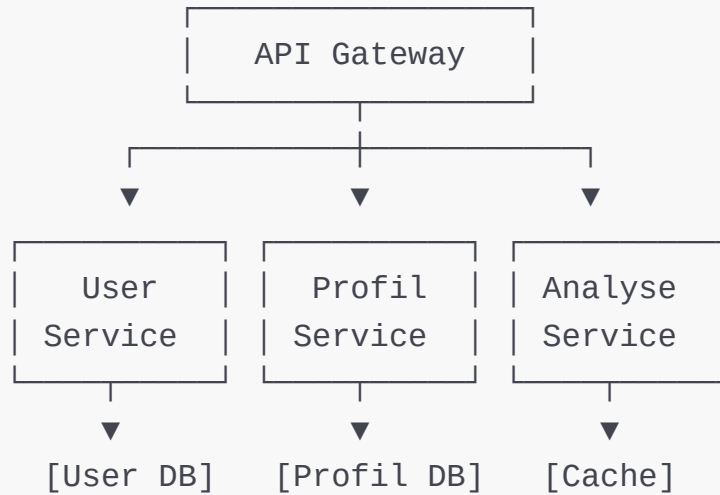
Komponente	Verantwortlichkeit
Clients	Anfragen initiieren, Ergebnisse anzeigen, Benutzereingaben verarbeiten
Server	Anfragen verarbeiten, Daten verwalten, Geschäftsregeln durchsetzen
Protokoll	Meist HTTP/HTTPS für Webanwendungen

Road Profile Viewer ist bereits Client-Server:

- Client: Webbrowser mit JavaScript/Dash-Frontend
- Server: Python Dash/Flask-Prozess verarbeitet Callbacks

Microservices-Architektur

Microservices zerlegen ein System in kleine, unabhängig deploybare Services:



Monolith vs. Microservices

Monolith:

- Alles-oder-nichts-Deployment
- Alles zusammen skalieren
- Eine Sprache/Framework
- Ein Bug kann System crashen
- Einfaches Deployment

Microservices:

- Unabhängiges Deployment
- Services einzeln skalieren
- Bestes Tool für jeden Job
- Fehler sind isoliert
- Komplexes Deployment

Amazons Zwei-Pizza-Teams

2002 verfügte Jeff Bezos:

"Alle Teams müssen ihre Funktionalität über Service-Schnittstellen bereitstellen."

Die "Zwei-Pizza-Team"-Regel: Teams klein genug, um mit zwei Pizzen satt zu werden.

Ergebnis: Amazon Web Services — ursprünglich interne Infrastruktur, jetzt ein **80+ Milliarden Dollar Geschäft**.

Wann Microservices einsetzen

JA, verwenden wenn:

- Große Teams (können nicht koordinieren)
- Unterschiedliche Skalierungsbedarfe
- Technologie-Vielfalt erforderlich
- Teams brauchen Autonomie

NICHT verwenden wenn:

- Kleine Teams (3-5 Personen)
- Produkte in früher Phase
- Einfache Domänen
- Schnelle Iteration nötig

Road Profile Viewer: *Bleibt beim Monolith! Ihr habt 3 Entwickler und eine einfache Domäne.*

Schnellumfrage: Monolith oder Microservices?

Szenario: Euer Road Profile Viewer Team wächst von 3 auf 30 Entwickler. Ihr habt jetzt 5 Feature-Teams, die an verschiedenen Teilen arbeiten.

Welche Architektur würden Sie empfehlen?

- A)** Monolith behalten, nur Code besser organisieren
- B)** In Microservices aufteilen
- C)** Mit "modularem Monolith" starten und später aufteilen

AWS Cloud-Architektur

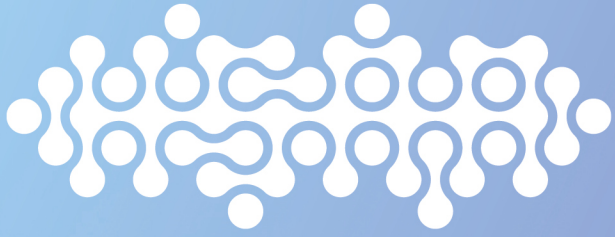
Cloud-Plattformen bieten Bausteine für verteilte Architekturen:

Users → CloudFront (CDN) → Load Balancer → ECS/Fargate → RDS
(Edge Cache) (Routing) (Container) (Datenbank)

Für Road Profile Viewer (Zukunft):

- **Render.com** oder **Railway.app** — einfaches PaaS, kostenlose Stufe
- **Supabase** — verwaltetes PostgreSQL mit kostenloser Stufe

Ihr braucht keine AWS-Komplexität für ein Studentenprojekt!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Muster auf Road Profile Viewer anwenden

4

Aktueller Zustand: Vom Monolith zu Modulen

In Vorlesung 4 (Refactoring) haben wir eine monolithische `main.py` in Module aufgeteilt:

```
road-profile-viewer/  
├── src/  
│   ├── geometry.py      # Strahl-Schnitt-Berechnungen  
│   ├── road.py          # Profil-Generierung  
│   ├── visualization.py  # Chart-Erstellung  
│   └── main.py           # Anwendungs-Einstiegspunkt  
└── tests/
```

Guter Start! Aber wir können mit einer echten Schichtenarchitektur weitergehen.

Vorgeschlagene weiterentwickelte Architektur

```
road-profile-viewer/
├── src/
│   ├── domain/                # Kern-Geschäftskonzepte
│   │   ├── models.py          # RoadProfile, Measurement
│   │   └── services.py        # GeometryCalculator
│   ├── infrastructure/        # Externe Systeme
│   │   └── repositories.py    # ProfileRepository
│   ├── presentation/          # Benutzeroberfläche
│   │   ├── callbacks.py       # Controller
│   │   └── charts.py          # Views
│   └── api/                   # Optional: REST API
│       └── routes.py           # FastAPI-Endpunkte
```

Eine REST-API-Schicht hinzufügen

Warum eine API, wenn Dash bereits funktioniert?

- **Mehrere Frontends:** Gleiches Backend für Web, Mobile, CLI
- **Testen:** APIs sind einfacher zu testen als UI-Callbacks
- **Integration:** Andere Systeme können auf eure Daten zugreifen

```
@app.get("/api/profiles/{profile_id}")
def get_profile(profile_id: str) -> ProfileResponse:
    profile = repository.get(profile_id)
    if not profile:
        raise HTTPException(status_code=404)
    return ProfileResponse(**profile.dict())
```

Zukünftige Skalierbarkeit

Falls ihr Road Profile Viewer jemals skalieren müsst:

Änderung	Was zu ändern ist
Datenbank	SQLite durch PostgreSQL ersetzen — nur <code>infrastructure/</code> ändern
Caching	Redis-Cache hinzufügen — neue Datei in <code>infrastructure/</code>
Deployment	Mit Docker containerisieren — keine Code-Änderungen
Load Balancing	Mehrere Instanzen hinzufügen — nur Infrastruktur

Die Schichtenarchitektur macht Änderungen lokal!

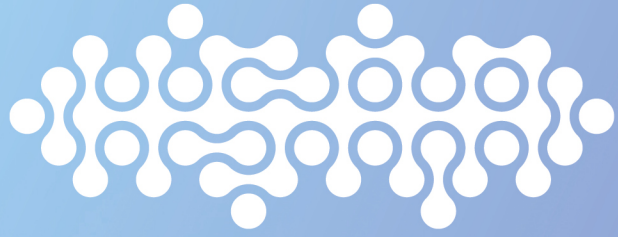
Interaktive Übung: Das nächste Feature entwerfen

Szenario: Der Product Owner möchte "Profil als PDF exportieren" hinzufügen.

Beantwortet zu zweit:

1. Zu welcher Schicht gehört die PDF-Generierung?
2. Braucht es einen neuen Service in der Geschäftsschicht?
3. Wie würdet ihr es bereitstellen (Button im UI? API-Endpunkt? Beides?)
4. Welche bestehenden Komponenten würdet ihr wiederverwenden?

5 Minuten, dann diskutieren wir!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Architektur & Agile

5

Der scheinbare Konflikt

Manche Entwickler glauben, Agile und Architektur stehen im Konflikt:

Extrem 1:

"Agile heißt kein Vorab-Design — einfach anfangen zu coden und refactoren!"

Extrem 2:

"Architektur erfordert monatelange Planung, bevor Code geschrieben wird!"

Beide Extreme sind falsch.

Emergente Architektur in Scrum

Der agile Ansatz: Emergentes Design mit absichtlicher Struktur.

1. **Genug Entscheidungen treffen, um zu starten:** Sprache, Framework, Grundstruktur wählen
2. **Funktionierende Software implementieren:** Echte Features bauen, keine hypothetische Infrastruktur
3. **Refactorn, wenn Muster entstehen:** Bei Duplikation umstrukturieren
4. **Entscheidungen regelmäßig überprüfen:** Sprint-Retros können technische Schulden adressieren

Technische Spikes

Ein **technischer Spike** ist eine zeitbegrenzte Untersuchung zur Reduktion von Unsicherheit.

Beispiel-Spike-Story:

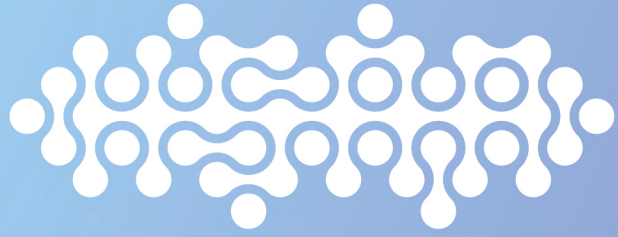
Als Entwickler

***möchte ich** PostgreSQL vs SQLite für unsere Datenbank untersuchen*

***damit** wir eine informierte Entscheidung treffen können*

***Zeitlimit:** Maximal 2 Tage*

Nach dem Spike: Eine *informierte* architektonische Entscheidung treffen, kein Raten.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Zusammenfassung

6

Zusammenfassungstabelle

Konzept	Kernpunkt	Anwendung
Muster	Bewährte Lösungen für wiederkehrende Probleme	MVC für vorhersagbare Struktur
MVC	Model, View, Controller trennen	Pydantic-Models, Charts, Callbacks
Schichten	Präsentation → Geschäft → Daten	presentation/ , domain/ , infrastructure/
Verteilt	Client-Server, Microservices	Vorerst monolithisch bleiben
Agile	Emergentes Design, technische Spikes	Entscheiden, implementieren, refactorieren

Kernaussagen

1. **Muster sind bewährte Lösungen** — MVC, Schichtenarchitektur und Client-Server lösen wiederkehrende Probleme
2. **MVC trennt Belange** — Model kennt Daten, View kennt Anzeige, Controller koordiniert
3. **Schichten schaffen Grenzen** — Abhängigkeiten fließen nur abwärts
4. **Microservices sind nicht immer die Antwort** — Kleine Teams profitieren von gut strukturierten Monolithen
5. **Architektur entwickelt sich mit Agile** — Einfach starten, refactoren wenn Muster entstehen

Reflexionsfragen

1. **Muster-Identifikation:** Welchem Muster folgt euer Road Profile Viewer derzeit?
2. **Schicht-Zuordnung:** Zu welcher Schicht gehört jede eurer Python-Dateien?
3. **Abhängigkeitsrichtung:** Zeigen alle eure Imports "abwärts"?
4. **Skalierungs-Trigger:** Welche Metrik würde euch sagen, dass es Zeit ist, von SQLite zu PostgreSQL zu wechseln?

Was kommt als Nächstes: Auf euer Projekt anwenden

Eure Aktionspunkte:

1. **Aktuelles Muster identifizieren:** Welchem Muster folgt euer Code (wenn überhaupt)?
2. **Schichten zuordnen:** Welcher Code ist Präsentation? Geschäft? Datenzugriff?
3. **Eine Schicht refactoren:** Den unordentlichsten Bereich wählen und Trennung anwenden
4. **Architektur dokumentieren:** Eine einfache `ARCHITECTURE.md` erstellen
5. **API in Betracht ziehen:** Würde eine REST-API eurem Projekt nützen?

Weiterführende Literatur

Bücher:

- Ian Sommerville: *Software Engineering* (Kapitel 6)
- Robert C. Martin: *Clean Architecture*
- Martin Fowler: *Patterns of Enterprise Application Architecture*

Online:

- [Martin Fowler über MVC](#)
- [Netflix Tech Blog](#)
- [The Twelve-Factor App](#)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK