Coding Challenge - Lösung.

https://github.com/dacchar/QvestDigitalTrainingOffering

# API-Design.

Die REST-API wurde mithilfe von ChatGPT konzipiert und validiert. Die entsprechende OpenAPI-Spezifikation befindet sich in der Datei OpenAPISpezifikation.yaml und kann im Swagger Editor unter <https://editor.swagger.io/> visualisiert und angepasst werden.

# Coding

## 2.1 Backend

Das Backend wurde mit Java Spring Boot entwickelt und befindet sich in <https://github.com/dacchar/QvestDigitalTrainingOffering>. Es implementiert die CRUD-Operationen für die Ressource trainings, verwendet jedoch intern den Endpoint /courses anstelle von /trainings. Die Daten werden in einer In-Memory-Datenbank (H2) gespeichert und beim Beenden der Anwendung nicht persistiert.

### 2.1.1 Testing

Zu Demonstrationszwecken habe ich JUnit-Tests für alle drei Schichten des Backends entwickelt: Repository, Service und Controller. Auch wenn dies für eine so kleine Anwendung nicht zwingend erforderlich ist, ist es in einer produktiven Anwendung absolut sinnvoll, um die Qualität, Wartbarkeit und Testbarkeit des Codes sicherzustellen. Mein Ziel war es, die zugrunde liegende Testtechnik zu zeigen und deren Einsatz zu veranschaulichen.

## Frontend

Das Frontend wurde mit Hilfe von Angular entwickelt. Das GUI-Design wurde bewusst einfach und funktional gehalten.

## 2.2.1 Testing

Die Tests wurden mit Hilfe von Jasmin Testing Framework entwickelt. Die Tests kann man mit Hilfe von „Karma Test Runner“ laufen lassen und die Ergebnisse anschauen. Im Rahmen der Demo wurde ein einzelner Anwendungsfall getestet: Wenn das Backend zwei Kurse zurückliefert, soll das Frontend diese korrekt darstellen.  
Der Test validiert, ob genau zwei Einträge in der Benutzeroberfläche erscheinen.

# Theory

## Security

Die Kommunikation zwischen Frontend und Backend soll ausschließlich über HTTPS erfolgen. Für die Authentifizierung stehen zwei Optionen zur Verfügung: JWT (JSON Web Token) und Session-basierte Authentifizierung. Da Session Authentication in verteilten Systemen schlechter skalierbar ist als JWT, würde ich JWT-basierte Authentifizierung bevorzugen. Für die Autorisierung beabsichtige ich, das OAuth 2.0-Protokoll in Kombination mit dem Identity Provider Auth0 zu verwenden.

## CI/CD

Eine mögliche CI/CD-Pipeline, z. B. Jenkins für das Projekt umfasst folgende Schritte:

1. **Quellcode-Checkout**

Der Code wird aus einem Source Repository wie GitHub geladen.

2. **Build und Kompilierung**

Der Code wird kompiliert mit Dependency-Management Tools (z. B. mit Maven oder Gradle).

3. **Code-Analyse**

Tools wie SonarQube prüfen den Code auf Qualität, Testcoverage, Sicherheitsprobleme und Verstöße gegen Clean-Code-Prinzipien.

4. **Containerisierung**

Die Build-Artifakten werden in ein Docker-Image gepackt.

5. **Bereitstellung ins Artifakt-Repository**

Das Image wird in ein zentrales Repository wie Nexus oder JFrog Artifactory hochgeladen.

6. **Deployment**

Je nach Infrastruktur erfolgt das Deployment z. B. auf AWS EC2, AWS ECS oder AWS EKS. Um eine Vendor-Lock-in-Situation zu vermeiden und die Anwendung flexibel in verschiedenen Cloud- oder On-Premises-Umgebungen betreiben zu können, bietet sich die Verwendung von Kubernetes als plattformunabhängige Orchestrierungslösung an.

## Cloud

Nach dem **AWS-Konzept** wäre die bevorzugte Technologie für serverlose Anwendungen **AWS Lambda.** Da das Backend jedoch mit **Java Spring Boot** entwickelt wurde – was sich nicht optimal für Lambda eignet, insbesondere wegen der Kaltstartzeiten und langen Initialisierung –, ist es sinnvoller, auf **ECS (Elastic Container Service)** oder **EKS (Elastic Kubernetes Service)** zu setzen.  
Beide Services bieten automatische **Skalierbarkeit** und eignen sich gut für containerisierte Anwendungen.

Sollte **Cloud-Anbieter-Unabhängigkeit** ein Ziel sein, empfiehlt es sich, **Kubernetes** als Plattform zu wählen. Kubernetes kann sowohl in AWS (über EKS), als auch in anderen Cloud- oder On-Premises-Umgebungen betrieben werden – und schützt somit vor einem **Vendor Lock-in.**

Für die Datenbank würde ich **Amazon RDS** empfehlen, da es eine vollständig verwaltete relationale Datenbanklösung ist. In verteilten Systemen sind jedoch oft **NoSQL-Datenbanken** geeigneter, da diese **horizontal skalierbar** sind, während relationale Datenbanken eher **vertikal skaliert** werden. Eine passende NoSQL-Lösung in AWS wäre z. B. **DynamoDB**, die speziell für hohe Verfügbarkeit, Performance und Skalierbarkeit ausgelegt ist – insbesondere bei stark verteilten Workloads.

Sowohl das **Backend** als auch das **Frontend** sollten innerhalb einer **privaten VPC** (Virtual Private Cloud) deployt werden, um maximale Sicherheit zu gewährleisten. Das **API Gateway**, das eingehende **HTTP-Anfragen** entgegennimmt, muss hingegen in einer **öffentlichen VPC** verfügbar gemacht werden, um von außen erreichbar zu sein.

Die **Skalierbarkeit** der Anwendung wird durch die **verwalteten AWS-Services ECS und/oder EKS** sichergestellt – diese müssen jedoch entsprechend konfiguriert und mit Auto-Scaling-Regeln ausgestattet werden. Diese Regeln könnten durch spezifische Konfigurationen (e.g. CPU load), weitere Compute Instanzen (e.g. EC2) starten.

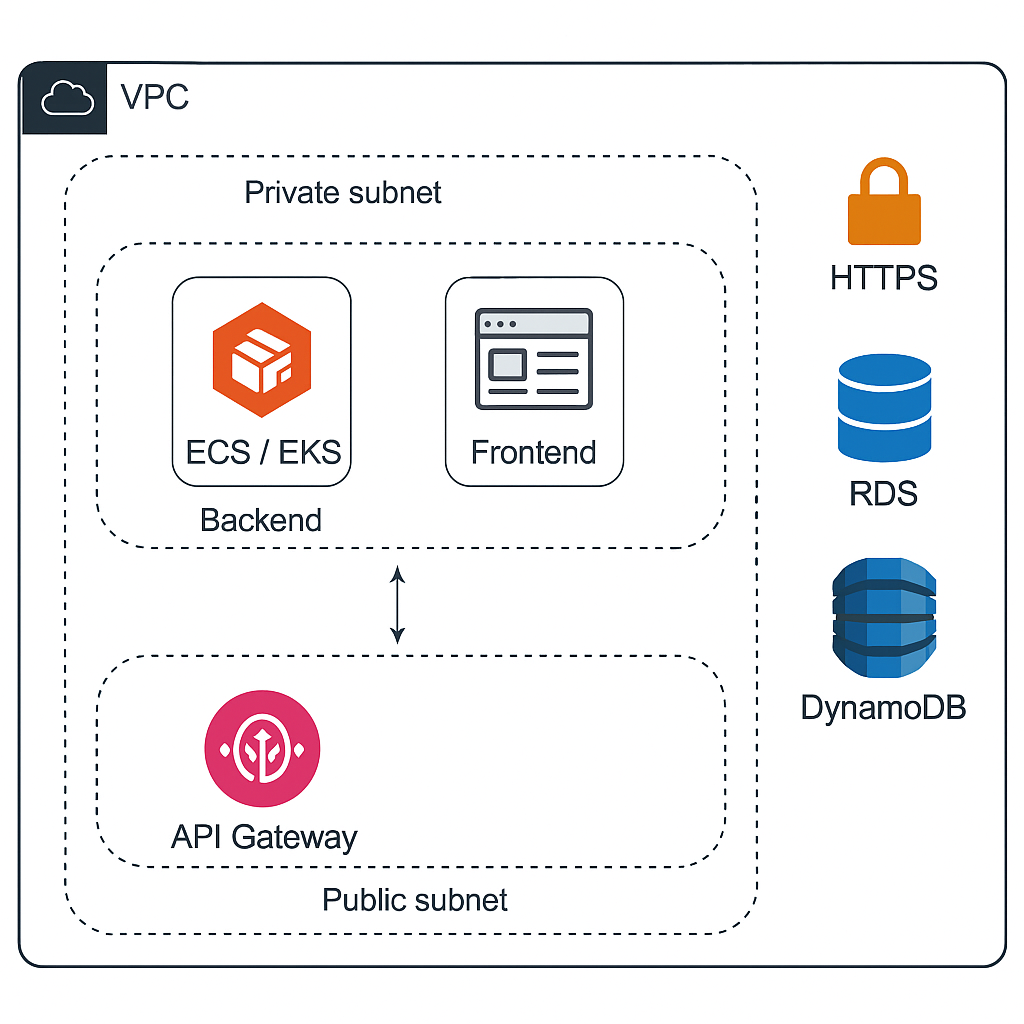


Bild 1. Architekturdiagramm.

Auf dem Bild 1 ist ein mögliches Architekturdiagramm dargestellt.

## Code Quality

In der Beispiel-Implementierung wurden Unit-Tests entwickelt. In produktiven Anwendungen sollten zusätzlich auch Integrationstests und End-to-End-Tests (E2E-Tests) implementiert werden. Die E2E-Tests können beispielsweise auf Selenium WebDriver basieren. Für geschäftskritische Anwendungen kann außerdem Canary Testing vorgesehen werden, bei dem die Anwendung zunächst nur einem kleinen Prozentsatz der Nutzer zur Verfügung gestellt wird (e.g. 10%). Falls keine Fehler auftreten, wird die Applikation für die restlichen Nutzern freigestellt. Dies reduziert den „blast radius“ von einem Fehler.

Für Code-Qualitätssicherung würde ich SonarCube benutzen. SonarQube ist ein Werkzeug zur Qualitätssicherung von Quellcode. Es analysiert den Code automatisiert, sammelt relevante Metriken (z.B. Test code Coverage, …) und stellt umfassende Berichte zur Codequalität des Projekts bereit.

## Monitoring & Logging

Nach AWS-Konzept muss man für Monitoring und Logging AWS CloudWatch benutzen. Bei AWS CloudWatch ist es möglich 4xx und 5xx Fehlercodes vom API-Gateway zu monitoren. Bei Server-Fehlern 5xx kann man strikte Alarme konfigurieren, um schnellstmögliche Maßnahmen vorzunehmen.

Mit AWS CloudWatch kann man auch CPU, Memory und Applikation Logs monitoren und bei Bedarf alarmieren.

Als alternative zu AWS CloudWatch für Monitoring kann man auch Prometheus und Grafana benutzen. Prometheus eignet sich hervorragend zum Sammeln und Speichern von Zeitreihendaten, während Grafana eine leistungsstarke Plattform zur Visualisierung dieser Daten in Dashboards bietet. Gemeinsam ermöglichen sie ein effizientes Monitoring und Alerting in komplexen Systemumgebungen.