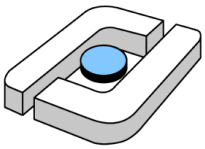

Hochschule Osnabrück

University of Applied Sciences

Fakultät
Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Quiz-API

im Rahmen des Moduls
Software-Architektur – Konzepte und Anwendungen,
des Studiengangs Informatik-Medieninformatik

Autor:	Johanna Bernhard
Matr.-Nr.:	840686
E-Mail:	johanna.bernhard@hs-osnnab-rueck.de

Autor:	Laura Peter
Matr.-Nr.:	869876
E-Mail:	laura.peter@hs-osnnabrueck.de

Themensteller: Prof. Dr. Rainer Roosmann

Abgabedatum: 07.08.2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Source-Code Verzeichnis

Abkürzungsverzeichnis

CDI	Context and Dependency Injection for the Java EE Platform
ECB	Entity-Controller-Boundary Pattern
Java EE	Java Enterprise Edition, in der Version 7
SWA	Software-Architektur
OIDC	OpenID Connect
RBAC	Role Based Access Control
ABAC	Attribute Bases Access Control
JWT	JSON Web Token
API	Application Programming Interface
DTO	Data Transfer Object
URI	Uniform Resource Identifier
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
JSON	JavaScript Object Notation
JSON-B	JSON Binding
JAX-RS	Jakarta RESTful Web Services
ORM	object-relational mapping
ACL	Anti-Corruption Layer
JPA	Java Persistence API

1 Einleitung (Bernhard + Peter)

Quizanwendungen werden oft im Rahmen von Lehrveranstaltungen aber auch im Alltag von vielen Menschen verwendet, um den eigenen Wissensstand zu testen oder sich mit Freunden und anderen Nutzern zu messen. Gute Beispiele für Quiz-Apps sind „Quizduell“ oder „Kahoot!“. Bei „Quizduell“ tritt man beispielsweise gegen andere Nutzer an und kann bei einem Quiz wählen aus welcher Kategorie Fragen gestellt werden. „Kahoot!“ lässt einen zudem selbst Quizze erstellen und mehrere Nutzer daran teilhaben. Beim Spielen eines Quizzes erhält der Nutzer Punkte pro richtig beantworteter Frage, woraus am Ende ein Leaderboard entsteht, dass die Nutzer nach ihrer erzielten Punktzahl in einer Rangliste zuordnet.

Gamification ist zurzeit ebenfalls ein wichtiger Begriff, der in Lehrmethoden dadurch ausgezeichnet wird, spielerisch Lehrinhalte zu vermitteln. Da aktuell viel „Home-Learning“, also „Lernen Zuhause“, stattfinden muss, ist es wünschenswert Lehrmethoden mit Gamification-Ansatz über Online-Anwendungen, wie genannte Quiz-Apps zu unterstützen.

Neben verschiedenen Quiz-Apps gibt es jedoch keine API, die es einem einfach machen könnten, Quizanwendungen umzusetzen.

Aus diesen Gründen entsteht im Rahmen dieses Projekts eine Quiz-API, die den Namen „QuizFest“ trägt.

Bei der Entwicklung wird besonders darauf geachtet, dass eine möglichst developerfreundliche API entsteht, welche es ermöglicht in weiteren Projekten eine Client-Side-Rendering Anwendung zu entwickeln. Zusätzlich ist das Ziel ein sauberes Softwarearchitekturdesign aufzuweisen.

1.1 Vorstellung des Themas

QuizFest soll es einem ermöglichen Quizze zu erstellen, veröffentlichen und eigene sowie die anderer Nutzer zu spielen. Die Quizze sollen Kategorien zugeordnet werden können, sodass bei der Suche nach einem Quiz nach ihnen gefiltert werden kann. Außerdem soll ein User-System bereitgestellt werden, wodurch nach Login nur selbsterstellte Quizze bearbeitet werden können und in der Anwendung zwischen Admin und User unterschieden werden kann.

1.2 Ziel der Ausarbeitung

Ziel der Ausarbeitung ist die Entwicklung einer API mit dem Java-Framework „Quarkus“. In diesem Sinne, werden Technologien ausgewählt werden, die den Anforderungen von QuizFest gerecht werden und in der Implementierung der Softwarearchitektur von Nutzen sind. Das erstellte Softwaredesign soll begründet und erläutert werden.

1.3 Aufbau der Hausarbeit

Zur Erläuterung des Entwicklungsprozesses werden im zweiten Kapitel zunächst Grundlagen erläutert, um für das Projekt wichtige Prinzipien vorzustellen und zu begründen warum bestimmte Technologien, Prinzipien und Softwarearchitekturansätze gewählt worden.

Im dritten Kapitel wird die Anwendung vorgestellt und zunächst die Anforderungen für die API benannt, um die danach aufbauend die entstandene Softwarearchitektur herzuleiten und zu begründen. Dabei werden die zuvor vorgestellten Grundlagen mit ihrem Einsatz in der Implementierung verknüpft. Außerdem werden die eingesetzten Testing- und Dokumentationsmethoden aufgezeigt, die zur Absicherung der Softwarequalität dienen.

Im letzten Kapitel wird zusammengefasst wie die zuvor aufgearbeiteten Anforderungen umgesetzt wurden und welche Kann-Anforderungen nicht in den Rahmen des Projekts gepasst haben aber die Anwendung im Rahmen eines Ausblicks erweitern könnten.

2 Darstellung der Grundlagen

In den folgenden Unterkapiteln werden Grundlagen erläutert, die zum Verständnis der in Kapitel 3 dargestellten Umsetzung des Projekts beitragen sollen.

Zunächst werden verschiedene Softwarearchitekturstile vorgestellt, wobei erörtert wird, welche der Stiele für das Projekts am geeignetsten sind.

Daraufhin werden Security- und Trust Boundary Technologien in gleicher Weise verglichen. Zuletzt werden die verschiedenen Prinzipien der Quarkuserweiterungen erläutert, die in der Implementierung eingesetzt werden.

2.1 Softwarearchitektur

In diesem Abschnitt wird auf verschiedene Softwarearchitektur und Softwareprinzipien eingegangen werden.

Insbesondere werden hierbei das Entity-Control-Boundary Pattern und Onion- bzw. Hexagonal-Architekturen betrachtet, sowie das Backend for Frontend Pattern.

Ebenso wird ein kurzer Überblick über die SOLID Prinzipien gegeben und er-klärt was im Richardson Maturity Model als Level 2 und Level 3 zu verstehen ist.

2.1.1 Backend for Frontend (Peter)

BFF (Backend for Frontend) beschreibt einen Softwarearchitekturansatz, der zum Einsatz kommt, wenn bei einer Anwendung für verschiedene Plattformen verschiedene Ansichten oder Funktionalitäten angeboten werden sollen. Zum Beispiel würde sich die Desktopansicht gegebenenfalls von der Mobilansicht unterscheiden und unterschiedliche API-Aufrufe tätigen.

Bei einem einzelnen API-Backend könnte das zu einer Bottleneck-Problematik führen. Außerdem wäre es wünschenswert nicht mehr Daten pro Anfrage zu transportieren als nötig, was bei einer General-Purpose-API schwierig ist. Bei BFF wird die General-Purpose-API aufgeteilt in Services für verschiedene Funktionalitäten und Plattformen.

Im Rahmen dieses Projekts wird kein Frontend entwickelt. QuizFest ist als Webanwendung angedacht, wird aber serverseitig keine signifikanten Abgrenzungen der Funktionalitäten für Desktop und mobile Endgeräte aufweisen. Deshalb passt dieser Ansatz nicht zu dem umzusetzenden Projekt.

2.1.2 Onion-/ Hexagonal Architekturen (Peter)

Ein gängiger Ansatz des Softwaredesigns ist die Schichtenarchitektur. Diese kann den Eindruck vermitteln, dass die Datenbank das Fundament der Software ist. Dadurch kann gegebenenfalls eine Abhängigkeit vom fachlichen Code der Domain-Layer und technischem Code im Data-Access-Layer bestehen. Das zwingt einen manchmal fachlichen Code anzupassen, wenn sich nur technische Infrastruktur geändert hat.

Durch das Dependency Inversion Pattern lässt sich dies vermeiden und Entkoppelung schaffen. Man hat jedoch oft noch mehr Schnittstellen außer UI und Datenbank. Häufig

werden noch externe APIs, Logging in Dateien oder ggf. E-Mail-Benachrichtigung mit-einbezogen.

Alistair Cockburn begann Mitte der 1990er Jahre diese Architektur mit Hexagon zu visualisieren:

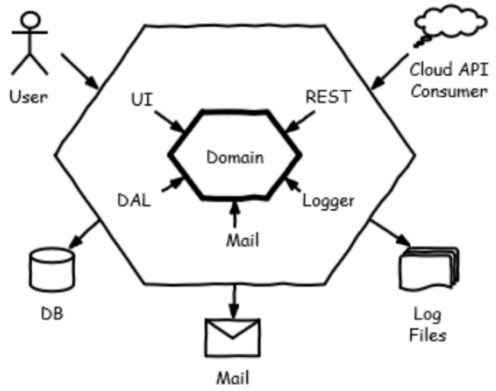

Abbildung 1: Hexagonal Architektur

[@]https://www.maibornwolff.de/sites/default/files/7_hex.png

Das Innere beschreibt die Domain und das Äußere deren Implementierung. Diese Schicht erfüllt Adapterfunktion zwischen Anwendungskern und User, DB, Logdateien, Fremdsystemen etc.

Jeffrey Palermo beschreibt 2008 die Onion-Architektur, wo sich von dem 6-Eck entfernt wird und die Architektur in die Ebenen Domain-Model, Domain-Services, Application-Services. Dabei verlaufen die Abhängigkeiten von außen nach innen, sodass das Domain-Model frei von allen Abhängigkeiten ist.[1]

Diese Architekturen machen Sinn, wenn viele externe Systeme, wie zum Beispiel APIs in die Architektur mit einbezogen werden sollen. Da außer durch Keycloak keine weiteren Abhängigkeiten zu Fremdsystemen bestehen, ist weder die Onion- noch die Hexagonal Architektur passend für dieses Projekt.[2]

2.1.3 Entity-Control-Boundary (Bernhard)

Beim Entity-Control-Boundary Pattern handelt es sich um eine Ausprägung des Model-View-Controller Patterns. Hierbei entspricht die Entity dem Model, die Boundary der View und Control dem Controller.

Zur Schichtung des Modells wird zunächst der fachliche Aspekt betrachtet, welche Elemente zu welchem Modul zugehörig sind, wird aus den fachlichen Grundlagen abgeleitet.

Anschließend wird innerhalb dieser fachlichen Module eine technische Schichtung vorgenommen. Die Entitäten innerhalb des Entity-Layer leiten sich aus den im Domainmodell vorhandenen Objekten ab.

Die Services des Control-Layer stellen die Umsetzung der verschiedenen Use Cases der Anwendung dar. Der Boundary-Layer bietet eine Schnittstelle nach außen, dies

[1] [@Pal]
[2] [@mai]

kann direkt in Form einer Mensch-zu-Maschine Schnittstelle passieren, bspw. mit einer grafischen Nutzeroberfläche.[3]

Im Fall dieses Projekts ist die Schnittstelle in der Form Maschine-zu-Maschine, um in einem aufbauenden Projekt eine Webanwendung mit Client Side Rendering bereitstellen zu können.

Das ECB Pattern bietet sich vor allem für Serviceorientierte Architekturen an, es trennt die zu Persistierenden Daten (Entitys) von der Logik (Control) des Systems. Dies setzt das prozedurale Softwareparadigma im großen Umfang um.

Im Gegensatz dazu werden beim Domain Driven Design Ansatz die Daten mit den Prozessen kombiniert, was wiederum dem objektorientierten Softwareparadigma entspricht.[4]

2.1.4 SOLID Prinzipien (Bernhard)

Bei den SOLID Prinzipien handelt es sich um eine Sammlung von Softwarearchitektur Prinzipen im Folgenden soll ein kurzer Überblick über Single Responsibility, Open Closed, Liskov Substitution, Interface Segregation und Dependency Inversion als Prinzipen gegeben werden.[5]

Single Responsibility

Ein Softwarebaustein hat eine klar definierte Verantwortlichkeit, so werden Änderungen lokal gehalten dies verbessert die Wartbarkeit der Software.

Single Responsibility kann erreicht werden, indem die Softwarestruktur die Organisation des Nutzers abbildet.

Open Closed

Bertrand Meyer schreibt dazu "Ein Softwareartefakt sollte offen für Erweiterung sein, aber geschlossen vor Modifikation."[6]

Das Verhalten lässt sich also anpassen, ohne das Source-Code geändert werden muss.

Abstraktion werden festgelegt, sodass konkrete Implementierungen ausgetauscht werden können.

Hinzu ist es sinnvoll vorerst nur das umzusetzen was gefordert, wenn mehr benötigt wird dann erfolgt erst ein überarbeitetes Design nach dem Open Closed Prinzip.

Liskov Substitution

Objekte eines Typen können durch Objekte eines Subtyps ersetzt werden. Auf diesem Prinzip beruhend können Programme auf Basis gemeinsamer Typen anpassbar gemacht werden.

Verwendung findet dies beispielsweise im State-Pattern, um Verhaltensweisen des Codes basierend auf zuständen zu ändern.

---

[3] Quelle: Foliensatz 2
[4] [@SJU]
[5] Quelle: Foliensatz 3
[6] Meyer, Bertrand (1988). Object-Oriented Software Construction

Interface Segregation

Schnittstellen sollen für Softwarebausteine bereitgestellt werden, die konkret benötigt werden. Viele verschiedene Schnittstellen deren Funktion eindeutig ist, sind besser als ein generelles Interface, welches alle Funktionalitäten beinhaltet.

Dependency Inversion

Abhängigkeiten sollten auf abstrakten Softwareteilen basieren anstelle direkt auf konkreten Softwarebausteinen zu basieren. Durch diese Art der Abstraktion entsteht eine lose Kopplung zwischen einzelnen Bausteinen. Dadurch werden Abhängigkeiten zur Compile-Zeit umkehrbar gemacht.

2.1.5 Richardson Maturity Model (Bernhard)

Das Richardson Maturity Model klassifiziert Web-APIs in vier verschiedene Level. Es unterteilt die Best-Practice von RESTful Design in drei Schritte: Ressourcen Identifikation (URIs), HTTP-Verben, und Hypermedia-Controls (bspw. Hyperlinks).

Im Folgenden werden Level 2 mit HTTP-Verben und Level 3 mit Hypermedia-Controls genauer erläutert.[7]

Level 2 – HTTP-Verben

HTTP Verben werden wie in HTTP selbst (oder so nah wie möglich) verwendet.

GET wird verwendet, um Informationen anzufragen, es ist eine sichere Operation, die keine signifikanten Änderungen an Zuständen vornimmt. Dies bedeutet, dass GET beliebig oft verwendet werden kann und immer dasselbe Ergebnis erhält.

POST wird verwendet, um eine Ressource zu erstellen, dabei liefert ein Erfolg 201 mit der Location URI, unter der der Zustand der Ressource abgefragt werden kann. Wenn etwas fehlschlägt, wird dies mit dem entsprechenden Fehlercode 4xx mitgeteilt.

Ebenso kann PUT verwendet werden, um Ressourcen zu bearbeiten, im Gegensatz zu POST welches immer neu Ressourcen erstellt, sollte PUT immer dasselbe Ergebnis erzielen, auch wenn im Gegensatz zu GET Änderung an Zuständen erzielt werden.

DELETE kann verwendet werden, um Ressourcen zu löschen.

Level 3 – Hypermedia Controls

HATEOAS (Hypertext As The Engine Of Application State)

Das Level adressiert die Fragestellung wie man von einer Ressource zu einer andern gelangt.

Zu Antworten werden Link Elemente hinzugefügt, über die die nächste Ressource aufgerufen werden kann. Die Idee hinter Hypermedia Controls ist, dass sie da legen, was als nächstes gemacht werden kann und welche Ressource (URI) benötigt wird, um den Schritt auszuführen.

[7] [@Wik]

Ein Vorteil hiervon ist, dass es dem Server gestattet das URI Schema zu ändern, ohne dass der Client kaputtgeht, solange der Client die bereitgestellten Links verwendet.

Ein weiterer Vorteil ist, dass es Client-Developern die Möglichkeit bietet, einfach das Protokoll zu erkunden, da die Links Hinweis darauf geben, was als nächstes möglich ist. Somit ist die API dem Ziel der Selbstdokumentation näher.

Außerdem ermöglicht es dem Server neue Fähigkeiten anzubieten, indem die Links dafür bereitgestellt werden.[8]

[8] [@Mar]

2.2 Security

Im Folgenden werden Security Technologien vorgestellt, die für die API in Erwägung gezogen wurden. Dabei werden Gründe benannt, die entweder gegen die entsprechende Technologie sprechen und deshalb nicht zur Umsetzung des Projekts verwendet wurden oder solche, die für die Verwendung sprechen. Es werden http-Basic, Microprofile JWT, OAuth2, OIDC, Input Bean Validation, RBAC und ABAC betrachtet.

2.2.1 HTTP Basic (Peter)

Die Basic Authentication ist ein einfaches Authentifizierungsschema, dass in das HTTP-Protokoll eingebaut ist. Dabei sendet der Client eine Anfrage, die im Header das Wort Basic enthält, dass gefolgt wird von einem String, der mit base64 kodiert ist und Username sowie Passwort enthält. [9]

Ein Problem dabei ist, dass base64 sehr leicht zu decodieren und bietet keine Verschlüsselung. Die Zugangsdaten werden unverschlüsselt an den Webserver übermittelt und gegebenenfalls gecached. [10]

Diese Punkte entsprechen nicht der Vorstellung der Security für das Projekt, weshalb reines http-Basic nicht für die Implementierung verwendet wird.

2.2.2 Microprofile JWT (Peter)

MP JWT ist ein zustandsloser Authentifizierungsansatz, der JWT mit jedem Request mitgibt. Die Eclipse MP JWT Auth stellt sicher, dass der Security Token aus dem Request extrahiert, validiert und ein Security Kontext aus der entnommenen Information kreiert wird. [11]

JWT steht dabei für JSON-Web-Token und ist ein offener Standard, um abgesichert Informationen zwischen Parteien in Form eines JSON-Objects zu übermitteln.

Diese Information wird digital signiert. Signiert wird entweder über ein Secret (HMAC Algorithmus) oder über ein public/private Key-Pair bei Verwendung von RSA oder ECDSA.

Die signierten Tokens verifizieren die darin geschachtelten Claims, während verschlüsselte Tokens diese vor anderen Parteien verstecken. Wenn ein Public-Key-Verfahren angewendet wird, wird zusätzlich verifiziert, dass die Partei mit dem private Key signiert hat.[12]

Nach Login enthält jede Anfrage ein JWT, die dem Benutzer bestimmte Pfade, Ressourcen & Services freigibt.

---

[9] [@swg]

[10][@msc]

[11][@phr]

[12][@jwt]

Mit der Quarkuserweiterung SmallRye JWT lassen sich JSON-Web Token verifizieren und als MicroProfile JWT repräsentieren. Im Anwendungsfall mit Quarkus benötigt es dabei an Annotationen die RBAC umsetzen. Da man den Issuer für die JWTs selbst bereit stellt müssen zusätzlich Public- und Private-Keys generieren lassen, um die JWTs generieren zu können. [13]

2.2.3 Access Control (Bernhard)

Es gibt verschiedene Arten der Zugriffskontrolle (Access Control), hier werden die Rollenbasierte Zugriffskontrolle und die Attributbasierte Zugriffskontrolle miteinander verglichen.

Role Based Access Control (RBAC)[14]

Das Prinzip der rollenbasierten Zugriffskontrolle basiert darauf, dass Nutzern bestimmte Rollen zugewiesen werden.

Dies Rollen können sich beispielsweise aus Rollen innerhalb einer Organisation ableiten. Dieses Vorgehen ermöglicht es den Nutzern des Systems die Rollen einfach zu begreifen.

Es entsteht ein einfacher Ansatz bei dem Nutzern-Rollen zugewiesen werden können und es vermieden wird individuelle Rechte zu vergeben. Somit werden Nutzer in Rollen gruppiert, was die Handhabung gesamter Gruppen erleichtert.

Der Best Patrice Ansatz ist es Nutzern minimale Rechte zu geben, sodass sie genau ausreichend Berechtigungen haben, um ihre zugewiesenen Funktionen ausüben zu können.

Es ist eine besonders geeignete Zugriffskontrolle im Zusammenhang mit Domain-Driven-Design, da sich dort Rollen einfach ableiten lassen.

Attribute Based Access Control (ABAC)[15]

Attributbasierte Zugriffskontrolle bietet sehr viele Möglichkeiten Rechte anhand verschiedenen Variablen kontrolliert zu vergeben.

Das Risiko unbefugter Zugriffe kann verringert werden, da der Zugriff genauer kontrolliert werden kann.

Anstelle Nutzer einer bestimmten Rolle immer Zugriff auf eine Ressource zu geben, werden bei ABAC Zugriffe anhand von Attributen weiter eingeschränkt.

In einem Unternehmen könnten diese Attribute etwa bestimmte Zeiten sein oder Zugehörigkeit zu einem Standort.

---

[13] [@Qua1]

[14] [@Aut1]

[15] [@dns]

Vergleich

RBAC ist die unkompliziertere Art der Zugriffskontrolle der beiden Filtermethoden und kann schneller ausgeführt werden.

In vielen Fällen kann es sinnvoll sein einen hierarchischen Ansatz zu verfolgen, bei dem zunächst eine grob granulare Filterung mit RBAC vorgenommen wird, und anschließend eine feingranulare Filterung mit ABAC verwendet wird.

Somit können die Vorteile beider Methoden vereint werden.

2.2.4 OpenID Connect (Bernhard)

OpenID Connect ist ein einfacher Identitäts-Layer auf Basis des OAuth2 Protokolls. Es ermöglicht Clients die Identität von Endnutzern zu verifizieren, basierend auf Authentifizierung eines Autorisierungs-Servers, sowie erhalten von grundlegenden Profilinformationen des Endnutzers in einem interoperable REST-artigen Verfahren.[16]

Die Quarkus OIDC-Erweiterung bietet Unterstützung für Bearer Token, Authorization Code Flow Mechanismen und ist ausgelegt für Multi Tennancy, Interoperabilität und Reaktivität. Hierbei werden ID und Zugangs JWT verifiziert mit einem nachladbaren JWK-Set. Für sowohl JWT als auch binäre Token besteht die Möglichkeit der remote Überprüfung.

Bearer Token können ausgestellt werden von OpenID Connect und OAuth 2.0 kompatible Server wie Keycloak. Bearer Token Autorisierung prüft die Existenz und Validität des Bearer Token, welches wertvolle Informationen liefert, um festzustellen wer den Aufruf tätigt und ob auf die angeforderte HTTP Ressource zugegriffen werden darf oder nicht.[17]

Die Quarkus OIDC-Client Erweiterung bietet die Möglichkeit Access Token von OpenID Connect oder OAuth2 Providern zu erhalten und zu erneuern.

Dabei werden Client-Credentials, Refresh Token und Password zur Ausstellung von Token unterstützt.[18]

Die Client-Filter Erweiterung von Quarkus beruht auf der Client-Erweiterung und bietet JAX-RS OidcClientRequestFilter um das Access Token aus dem OidcClient im HTTP Authorization-Header als Bearer Wert zu setzten. Dieser Filter kann in Microprofile Rest Client Implementierungen registriert werden.

Die Token-Propagation Erweiterung bietet JAX-RS TokenCredentialRequestFilter welche die OpenID Connect Bearer oder Authorization Code Flow Access Token als HTTP Authorization-Header Bearer Wert setzten. Dieser Filter kann verwendet werden, um Accesstoken zu Downstream Services zu propagieren.

Bei OAuth2 handelt es sich um ein Autorisierungsframework, welches es Anwendungen ermöglicht Zugang zu HTTP Ressourcen im Interesse von Nutzern zu erlangen. Es kann beispielsweise verwendet werden, um Anwendungen zu implementieren, deren Token

[16] [@oid]
[17] [@Qua2]
[18] [@Qua3]

basierte Authentifizierung an externe Server delegiert wird. Diese liefen dann ein Token für den Authentifizierungskontext.

Bei Quarkus wird Elytron verwendet, um hauptsächlich Einblick in remote verschleierte Token zu erhalten. Im Zusammenhang mit JWT Bearer Token ist Quarkus besser darauf ausgelegt mit den OpenID Connect oder SmallRey JWT Erweiterungen verwendet zu werden . [19]

Zur Erläuterung Bearer Token sind Strings, die undurchsichtig sind, um keine Bedeutung für den Client, der sie benutzt zu haben. Sie sind die verbreitetste Form des Access Tokens und werden verwendet, um zu informieren, dass der Träger des Tokens autorisiert ist die API zu verwenden. Typischerweise enthält das Token einen Verweis auf Informationen im Persistierten Speicher des Servers.[20]

JSON Web Token (JWT) enthalten Informationen über eine Entität in Form von Claims. Sie sind in sich geschlossen, darum muss der Server nicht angefragt werden, um das Token zu validieren.[21]

Authentication Code Flow beschreibt den Ablauf, bei dem zunächst eine Standardwebanwendung verwendet wird, bei der der Nutzer sich anmeldet, anschließend authentifiziert sich der Nutzer und stimmt den Bedingungen zu, die Webanwendung generiert einen Autorisierungscode eine Client-ID und ein Client-Secret mit der die eigene Anwendung später die abgestimmten Daten aus der Webanwendung abfragen kann.[22]

Multi Tennancy beschreibt, dass eine einzige Instanz derselben Software für mehrere Accounts verwendet wird. Ein Tennant kann ein einzelner Nutzer sein, bezieht sich üblicherweise auf Kundenorganisation, dieser Tennant hat grundlegenden Zugang zur Anwendung und Privilegien innerhalb. Die Daten der Tennants sind voneinander isoliert, sodass Datensicherheit und Privatsphäre gesichert sind. [23]

Keycloak ist ein Open Source Identitäts- und Zugangsmanagementlösung, es richtet sich an moderne Anwendungen und Services.

Es bietet Single-Sing On wodurch Nutzer sich mit Keycloak authentifizieren anstelle bei jeder einzelnen Anwendung. Es werden ebenfalls Möglichkeiten zum Social-Login geboten. Da es auf Standardprotokollen basiert werden OpenID Connect, OAuth2.0, und SAML unterstützt. Neben RBAC gibt es präzise Autorisierungsservices in Keycloak durch genaues festlegen des Regelwerks in der Admin-Konsole. [24]

2.2.5 Bean Validation (Peter)

Durch Bean Validation lassen sich Input und Output von Restservices, sowie Parameter und return-values von Methoden in Businessservices validiert werden.

[19] [@Qua4]
[20] [@oau]
[21] [@Aut2]
[22] [@Aut3]
[23] [@IMB]
[24] [@Kyc]

Dabei können bestimmte Constraints auf Ebene der Objektmodelle in Form von Annotationen gesetzt werden. Neben den Standard-Annotationen können auch benutzerdefinierte Constraints angelegt werden.

Am Ende einer Validierung kann ein Bericht über die Sammlung aller Verletzungen der Constraints zusammengestellt werden, falls welche auftreten.

Da in der API sämtliche Eingaben der Quizze und dessen Fragen geprüft werden müssen und einige Attribute bereitgestellt werden müssen, wenn Quizze erstellt werden, ist Bean-Validation eine passende Technologie, die bei der Umsetzung der API eingesetzt wird.[25] [26]

[25] [@Qua5]
[26] [@bnv]

2.3 Quarkus-Technologien

Im Folgenden wird ein Überblick über die verwendeten Quarkus-Technologien gegeben.

Dabei wird ein Blick auf die Verwendung von Hibernate mit der Verwendung von Panache gegeben, sowie eine Beschreibung zu Swagger UI mit Open API, Tests mit RETS Assured und Context Dependency Injections.

2.3.1 Hibernate ORM mit Panache (Bernhard)

Hibernate ist eine JPA Implementation und bietet die volle Bandbreite eines Objekt Relationale Mapper (ORM). Es ermöglicht komplexes Mapping, jedoch sind einfache oder übliche Mappings weiterhin nicht trivial.

Hibernate ORM mit Panache legt den Fokus darauf, dass Entitäten trivial und einfach zu schreiben sind.

Es bietet die Möglichkeit der Verwendung des Repository Pattern, dazu werden die Entitäten mit @Entity annotiert und das Repository implementiert PanacheRepository<T>.

Jedes Repository ist auf ein Entity Typ zugeschnitten, dadurch bleiben die Operationen Typsicher.

Dies ermöglicht es über das Repository alle durch Panache bereitgestellten Methoden verwendet werden. Einige Beispiele sind, persist() zum Speichern einer Entität, delete() zum Löschen oder listAll() um alle Entitäten aufzulisten.

Nützlich ist find() welches es ermöglicht mit parametrisierten Queries nach Entitäten zu suchen.[27]

```
//find a Question with a parametrized Query
Question q = find("quiz_id = ?1 and questionnr = ?2", quizID,
      questionNr).firstResult();
```

Snippet 1: Beispielhafte Verwendung parametrisierte Queries

Alle list() Methoden haben ein stream() Äquivalent, mit der Möglichkeit die Entitäten zu streamen, so können Streamfilter und Mapping verwendet werden.

Die Methoden zum Finden von Entitäten bieten ebenfalls Möglichkeit zur Pagination um zu verhindern das zu große Mengen an Entitäten auf einmal geladen werden.

```
//Pagination with Panache
PanacheQuery<Quiz> ownQuizzes = find("creatorname", UserName);
dtos = ownQuizzes.page(Page.of(page, pageSize)).stream()
      .map(q -> quizToListDTO(q)).collect(Collectors.toList());
```

Snippet 2: Beispielhafte Verwendung Pagination mit Streammapping

[27] [@Qua6]

2.3.2 OpenAPI und Swagger UI (Bernhard)

Mittels OpenAPI-Spezifikationen können Anwendungen und deren API-Beschreibungen verfügbar gemacht werden. Außerdem könne mit Swagger UI die API-Endpunkte nutzerfreundlich getestet werden.

So kann eine einfache und schnelle Visualisierung der API-Operationen und Schemata gegeben werden.

Mit zusätzlichen Annotationen können weitere Informationen zu Operationen gegeben werden. Über @Tag können Operationen beispielsweise gruppiert werden.

Mit @Operation können Detailinformationen zu einer Operation geliefert werden, darunter können z.B. unterschiedliche Response und deren Bedeutung dokumentiert werden.

Es können Methoden Beschreibungen hinzugefügt werden oder Beschreibungen zu Parametern gegeben werden.

Mit der @OpenAPIDefinition können Informationen über die gesamte API gegeben werden, wie beispielsweise Sicherheitsanforderungen, die für die gesamte Anwendung gelten.[28]

2.3.3 REST Assured (Peter)

REST Assured ist ein Testing-Framework, das einem ermöglicht Endpunkte eines REST-Services abzufragen und zu validieren, ob erwartete Responses eintreten. An einer Response können mehrere Eigenschaften, wie der Inhalt des Response-Bodies, -Headers, Statuscodes und Statusmessages validiert werden.[29]

Mit REST Assured lassen sich Tests nach Behavioral-Driven-Design Pattern formulieren, sodass jedes Szenario in dem Muster Given, When, Then aufgebaut ist. „Given“ gibt den Kontext beziehungsweise die Vorbedingungen an. Mit „when“ wird die getestete Aktion beschreiben und „then“ markiert das Ergebnis, das überprüft wird.[30]

Ein Beispiel für ein Testszenario ist in Snippet 3 zu sehen. Nach „given()“ werden die Vorbedingungen getroffen, dass im Requestbody ein Objekt vom ContentType JSON übergeben wird. „when“ wurde in diesem Fall mit der Methode post() ersetzt, womit die Aktion ein Post-Request auszuführen festgelegt wird. Nach „then()“ soll überprüft werden, ob der nach der Anfrage zurückgegeben Statuscode derselbe wie erwartet ist.

[28] [@Qua7]
[29] [@Qua8]
[30] [@MSe]

```
given().contentType(ContentType.JSON)
.body(quiz)
.post("/quiz-fest/api/quizzes")
.then()
.statusCode(400);
```

Snippet 3: REST Assured Beispiel

2.3.4 CDI Injection (Peter)

Durch das Dependency-Injection Pattern, wird eine lose Kopplung realisiert die Instanzvariable einer Klasse mit @Inject annotiert wird bei Laufzeit wird die Annotierte Instanzvariable erzeugt, sodass keine Konstruktor Aufrufe benötigt werden. In Quarkus ist Dependency Injection basiert auf ArC, was eine CDI-basierte Dependency Injection Lösung ist und auf Quarkus zugeschnitten ist.[31]

[31] [@Qua9]

3 Anwendung

In diesem Kapitel wird die entwickelte Anwendung vorgestellt. Dazu werden im ersten Teil Anforderung an die API festgelegt und Prinzipien der Domain beschreiben, die in der Implementierung umzusetzen ist. Anschließend wird die Strukturierung der API dargelegt.

Hierbei wird genauer auf die Modellierung eingegangen, indem die Einzelnem Packages mit den jeweiligen Technologien durchgegangen werden.

Zuletzt wird beschrieben, wie beim Testen und Dokumentieren vorgegangen wird.

3.1 Anforderungen und Prinzipien von QuizFest (Peter)

Anlegen eines Quizzes

Ein User soll ein Quiz erstellen können. Dazu muss dieser User registriert sein, damit sich das Quiz dem User zuordnen lässt. Der User muss dabei eine Kategorie angeben, die bereits existiert, damit beim Suchen nach einem Quiz nach einer Kategorie gefiltert werden kann.

Die Quizze unterliegen beim Erstellen bestimmten Kriterien und Beschränkungen. Ein Quiz muss mindestens eine Frage beinhalten, einer existierenden Kategorie zugeordnet werden und einen nicht leeren Quiz Titel besitzen. Jede Frage im Quiz muss einen nicht leeren Fragetext beinhalten und mehr als eine Antwortmöglichkeit anbieten. Jede Antwort darf ebenfalls keinen leeren Antworttext vorweisen und innerhalb der Antworten einer Frage muss eine Antwort als korrekte Antwort markiert sein.

Bearbeiten eigener Quizze

Als Ersteller eines Quizzes soll man auf seine eigenen Quizze zugreifen können und sie bearbeiten können. In der Übersicht aller eigenen Quizze werden die Quizze mit ihrem Titel und Links zum Editieren und Spielen der Quizze dargestellt. Beim Editieren eines Quizzes sollen die gleichen Bedingungen für ein valides Quiz gelten wie beim Erstellen. Außerdem sollen neben dem Quiz auch nur einzelne Fragen bearbeitbar sein können.

Wählen und Spielen eines Quizzes

Ein Quiz muss über seine Kennung für User auffindbar sein, um ein Spiel zu beginnen. Dem spielenden User müssen die Fragen mit ihnen Antwortmöglichkeiten nacheinander angezeigt werden. Nach der Darstellung der Frage soll der Nutzer eine gewählte Antwort angeben können und danach eine Rückmeldung darüber bekommen, ob die gewählte Antwort eine korrekte war. Für jede richtig beantwortete Frage sollen Punkte vergeben werden. Falls der Nutzer die Frage falsch beantwortet hat, soll angezeigt werden welche Antworten korrekt gewesen wären. Nach Beantwortung einer Frage soll zur nächsten Frage im Quiz weitergeleitet werden.

Kategorien und Administrator Rolle

Kategorien, in denen Quizze von Usern angelegt werden, sollen nicht von Usern bearbeitbar sein. Es muss also eine separate Administrator Rolle geben, in der Kategorien angelegt und gelöscht werden können. Dabei darf eine Kategorie nur angelegt werden, wenn ihre Benennung nicht die gleiche wie eine bereits existierende Kategorie ist. Gelöscht werden darf eine Kategorie nur, wenn sie keine Quizze enthält, da diese sonst nicht mehr über ihre Kategorie auffindbar wären.

Usersystem

Neben der Administrator Rolle muss es ein Usersystem geben, wodurch User mit ihren Zugangsdaten voneinander unterschieden werden können. Dadurch wird zum Bearbeiten eines Quizzes nur der Nutzer dazu berechtigt, der das Quiz erstellt hat, da der Username eindeutig einem Quiz bei seiner Erstellung zugeordnet wird.

3.2 Aufbau der API (Bernhard)

/category/	
GET	Liste der Vorhandenen Kategorien
PUT	--
POST	--
DELETE	--

/category/{categoryName}	
GET	Liste der Quizzes in angegebener Kategorie
PUT	--
POST	Neue Kategorie mit angegebenem Namen erstellen
DELETE	Leere Kategorie löschen

POST und DELET nur für Nutzer mit category-admin Rolle

/quizzes/	
GET	Liste der eigenen Quizzes
PUT	--
POST	Neues Quiz erstellen
DELETE	--

/quizzes/{QuizID}/edit	
GET	Quiz mit angegebener ID in bearbeitbarer Form holen
PUT	Quiz mit angegebener ID ändern
POST	Neue Frage zum Quiz hinzufügen
DELETE	Quiz mit angegebener ID löschen

/quizzes/{QuizID}/edit/{QuestionNr}	
GET	Frage der angegebenen Nummer aus Quiz holen
PUT	Frage der angegebenen Nummer ändern
POST	--
DELETE	Frage der angegebenen Nummer löschen

Operationen der edit Ressource nur für den ursprünglichen Erstelle zugelassen

/quizzes/{QuizID}/play	
GET	erste Frage des Quizzes mit ID holen
PUT	--
POST	--
DELETE	--

/quizzes/{QuizID}/play/{questionNr}	
GET	Frage von Nummer aus Quiz mit ID holen
PUT	--
POST	Frage beantworten mit Antwort Nummer liefert richtige Antworten und vergebene Punkte
DELETE	--

GET Operationen die Listen zurückgeben bieten Pagination.

3.3 Modellierung (Bernhard + Peter)

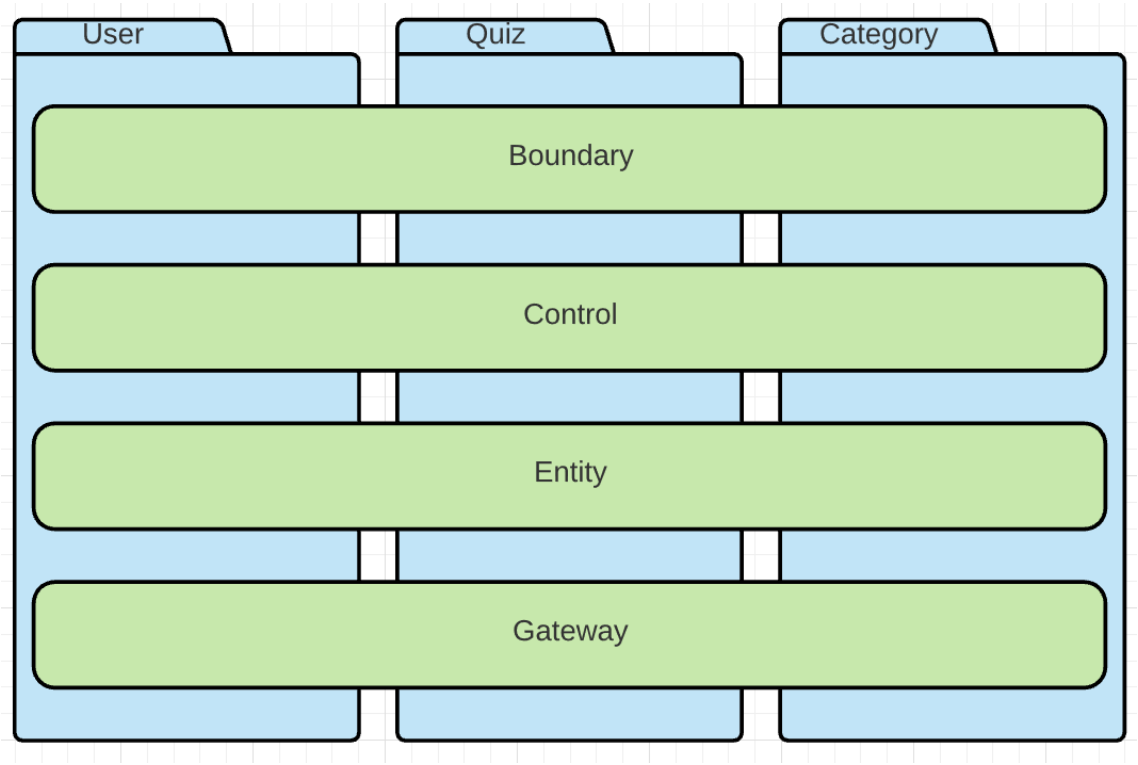

Abbildung 2: Module mit technischer Sichtung nach ECB

Da das Usersystem, die Quiz-Domain und das Kategorie Management keine logischen Einheiten bilden, werden sie in der Modellierung in einzelne Module aufgetrennt, um eine fachliche Trennung zu realisieren. Innerhalb der einzelnen Module wird jeweils eine technische Schichtung nach Entity-Control-Boundary-Gateway vorgenommen (siehe Abbildung 2).

Dadurch werden die zu persistierenden Daten, pro fachlicher Ebene von der Logik getrennt. Im Falle des Quiz-Moduls sind zu speichernde Entitäten wie ein Quiz von der Logik in der Control getrennt, die beispielsweise die Bedingungen bei der Erstellung eines Quizzes überprüft.

Im User Modul sind durch den Einsatz von OIDC und Keycloak technische Schichten wie Gateway und Entity weggefallen, da Keycloak das Organisation und die Persistierung übernimmt.

Für den QuizFest Anwendungsfall ist der Serviceorientierte Ansatz geeignet, denn bei spielerischen Anwendungen muss die Logik leicht änderbar sein, wenn bestimmte Mechanismen nicht nutzerfreundlich sind oder die Bedienbarkeit erschweren, sollten sie leicht austauschbar sein.

Die Abhängigkeiten zwischen technischen Schichten sind in allen Modulen so organisiert das von der Entity-Ebene aus keine Anhängigkeiten nach außen bestehen, sondern nur aus zugreifenden- und kontrollierenden Schichten wie der Control und der Infrastruktur im Gateway. Somit muss zum Beispiel kein technischer Code in der Data-Access-Layer geändert werden, falls diese vom Code in der Domain-Layer abhängen und sich verändert.

Zwischen die fachlich voneinander getrennten Module wurden jeweils da Anti-Corruption-Layer eingebaut, wo aus einem Modul auf Funktionalitäten aus einem anderen Modul zugegriffen werden muss. Um jedoch eine lose Kopplung zu bewahren, befinden sich in den acl-Packages Interfaces, die in den fremden Modulen implementiert werden. Beispielsweise muss, bei schreibenden Operationen, auf einem Quiz der aktuelle User abgefragt und überprüft werden, ob dieser Nutzer berechtigt ist diese Operationen zu verwenden. Die Funktionalität der Überprüfung wird in einem Service im User-Package umgesetzt, passt aber nicht in die Quizdomain. Damit wird auch das Dependency-Inversion-Prinzip angewandt, wodurch Abhängigkeiten nur zu abstrakten Softwarebausteinen aus dem acl bestehen anstelle konkreter Abhängigkeiten zu Bausteinen aus den anderen Modulen.

In der Control werden jeweils die Use-Cases für die einzelnen Module in Services aufgetrennt. So werden Verantwortlichkeiten gemäß dem Single-Responsibility-Prinzip klar definiert.

3.3.1 User Modul (Bernhard)

Die Verwaltung der User findet hauptsächlich über Keycloak statt. Innerhalb der API gibt es lediglich einen Endpunkt, über den der Zugreifende User erfragt werden kann.

Um Keycloak zu verwenden, wird ein Realm erstellt, innerhalb dieses Realms kann festgelegt werden welche Clients in welcher Weise zugreifen können.

Im Realm können Rollen festgelegt werden, es besteht auch die Möglichkeit diese Rollen auf Ressourcen Zugriff zu erteilen. Innerhalb eines Realm können ebenfalls Nutzer angelegt werden.

Für QuizFest wird bei benötigten Methoden Rollenbasiert der Zugriff festgelegt.

Um Zugriff auf die API zu erhalten, wird erst bei Keycloak ein JWT abgefragt mit diesem Token kann sich dann gegen die Anwendung authentifiziert werden.

Das Token hat eine begrenzte Gültigkeit, es kann allerdings mit einem Refresh-Token erneut angefragt werden.

In der API wird aus der Security Identity, wenn nötig der Nutzername oder die Rolle extrahiert.

Wenn ein Quiz bearbeitet werden soll, wird der Nutzer abgefragt und das Bearbeiten ist nur möglich, wenn dieser Nutzer mit dem Ersteller übereinstimmt.

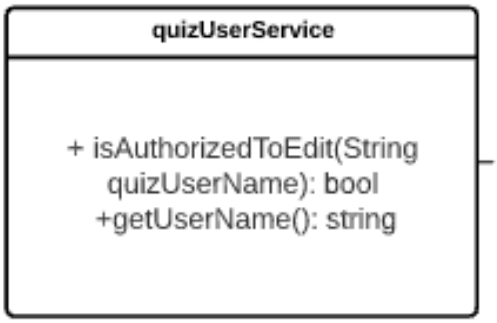

Abbildung 3: QuizUserService im User-Modul

Im Fall von QuizFest ist besonders wichtig, dass nur der ursprüngliche Ersteller eines Quizzes dieses auch bearbeiten kann, um zu verhindern, dass eigene Quizze von anderen verändert werden.

Es soll nur Administratoren gestattet sein, neue Kategorien zu erstellen oder leere Kategorien zu löschen, dies verhindert, dass jeder Nutzer Kategorien erstellen kann, was zu einem Überfluss und Redundanz von Kategorien führen würde.

Deshalb wird für das Bearbeiten von Quizzes ABAC eingesetzt und für das Erstellen neuer Kategorien ist eine RBAC ausreichend.

3.3.2 Quiz Modul (Peter)

Im Quizmodul werden die Funktionalitäten implementiert, die den Zugriff und das Bearbeiten von Quizzen, Fragen und ihren Antworten ermöglicht.

Der Kern des Quizmoduls ist die Domain, die sich in den Quiz-Entitys im Entity-Package abzeichnet, (siehe Abbildung 4).

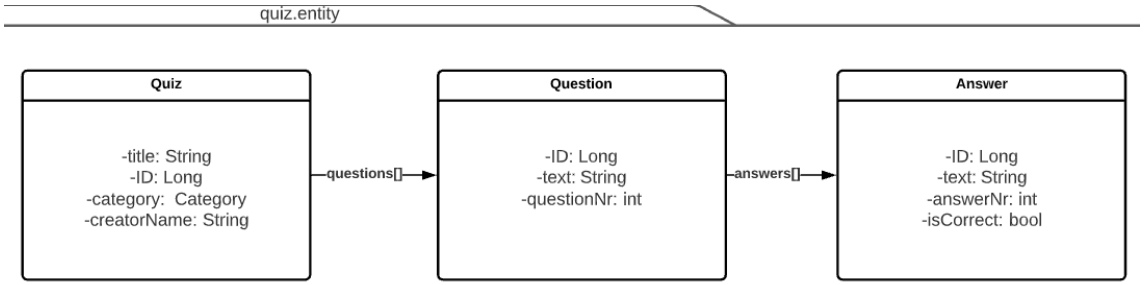

Abbildung 4: Quiz Domain

Ein Quiz besteht immer aus einem Quiztitel, einer zugehörigen Kategorie, dem Username des Erstellers und einer ID, die das Quiz von anderen Quizzes unterscheidet. Ein Quiz besteht außerdem aus einer oder mehreren Fragen.

```java
@Entity
public class Quiz {
    @Id @GeneratedValue(strategy = GenerationType.TABLE)
    @Column(name = "quiz_id")
    private Long id;

    @NotBlank(message = "Quiz Title shall not be blank")
    private String title;

    @NotBlank(message = "Creatorname shall not be blank")
    private String creatorName;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "quiz", orphanRemoval = true)
    @NotEmpty(message = "Quiz must contain questions")
    @NotNull(message = "Questions shall not be null")
    private Collection<Question> questions;

    @ManyToOne
    @JoinColumn(name = "category_name")
    @NotNull(message = "Category shall not be null")
    private Category category;
```

Snippet 4: Quiz Entity

Wie in Snippet 4 zu erkennen, wird die ID eines Quizzes mit der javax.persistence-Annotation @GeneratedValue generiert.

Der Titel des Quizzes, der mit dem String „title“ definiert wird, darf nicht darf beim Erstellen und Bearbeiten eines Quizzes nicht leer übergeben werden. Das wird mit Hilfe von Bean Validation sichergestellt. Mit der Annotation @NotBlank wird überprüft, dass ein String nicht leer ist. Das gleiche wird bei der Übergabe eines des Ersteller Namen geprüft.

Das Quiz enthält des Weiteren ein Category Objekt. Durch @ManyToOne wird deutlich gemacht, dass ein Quiz nur einer Kategorie angehören kann, aber eine Kategorie mehrere Quizze beinhalten kann. Durch die JPA Annotation JoinColumn wird sichergestellt, dass die Kategorie eines Quizzes über den Fremdschlüssel category_name referenziert wird. Auch die Kategorie darf bei der Persistierung eines Quizzes nicht fehlen. Deshalb taucht hier die Validation-Annotation @NotNull auf.

Die Fragen eines Quizzes werden in der Collection „questions“ gehalten. Eine Frage kann nur zu einem Quiz gehören, während zu einem Quiz mehrere Fragen gehören können. Über die Option cascade in @OneToMany wird festgelegt welche JPA Operationen auf das entsprechende Target kaskadiert wird. CascadeType.ALL sagt aus, dass alle Operationen, sprich persist, merge, remove, refresh und detach auch auf „questions“ angewendet werden, wenn sie auf einem Quiz angewendet werden. Referenziert wird das Quiz von einzelnen Question-Objekten über die Objektvariable „quiz“, was durch die Option „mappedBy“ definiert wird. Wird die Option „orphanRemoval“ auf true gesetzt, wird festgelegt, dass eine Question auch gelöscht wird, wenn die Referenz auf ein Quizobjekt besteht, dass zuvor gelöscht worden ist.

Die Question-Entity ist ähnlich aufgebaut, wie Quiz (siehe Question Entity).

```
@Entity
public class Question {
    @Id @GeneratedValue
    @Column(name = "question_id")
    private Long id;

    @NotBlank(message = "Questiontext shall not be blank")
    private String text;
    private int questionNr;

    @OneToMany(mappedBy = "question", cascade = CascadeType.ALL, orphanRemoval = true)
    @NotEmpty(message = "Question must contain Answers")
    @NotNull(message = "Answers shall not be null")
    private Collection<Answer> answers;
    @ManyToOne @JoinColumn(name="quiz_id", nullable = false)
    private Quiz quiz;
```

Snippet 5: Question Entity

Auch die Question-Entität ist durch eine ID einzigartig. Zusätzlich hat eine Question eine Fragenummer (questionNr), mit der alle Questions innerhalb einer Frage nummeriert werden. „text“ enthält die Formulierung einer Frage und darf nicht leer sein, damit die Spielbarkeit des Quizzes nicht verloren geht.

Zu den Antworten besteht dieselbe Dynamik wie zwischen einem Quiz und seinen Fragen. Die OneToMany, Annotaion ist daher gleich gestaltet. Da auch hier aus Gründen der Spielbarkeit Antworten existieren müssen, wenn eine Frage existiert, wurde die Answer-Collection mit @NotEmpty und @NotNull annotiert.

Zum Quiz besteht wiederum dieselbe Beziehung wie zwischen Quiz und Category, da ein Question-Objekt nur einem Quiz zugeordnet werden kann.

Eine Antwort innerhalb einer Frage wird mit der Answer-Entität repräsentiert.

```java
@Entity
public class Answer {
    @Id @GeneratedValue
    @Column(name = "answer_id")
    private Long id;

    @NotBlank(message = "Answertext shall not be blank")
    private String text;
    private int answerNr;
    private boolean isCorrect;
    @ManyToOne @JoinColumn(name="question_id", nullable = false)
    private Question question;
```

Snippet 6: Answer Entity

Ähnlich wie bei einer Frage hat eine Antwort eine ID und einen Antworttext. Diese sind entsprechend analog zu Question annotiert. Anworten werden innerhalb einer Frage annotiert und besitzen deshalb einer Antwortnummer („answerNr“). Zudem haben sie die Eigenschaft entweder eine richtige oder eine falsche Antwort zu einer Frage zu sein. Dies wird im Boolean „isCorrect“ festgehalten. Da mehrere Antworten zu einer eindeutigen Frage zugeordnet werden können wird die Objektvariable „question“ vom Typ Question mit @ManyToOne entsprechend annotiert.

3.3.3 Erstellen und Editieren eines Quizzes (Peter)

Das Erstellen und Bearbeiten von Quizzes wird über QuizzesRessource, EditQuizRessource und EditQuestion Ressource angestoßen. In QuizzesRessource sind die lesenden Zugriffe und Erstellungen eigener Quizze untergebracht. Mit GET auf dieser Ressource werden alle Quizze zurückgegeben, die der aktuelle Nutzer selbst erstellt hat

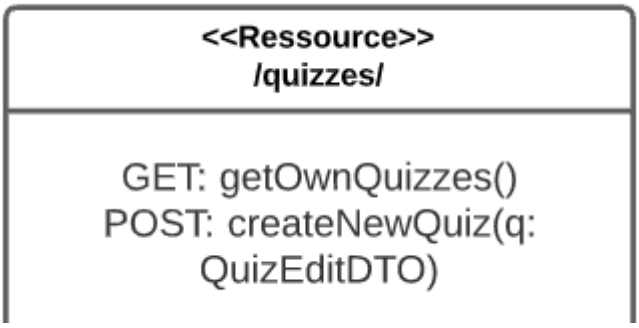

Abbildung 5: Quiz Ressource

(siehe Abbildung 5). Über den editQuizService, der im Gateway implementiert wird anhand des aktuellen Usernames alle Quizzes zurückgegeben, die den entsprechenden creatorName beinhalten.

Für den Datentransfer an die Boundary wird das QuizListDTO

verwendet, wodurch an den Nutzer pro Quiz nur Titel, ein Link zum Editieren und ein Link zum Spielen des jeweiligen Quiz zurückgegeben wird. Dies entspricht nach dem Richardson-Maturity-Models in Ansätzen Level 3.

Möchte man ein neues Quiz erstellen, muss bei der POST-Anfrage ein QuizEditDTO in Form eines JSON-Objekts übergeben werden. Dieses enthält den CategoryName, den Titel des Quizzes, eine Liste aus Question-Objekten, mit jeweils Fragestellungen und einer Liste von Answer-Objekten Antworttext und der Markierung, ob sie korrekt ist. Im EditQuizRepository werden vor der Persistierung alle Bedingungen überprüft, die ein Quiz aufweisen muss. Über die Methode checkValidQuiz werden auf dem speziell für Validierung ausgelegte QuizLogikService alle Verletzungen der Bean-Validation-Constraints gesammelt, die in den einzelnen Entitäten festgelegt sind. Außerdem werden Fälle überprüft, die nicht von der Bean Validation abgedeckt werden. Dazu gehört die Überprüfung, ob jede Frage mindestens eine Antwort aufweist, die als korrekt markiert ist und mindestens eine, die als inkorrekt markiert ist. Damit ist auch die Anforderung abgedeckt, dass eine Frage mehr als eine Antwort enthalten muss. Werden eine oder mehrere Constraints verletzt, werden die sogenannten ConstraintViolations gesammelt und falls welche auftreten in Form einer BadRequestException ausgegeben. Anderenfalls wird der CreatorName über den UserService gesetzt, bei den Frage-Objekten das neu erstellte Quiz als Fremdschlüssel gesetzt und bei den Antwort-Objekten die Fragen als Fremdschlüssel gesetzt. Zuletzt wird das Quiz persistiert.

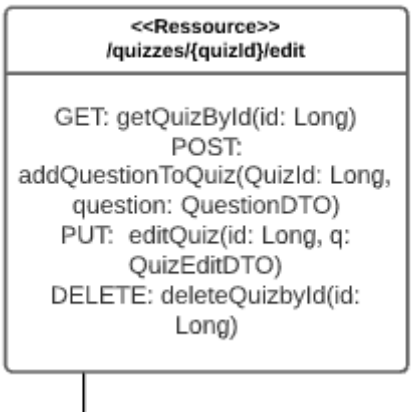

Abbildung 6: EditQuizRessource

Editiert werden kann das Quiz danach über die EditQuizRessource (siehe Abbildung 6). Über GET wird das QuizEditDTO eines einzelnen Quizzes zurückgegeben, dass alle Bestandteile eines Quizzes anzeigt, die bearbeitbar sind. Das heißt, die IDs der einzelnen Objekte werden nicht mitgegeben. Ein Objekt dieser Form lässt sich auch mitgeben, wenn über PUT und editQuiz(QuizEditDTO quiz) ein ganzes Quiz überarbeitet werden soll. Bei einem Update wird, wie beim Erstellen geprüft, ob alle Eingaben den Bedingungen eines validen Quizzes entsprechen. Ein Quiz lässt sich über diese Ressource vom Ersteller löschen und über POST lässt sich mit Übergabe eines QuestionDTOs dem Quiz eine Frage hinzufügen.

Über EditQuestionRessource lassen sich analoge Operationen auf Fragen der Quizze ausführen. Diese lässt sich mit Angabe der QuizId und der QuestionNr der zu überarbeitenden Frage updaten. Auch hier werden vorab alle Kriterien einer Frage validiert. Eine Frage lässt sich auch aus ihrem Quiz entfernen.

Alle Operationen zum Bearbeiten von Fragen werden im EditQuestionRepository realisiert und alle, die hingegen zum Quiz gehören werden im EditQuizRepository umgesetzt.

3.3.4 Verlauf eines Spiels (Bernhard)

Der Ablauf eines Spiels gestaltet sich folgendermaßen, zunächst kann der Client aus einer Liste der Quizze das zu spielende Quiz auswählen. Um die Verwendung zu vereinfachten, enthalten QuizListDTO und QuizForCategoryDTO jeweils einen Link zu spielen. Wenn ein Quiz unter „play“ aufgerufen wird, so leitet die Boundary dies mit einem Auf-ruf von chooseQuiz() an den PlayService in der Control weiter.

Die Implementierung im PlayRepository sucht die erste Question, welche die niedrigste QuestionNr hat und liefert diese zurück. Die Question wird in Form eines QuestionToPlayDTOs übergeben, dies enthält nicht direkt die Antworten, sondern nur den Antworttext und die Antwort Nummern in einer Map, so wird nicht mitgegeben welches die korrekten Antworten sind, um nicht die Antwort vorwegzunehmen. Das DTO bietet zudem den Vorteil, dass die Question Entity nicht geändert werden muss, wenn sich etwas an der Logik des Spielens ändern sollte.

Aus diesen Antworten kann nun der Client die wählen, die er für die Richtige hält. Zum Beantworten wird dann die QuizID die QuestionNr und die gewählte AnswerNr an die Anwendung per POST übermittelt. Hier wird POST verwendet, um vorzubereiten, zu einem späteren Zeitpunkt die erzielten Punkte im Nutzeraccount zu speichern für ein mögliches Leaderboard. Dadurch das QuizID und QuestionNr mitgegeben werden wird vermieden, dass der aktuelle Zustand des Clients im Server gespeichert werden muss. Die Implementation von answerQuestion sucht zunächst die angegebene Frage und Antwort. Sollten diese nicht existieren, wird eine NotFoundExeption geworfen. Sonst wird geprüft, ob die gewählte Antwort korrekt ist und wenn dies der Fall ist, wird ein festgelegter Punktsatz vergeben. Als Rückgabe wird ein ResultDTO zusammengestellt, dies beinhaltet die vergebenen Punkte, alle korrekten Antworten, sowie einen Link zur nächsten Frage, sofern das Quiz eine weitere Frage beinhaltet.

```
public class ResultDTO {
    public Collection<Integer> correctAnswers;
    public int points;
    public String linkToNextQuestion;
}
```

Snippet 7: Aufbau des ResultDTO

Mit diesen Informationen kann der Client, bei Bedarf die nächste Question anfragen und spielen, bis alle Fragen beantwortet sind. Wenn das Quiz beendet ist, kann der Client aus einer Category ein neues Quiz zum Spielen suchen.

3.3.5 Category Modul (Peter)

Das Category-Modul wurde vom Quiz Modul getrennt, obwohl die Kategorie eng mit dem Quiz zusammenhängt. Grund dafür ist, dass die schreibenden Operationen, die auf einer Kategorie ausgeführt werden, nur von Benutzern mit Administrator Rolle ausgeführt werden können. In Snippet 6 sieht man die Annotationen @RolesAllowed mit der Rolle „quizfest-category-table“ über den schreibenden POST und DELETE Methoden, die sicherstellen, dass nur ein Nutzer mit entprechender Rolle Kategorien hinzufügen oder entfernen kann. Diese Rolle wurde über OIDC und Keycloak konfiguriert.

```java
@Transactional
@POST @Path("/{categoryName}")
@RolesAllowed("quiz-fest-category-admin")
@Operation(description = "create a new category is only allowed for administrators")
public Category addNewCategory(@Parameter(example = "NeueKategorie") @PathParam("categoryName") String category){
    return categoryService.addCategory(category);
}

@Transactional
@DELETE @Path("/{categoryName}")
@RolesAllowed("quiz-fest-category-admin")
@Operation(description = "delete category by name is only allowed for administrator if category is empty")
public Response deleteEmptyCategory(@Parameter(example = "NeueKategorie") @PathParam("categoryName")String category){
    if(categoryService.deleteCategoryByName(category)){
        return Response.noContent().build();
    }
    return Response.notModified("Category is Not empty").build();
}
```

Snippet 8: Category-Endpunkte mit Zugriffsbeschränkung

Eine weitere Funktion des Category-Moduls ist das Bereitstellen einer Ansicht für alle Quizze einer bestimmten Kategorie. Dazu wird eine GET-Operation mit einem Kategorienamen als Pathparameter bereitgestellt, die eine Sammlung der passenden Quizze in bestimmter Form zurück liefert. In der Auflistung der Quizze werden, wie in Snippet 7 nur der Titel und der Link zum Quiz dargestellt, um die Übersicht in der Auflistung beizubehalten.

```java
public class QuizForCategoryDTO {
    public String title;
    public String linkToQuiz;
}
```

Snippet 9: QuizForCategoryDTO

Das CategoryRepository wird neben den Operationen auf einer Kategorie auch dafür verwendet, zu überprüfen, ob eine bestimmte Kategorie existiert, bevor sie einem Quiz zugeordnet wird. Deshalb wurde im Quizmodul eine Anti-Corruption-Layer eingebaut, die ein Interface enthält, das die entsprechende Methode anbietet und im gateway-package des Category-Moduls implementiert wird. Die Anti-Corruption-Layer ist im Quizmodul eingebaut, da über dieses Modul auf die Methode zugegriffen wird und von dort aus verwendet wird. Die verwendende Seite soll die Kontrolle über das Aussehen der Schnittstelle haben, während das Category-Package die Implementierung liefert. Das vollständige Category-Modul ist in Abbildung 7 zu sehen.

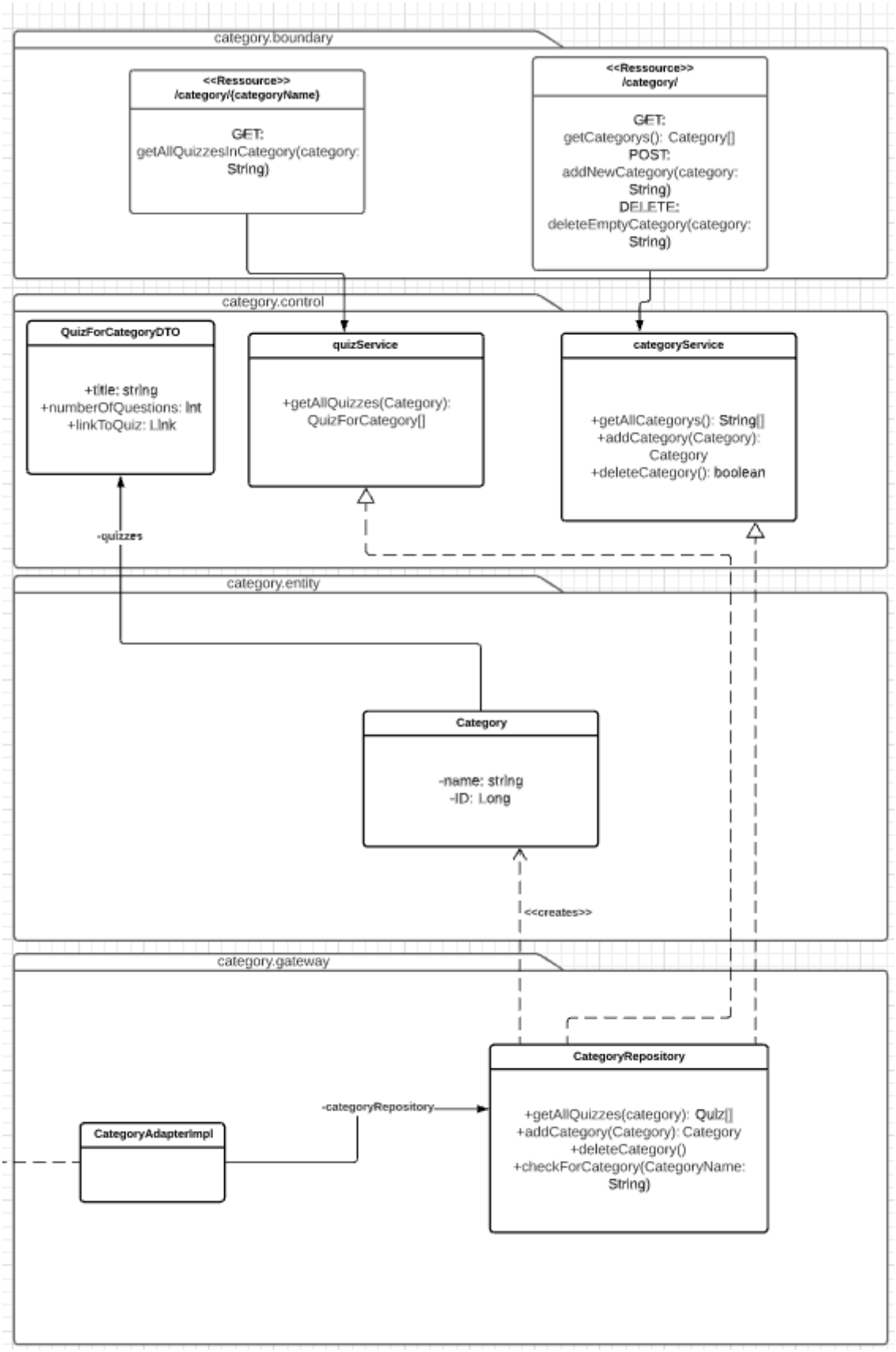

Abbildung 7: Category Modul

3.4 Tests und Dokumentation

Tests und Dokumentation sind ein Integraler Teil der Softwareentwicklung. Sie stellen sicher, dass eine Anwendung funktional korrekt ist und durch Anwendende und zukünftige Entwickelnde in der Art und Weise verwendet werden kann, wie es durch die Entwickler der Anwendung vorgesehen wird.

Im Folgenden wird beschrieben wie Äquivalenzklassentests entworfen wurden und mit REST Assured umgesetzt wurden. Anschließend wird dargelegt, wie mit OpenAPI die QuizFest API möglichst selbstdokumentierend gestaltet ist.

3.4.1 Test mit REST Assured (Peter + Bernhard)

Die Tests wird Äquivalenzklassentests organisiert. Testfälle setzen sich dabei aus ihren Eingaben und ihrer Umgebung zusammen. Für jeden Endpunkt, der getestet wird, werden für alle Input-Möglichkeiten verschiedene Äquivalenzklassen eingeteilt. In eine Äquivalenzklasse gehören alle Eingabefälle, die zum selben Ergebnis führen. In der Regel werden in Testszenarien zunächst die Äquivalenzklassen abgedeckt, bei dessen Zusammensetzung ein gültiges Ergebnis erwartet wird.

Als Beispiel wird der Testfall bei Bearbeitung einer Question eines Quizzes vorgestellt. Im Use Case muss beachtet werden, dass alle Attribute der übergebenen Question den Bedingungen der Domain entsprechen. Für all diese Bedingungen gibt es den Fall, dass diese erfüllt werden und eine gültige Eingabe erfolgt oder dass ungültige Werte übergeben werden. Die QuizID des zu überarbeitenden Quizzes muss bereits zu einem existierenden Quiz gehören. Wenn die angegebene ID noch zu keinem Quiz gehört, gilt dies als ungültige Eingabe und muss mit entsprechenden Fehlercode so markiert werden. Daraus ergeben sich bereits 2 Äquivalenzklassen, die alle zwei möglichen Szenarios liefern, die bei der Eingabe einer QuizID erzielt werden können. Eine ungültige Äquivalenzklasse wird jeweils immer mit gültigen Werten anderer Klassen überprüft. So wird sichergestellt, dass durch diese Eingabe der erwartete Fehler ausgelöst wird. [32]

Analog zur QuizID wird auch die QuestionNr eingeteilt. Entweder in dem Quiz gibt es eine Frage mit entsprechender Nummer oder nicht. Existiert QuizID oder QuestionNr nicht, führt das zum Fehlercode 404, da das entsprechende Objekt nicht gefunden werden konnte. Der Nutzer kann bei dieser Anfrage entweder ErstellerIn des Quizzes sein, was den Nutzer zu dieser Operation berechtigen würde, oder nicht, was zu einer Response mit Fehlercode 403 führen würde. Der Questiontext, sowie der Answertext dürfen nicht leer sein. Es muss mindestens zwei Antworten geben. Wird eine dieser Bedingungen nicht erfüllt, wird der Fehlercode Bad Request (400) zurückgegeben. Sind alle Eingaben gültig wird mit Statuscode Ok (200) geantwortet.

Die folgenden Tabellen stellen dar wie die beschriebenen Äquivalenzklassen notiert werden, und anschließend zeigt die zweite Tabelle einen Ausschnitt der daraus entstehenden Testfälle, insgesamt ergeben sich hier acht Testfälle, von denen die Hälfte dargestellt wird.[33]

---

[32] Aequivalenzklassenanalyse [Kle09]

[33] Alle 42 Tests können im Anhang als PDF gefunden werden

Eingabe	gültig	ungültig
QuizID	A1) existiert	A2) Existiert nicht
QuestionNr	A3) existiert	A4) Existiert nicht
Nutzer	A5) Ist Ersteller	A6) Ist nicht Ersteller
Title	A7) Nicht leer	A8) Leer
Answers	A9) Alle Texte nicht leer A13) mind. 2	A10) Ein Text leer A14) weniger als 2
CorrectAnswer	A11) Mind. eine	A12) keine

Tabelle 1: Äquivalenzklassen zum Bearbeiten einer Frage eines Quizzes

Testnummer	1	2	3	4
Geprüfte Ä-Klassen	A1, A3, A5, A7, A9	A2	A4	A6
QuizID	1	0	1	1
QuestionNr	1	1	5	1
Nutzer	laupeter	laupeter	laupeter	jobernhard
Title	Was ist keine Sukkulente	Was ist keine Sukkulente	Was ist keine Sukkulente	Was ist keine Sukkulente?

Answers	[
	{
	Nr: 1
	Text: Baum
	correctAnswer: true
	},
	{
	Nr:2
	Text: Aloe Vera
	correctAnswer: false
	}
	]
		[
		{
		Nr: 1
		Text: Baum
		correctAnswer: true
		},
		{
		Nr:2
		Text: Aloe Vera
		correctAnswer: false
		}
		]
			[
			{
			Nr: 1
			Text: Baum
			correctAnswer: true
			},
			{
			Nr:2
			Text: Aloe Vera
			correctAnswer: false
			}
			]
				[
				{
				Nr: 1
				Text: Baum
				correctAnswer: true
				},
				{
				Nr:2
				Text: Aloe Vera
				correctAnswer: false
				}
				]

Ergebnis	Ok(200)	Not Found(404)	Not Found(404)	Forbidden(403)

Tabelle 2: Ausschnitt Definition Äquivalenzklassentests

Der erste beschrieben Testfall sieht mit REST Assured dann wie folgt aus. Der angegebene User ist „laupeter“ die verwendetet QuizID ist auf 1 gesetzt ebenso wie die QuestionNr, der Titel ist gesetzt und es gibt zwei Antworten, eine richtige und eine falsche. Somit wird erwartet, dass die API mit dem Statuscode 200 antwortet und signalisiert, dass die Operation erfolgreich war.

```
@Test
@TestTransaction
@TestSecurity(user="laupeter")
public void editQuestionOk(){
  Long quizId = 1L;
  int questionNr = 1;
  ArrayList<AnswerDTO> answers = new ArrayList<>();
  answers.add(new AnswerDTO("Baum", true));
  answers.add(new AnswerDTO("Aloe Vera", false));
  QuestionDTO q = new QuestionDTO(questionText, answers);
  given().contentType(ContentType.JSON)
  .body(q).put("/quiz-fest/api/quizzes/"+quizId+"/edit/"+questionNr)
  .then().statusCode(200);
}
```

Snippet 10: Beispielhafter Test mit REST Assured

3.4.2 Dokumentation mit OpenAPI (Bernhard)

Um die Annotationen von OpenAPI zu verwenden, wird eine Erweiterung der Application Klasse verwendet, dort werden Informationen über die gesamte QuizFest API konfiguriert. Mit den Security-Einstellungen kann dasselbe Bearer Token in allen Anfragen verwendet werden, um gegen die Anwendung zu authentifizieren.

Für kurze Beschreibungen der einzelnen Methoden wird die @Operation Annotation verwendet in der eine description gesetzt werden kann. An den einfachen Übergabeparametern werden Beispielwerte mit @Parameter und example gegeben.

Da komplexere Objekte jedoch nicht im example der @Parameter Annotation übergeben werden können, wird zusätzlich eine konfigurierte Postman Collection bereitgestellt.

4 Zusammenfassung und Fazit (Bernhard + Peter)

Zusammenfassend lässt sich sagen, dass die in der Anforderungsanalyse beschriebenen Muss-Kriterien und somit die benötigten Produktfunktionen umgesetzt sind. Die Wunschkriterien sind jedoch aus zeitlichen Gründen nicht umgesetzt. Der Fokus wurde darauf gelegt die Qualitätsmerkmale sicherzustellen.

Für die Korrektheit wurden umfangreiche Tests modelliert und geschrieben. Bei der Definition der API ist besonders berücksichtigt, dass die Struktur verständlich bleibt und durch Verwendung von bspw. http-Verben und Hyperlinks gut bedienbar ist. Die Sicherheit der Anwendung wird mit Keycloak und OpenID Connect hergestellt, was ein moderner und gut dokumentierter Ansatz ist, der sich in Kombination mit Quarkus verwenden lässt. Wobei OIDC ein wohldefiniertes Standard-Internetprotokoll ist. Für die Wartbarkeit und Änderbarkeit ist die Struktur der Softwarebausteine nach dem ECB entworfen worden und die SOLID Prinzipien wurden, wenn angemessen verwendet.

Als Fazit ergibt sich, dass die erarbeitete Anwendung qualitativ hochwertig verwirklicht wurde.
Zwischen den einzelnen Layern wurde möglichst viel abstrahiert und getrennt, wo es sinnvoll war. Mit DTOs in der Boundary konnte vermieden werden, dass die Entitäten von JSON-B abhängig sind, da sie nicht direkt in der Boundary verwendet werden. Die Anwendung wurde ausgiebig und methodisch getestet, sodass anzunehmen ist das die Funktionalität korrekt ist, auch wenn keine Tests vollständig garantieren können das eine Anwendung fehlerfrei ist.
Das Zusammenfassen von play und edit innerhalb des Quiz-Moduls ist kritisch und eine höhere separation of concerns wäre wünschenswert, jedoch sind im Grunde von beiden aufgaben Bereichen die gleichen Entitäten verwendet und benötigt, weshalb die Entscheidung getroffen wurde dies, als ein Modul zu behandeln.
Zu Keycloak ist zu sagen, das nicht das volle Potenzial ausgeschöpft wurde, da dieses Tool noch deutlich mehr zu bieten hat. Die Ressource Based Access Control, die dadurch breit gestellt wird, wurde nicht verwendet und es wurde nur die Nutzerelemente von Keycloak beansprucht. Jedoch kann dies bei einer Skalierung vorteilhaft sein, da die Infrastruktur dann bereits verfügbar ist und ohne Codeänderung ausgebaut werden kann.
Als Ausblick, es besteht die Möglichkeit zur Erweiterung, beispielsweise kann ein Leaderboard hinzugefügt werden oder mehr Quizattribute wie Zeitlimits zum Beantworten der Fragen, Beschreibungen zu Quizzes und Bildressourcen für Fragen. Ein Zeitlimit wäre im Server zwar schwierig umzusetzen, wenn das Beantworten von Fragen weiterhin zustandslos bleiben soll, aber wenn ein Zeitlimit angegeben wäre, könnte dieses im Client gehandhabt werden. Des Weiteren besteht die Möglichkeit, dass andere Nutzer speziell Quizze nach einem Ersteller filtern können. Im User Modul könnte erweitert werden, dass gespielte Spiele abgelegt werden, damit man sich mit vergangenen Ergebnissen messen kann.

5 Referenzen

Web-Seiten zuletzt am 06.08.2021 abgerufen.

[@Pal] Jeffrey Palermo, The Onion Architecture
https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[@Qua] Quarkus Guides
1 https://quarkus.io/guides/security-jwt
2 https://quarkus.io/guides/security-openid-connect
3 https://quarkus.io/guides/security
4 https://quarkus.io/guides/security-oauth2
5 https://quarkus.io/guides/validation
6 https://quarkus.io/guides/hibernate-orm-panache
7 https://quarkus.io/guides/openapi-swaggerui
8 https://quarkus.io/guides/getting-started-testing
9 https://quarkus.io/guides/getting-started#using-injection

[@mai] Silas Graffy, Hexagonale Architekturen erkärt
https://www.maibornwolff.de/blog/von-schichten-zu-ringen-hexagonale-architekturen-erklaert

[@SJU] San Jose University, Implementing Use Cases
http://www.cs.sjsu.edu/~pearce/modules/patterns/enterprise/ecb/ecb.htm

[@Wik] Wikipedia, Richardson Maturity Model
https://en.wikipedia.org/wiki/Richardson_Maturity_Model

[@Mar] Martin Fowler, steps towards the glory of REST
https://martinfowler.com/articles/richardsonMaturityModel.html

[@swg] Swagger, Basic Authentication
https://swagger.io/docs/specification/authentication/basic-authentication/

[@mcs] Microsoft, Grundlagen der HTTP-Authentifizierung
https://docs.microsoft.com/de-de/dotnet/framework/wcf/feature-details/understanding-http-authentication

[@phr] Philip Riecks, MicroProfile Config
https://rieckpil.de/whatis-eclipse-microprofile

[@jwt] JWT, Introduction to JSON Web Token
https://jwt.io/introduction

[@Aut] Auth0 Docs, Role-Based Access Control
1 https://auth0.com/docs/authorization/rbac
2 https://auth0.com/docs/tokens/access-tokens
3 https://auth0.com/docs/flows/authorization-code-flow

[@dns] DNSstuff, RBAC vs ABAC
https://www.dnsstuff.com/de/rbac-vs-abac-zugriffskontrolle

[@oid] OpenID Documentation
https://openid.net/connect

[@bnv] Jakata Bean Validation
https://beanvalidation.org

[@MSe] REST Assured Tutorial 8 – BDD Style in Rest Assured - Make Selenium Easy
http://makeseleniumeasy.com/2019/11/19/rest-assured-tutorial-8-bdd-style-in-rest-assured/

[@oau] OAuth 2.0 Bearer Token Usage
https://oauth.net/2/bearer-tokens/

[@IMB] IMB, Multi-Tennant
https://www.ibm.com/cloud/learn/multi-tenant

[@Kyc] Keycloak, About
https://www.keycloak.org/about.html

Sonstige Quellen:

[Kle09] S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

Sowie Foliensatz 02 bis 08 zur Veranstaltung Software-Architektur von Prof. Dr.-Ing. Rainer Roosmann

Anhang:

Klassenmodell: Moddelierung_QuizFest.pdf

Definition der Tests: Äquivalenzklassen_Tests_QuizFest.pdf

Eidesstattliche Erklärung

Hiermit erkläre ich/ erklären wir an Eides statt, dass ich / wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe / haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Osnabrück, 07.08.2021..... ....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)