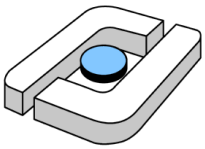

Hochschule Osnabrück

University of Applied Sciences

Fakultät
Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Digitales Sammelheft für Pokémon Karten

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

Autor:	Berkan Yildiz Johannes Belaschow
Matr.-Nr.:	932457 969082
E-Mail:	berkan.yildiz@hs-osnab-rueck.de johannes.belaschow@hs-osn-abrueck.de
Themensteller:	Prof. Dr. Rainer Roosmann

Abgabedatum: 17.02.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

CDI	Context and Dependency Injection for the Java EE Plattform
ECB	Entity-Controller-Boundary Pattern
Java EE	Java Enterprise Edition, in der Version 7
SWA	Software-Architektur
SFLB	Statefull Session Bean
SLSB	Stateless-Session Bean
SSO	Single Sign On
IAM	Identity and Access Management
JPA	Jakarta Persistence API
ORM	Objektrelationales Mapping
Bean	Klassen, die viele Objekte in einem einzigen Objekt kapseln
POJO	Plain Old Java Object

1 Einleitung

[jbelasch]

Die zu erledigende Hausarbeit wurde während der Implementation zu 100% in Pair Programmierung entwickelt. Verantworte im Projektbericht werden über dem Kapitel bzw. Unterkapitel bekanntgemacht.

Im Rahmen der Veranstaltung Software-Architektur bei Prof. Roosmann erlernten wir die wesentlichen Aspekte für die Erstellung von strukturierter Software. Die Prüfungsleistung besteht darin eigenständig ein Projekt auszuarbeiten und einen ausführlichen Bericht zu diesem zu formulieren.

In dem folgenden Kapitel werden die Vorstellung des Themas, Ziele der Ausarbeitung und Aufbau der Hausarbeit erläutert.

1.1 Vorstellung des Themas

Der Zweck dieses Projekts ist es, das digitale Sammeln von Pokémon-Karten durch eine Webanwendung mit der Rest-API zu unterstützen. Benutzer können Pokémon-Karten sammeln/kaufen, tauschen und ihre Sammlung anzeigen. Die Implementierung erfolgt in Java mit Quarkus.

1.2 Ziel der Ausarbeitung

Ziele der Ausarbeitung sind es, mit strukturiertem Vorgehen ein Stück Software zu entwickeln, welches robust, modular und einfach zu erweitern ist. Der Leser wird in dem Bericht ausführliche Einblicke in den Planungs-, sowie Entwicklungsprozess bekommen und sehen, wie praktische Methoden zur Entwicklung von Software angewendet werden.

1.3 Aufbau der Hausarbeit

Das erste Kapitel dient der allgemeinen Einführung in das Thema und soll dem Leser eine Übersicht verschaffen. Hier wurden bereits Rahmenbedingungen, Thema und Ziele angesprochen.

Im weiteren Kapitel werden theoretische Grundlagen aus der Veranstaltung angesprochen, die als essenziel für das Verständnis der Entwicklungsprozesse bzw. Der API erachtet. Hierzu zählen Themen wie: Grundlagen der API-Entwicklung, ECB-Pattern, lose Kopplung, hohe Kohäsion etc.

Im dritten Kapitel werden Anforderungen an die zu erstellende API formuliert. Dazu werden bekannte Techniken aus der Vorlesung, sowie der Veranstaltung objekt-orientierte Analyse und Design genutzt.

Das vierte Kapitel beschäftigt sich mit der Auswahl an Technologien, die genutzt werden neben der bereits festgelegten Verwendung von Quarkus in Java. Dazu zählen die Wahl des Frontendes, der Datenbank, Authentifizierungs-Services, sowie die Wahl von hilfreichen Tools, wie Swagger-UI, Postman.

Gegenstand des fünften Kapitels ist die konkrete Umsetzung und Implementation der festgelegten Anforderungen. Des Weiteren werden durchgeführte Test angesprochen, die zur Sicherung der Qualität der Software beitragen sollen.

Das letzte Kapitel dient dem Abschluss des Berichtes in dem auf Lernerfolge, mögliche Verbesserungen der entwickelten Software und ein Fazit eingegangen wird.

2 Darstellung der Grundlagen

[beryildi]

Das folgende Kapitel befasst sich mit der Darstellung der Grundlagen. In dieser wird die genutzte Software-Architektur erläutert und setzt sich ebenfalls mit Best Practices auseinander.

2.1 ECB-Pattern

Das ECB-Pattern ist ein Software-Architekturmuster für Anwendungen. Es trennt die Datenentitäten (Entity-Schicht) von den Steuerungslogiken (Control-Schicht) und dem Benutzerinterface (Boundary-Schicht).

Dieses Pattern hilft bei der Verbesserung der Übersichtlichkeit und Wartbarkeit der Anwendung, indem es sicherstellt, dass jede Schicht ihre eigene Verantwortung hat und voneinander isoliert sind. Im Rahmen des Pokecollect-Programms wurde das ECB-Modell um ein Gateway erweitert. Das Gateway ist die Schnittstelle zur Datenbank

Für die Verbindung von externen Systemen wurde das ACL-Pattern genutzt. Das Anti-Corruption-Layer (ACL) ist ein Architekturmuster, das verwendet wird, um den Integrationsaufwand zwischen unterschiedlichen Systemen zu reduzieren und die Integrität der Daten zu gewährleisten. Es fungiert als Übersetzungsschicht und verarbeitet die Daten so, dass sie für das integrierende System verwendbar sind, ohne dass dessen Datenmodelle direkt geändert werden müssen. [1]

2.2 SOLID

Der nächste Unterabschnitt behandelt die SOLID-Prinzipien von Robert C. Martin. Das SOLID-Modell ist ein Softwareentwicklungskonzept, das fünf Prinzipien beschreibt, die helfen sollen, organisierte, stabile und skalierbare Anwendungen zu entwickeln. [2]

2.3 SRP: Single-Responsibility-Prinzip

„Das Single-Responsibility-Prinzip besagt, dass **eine Klasse nur eine Verantwortlichkeit** haben soll. Änderungen an der Funktionalität sollen nur Auswirkungen auf wenige Klassen haben. Je mehr Code geändert werden muss, desto höher ist das Fehlerrisiko.“ [3] Das SRP unserer Anwendung wird durch verschiedene Managerklassen vorgegeben. Verwaltungs-, Benutzer- und Transaktionsfunktionen sind strikt voneinander getrennt.

2.4 OCP: Open-Closed-Prinzip

„Nach dem Open-Closed-Prinzip soll eine Klasse offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein. Das Verhalten einer Klasse darf erweitert, aber nicht verändert werden.“ Das OCP lässt sich durch Interfaces oder Vererbung realisieren. [3]

2.5 LSP: Liskov’sche Substitutions-Prinzip

„Das Liskovsche Substitutionsprinzip fordert, dass **abgeleitete Klassen immer anstelle ihrer Basisklasse einsetzbar** sein müssen. Subtypen müssen sich so verhalten wie ihr Basistyp.“ [3]

2.6 ISP: Interface-Segregation-Prinzip

„Das Interface-Segregation-Prinzip besagt, dass ein Client nicht von den Funktionen eines Servers abhängig sein darf, die er gar nicht benötigt. Ein Interface darf demnach **nur die Funktionen enthalten, die auch wirklich eng zusammengehören**.“ [3]

2.7 DIP: Dependency-Inversion-Prinzip

„Das Dependency-Inversion-Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollen. Dabei geht es nicht darum, die Abhängigkeiten einfach umzudrehen. Abhängigkeiten zwischen Klassen soll es nicht mehr geben; **es sollen nur noch Abhängigkeiten zu Interfaces bestehen** (beidseitig).“ [3]

3 Anforderungsanalyse

[jbelasch]

Es ist wichtig für die Anwendung klare Anforderungen zu definieren. Nach dem Einreichen des Projektvorschlages erhielten wir von Prof. Roosmann eine Liste von Muss- und Kann-Kriterien, sowie Rahmenanforderungen. Diese geben bereits eine grobe Liste an Anforderungen vor aus denen aufbauend die konkreten Anforderungen für “Pokecollect” erstellt werden können. Im Folgenden wird auf das Use-Case Diagramm eingegangen und dann die einzelnen definierten Anforderungen im Detail erklärt.

3.1 Use-Case Diagramm

Abbildung 1 Use-Case Diagramm

Die Nutzung der “Pokecollect” API sieht drei verschiedene Akteure vor. Den User selbst, der hauptsächlich die API nutzen wird. Einen Admin der in der Lage ist administrative Aufgaben auszuführen. Und schließlich das System selbst, welches das Gesamtkonstrukt in der Funktionalität erst ermöglicht.

3.2 Use-Cases User

Im ersten Abschnitt wird auf die Funktionalitäten des Users eingegangen.

3.2.1 Registrieren

Der User soll sich mit frei gewählten Zugangsdaten registrieren können. Bevor Der User die Anwendung bzw. Die API nutzen kann muss erst sich ein Konto anlegen können. Dies ist wichtig, weil diesem Konto im weiteren Ablauf der Anwendung der API, Informationen zugeschrieben werden müssen und es dabei um individuelle Informationen handelt, die für jeden User anders aussehen werden.

3.2.2 Authentifizieren

Entsprechend muss der User nach seiner Registrierung, die Möglichkeit haben mit seinen Zugangsdaten Zugang zu seinem Account zu bekommen. Diese geben ihm Zugriff auf seine persönliche Pokémon-Sticker Sammlung und alle anderen Funktionalitäten, auf die jetzt eingegangen wird.

3.2.3 Karten erwerben

Dem User soll die Möglichkeit zur Verfügung stehen Karten kostenlos zu erhalten oder erwerben zu können. Das Erhalten von kostenlosen Karten ist zeitlich limitiert.

3.2.4 Sammlung einsehen

Nach dem Erwerb der Pokémon-Karten sollen diese in der Sammlung des Users auftauchen und einzusehen sein. Duplikate von Pokémon, sollen dem User auch entsprechend angezeigt werden. Pokémon, die der User noch nicht in Besitzt hat, sollen im Sammel-Album entsprechend kenntlich gemacht werden.

3.2.5 Detaillierte Informationen einsehen

Nicht nur soll der User in seinem Pokémon-Karten-Album sehen können welche Pokémon er wie oft besitzt, sondern auch bei bereits gesammelten Karten weitere Informationen erhalten, wie ein Bild von dem Pokémon, den Namen und den Pokémon-Typen, denen er zugehört.

3.2.6 Pokémon Karten zum Tausch freigeben

Der User soll über seine Pokémon-Karten frei verfügen sollen. Dafür soll ein Featuren implementiert werden, welches ermöglichen soll, eine Karte gegen eine andere Karten zum Tausch freizugeben.

3.2.7 Pokémon Karten tauschen

Dieses Tauschangebot soll dann für alle anderen User der API sichtbar sein und Interessenten die Möglichkeit bieten diesen Tausch einzugehen, insofern die verlangte Karte im Besitzt des Interessenten ist.

3.2.8 Favoriten Pokémon setzen und entfernen

Auch soll der User die Möglichkeit haben seine Lieblings Pokémon als Favoriten zu markieren und in einer separaten Ansicht einsehen zu können. Pokémon-Karten, die bereits als Favoriten markiert wurden, sollen auch wieder von der Favoriten-Liste entfernt werden können.

3.3 Use-Cases System

Nachdem nun die Anforderungen für den User formuliert wurden, wird nun auf das System eingegangen. Die Anforderungen an das System stehen im engen Zusammenhang mit den Anforderungen des Users. Die entsprechenden Zusammenhänge werden im Folgenden erläutert.

3.3.1 Benutzer als Account anlegen

Dem User steht die Funktionalität zur Verfügung sich zu registrieren. Das System muss entsprechend diesen Registrierungs-Request entgegennehmen und verarbeiten. Das Konto für den User soll entsprechend angelegt werden.

3.3.2 Benutzer prüfen (authentifizieren)

Das System soll nach der erfolgreichen Registrierung des Users, bei jedem weiteren Zugang des Users, die Zugangsdaten des Users entgegennehmen und diese prüfen (Authentifizierung). Sollten die Zugangsdaten korrekt sein, wird dem User Zugang zu seinem Sammelalbum und allen zugehörigen Features gewährt.

3.3.3 Aller User einsehen

Auch dem steht dem User der Erwerb von Karten offen. Das System ist verantwortlich dafür, dass alle Kriterien für den Kauf geprüft werden und die erfolgreich erworbenen Karten entsprechen dem Account des Users zugeschrieben werden.

3.3.4 Zugriff auf externe API

Alle wichtigen Informationen zu Pokémon wie Pokémon-IDs, Bilder-URLs, Pokémon-Typen und Pokémon-Namen werden aus einer externen API bezogen. Unser System soll im besten Fall die Gesamtheit dieser Daten einmalig von der externen API abfragen und diese dann lokal in einer Datenbank persistieren. Diese Informationen werden dem User für Pokémon, die bereits gesammelt wurden zur Verfügung gestellt.

3.3.5 Persistieren aller wichtigen Daten

Des Weiteren wird das System dafür zuständig sein wichtiges Daten zu persistieren. Dazu zählen: Pokémon im Besitzt, Trainer (User) und offene Angebote von den Usern.

3.4 Use-Cases Admin

Zu guter Letzt kommen die Anforderungskriterien für die Funktionen des Admins.

3.4.1 Alle User bzw. spezifische User einsehen

Der Admin soll die Möglichkeit bekommen, sich die Liste aller User ausgeben zu können. Auch soll die Such nach Usern mit der User-ID möglich sein.

3.4.2 Pokémon vom User entfernen und hinzufügen

Eine weitere administrative Funktionalität, die der Admin zur Verfügung gestellt bekommt ist das Hinzufügen und Entfernen von Karten von spezifischen Usern.

3.5 Nicht funktionale Anforderungen

[beryildi]

In diesem Abschnitt werden ergänzende nicht-funktionale Anforderungen erläutert.

3.5.1 Ergänzende nicht-funktionale Anforderungen:

- Korrekte Anwendung der Prinzipien, Konzepte, Vorgehensweisen, Muster, Best-Practices und Werkzeuge im SWA-Modul.

- Strukturierung kann unter Verwendung taktischer Muster des Domain-Driven-Designs, SOLID-Muster, Onion-/Hexagonal-Architektur der Clean-Architecture und/oder Entity-Controller-Boundary-(Gateway) Muster erfolgen.
- Praktische Lösung soll einfach kompilierbar, analysierbar, testbar, startbar und anwendbar sein.
- REST-API soll optimale Entwickler-Erfahrung bieten. Bei Bereitstellung eines Web-Frontend soll optimale Benutzer-Erfahrung angestrebt werden.
- Web-Frontend sind zu realisieren, als Client-Side Rendering (CSR) Lösung für das Modul „Web-Technologien“, wobei die Backend-Entwicklungen den behandelten Prinzipien, Konzepten, Vorgehensweisen und Werkzeuge des SWA-Moduls folgt.
- REST-APIs sind mit JAX-RS, JSON-B und ggf. JSON-P zu realisieren, wobei zumindest der Level 2, besser noch Level 3, des Richardson Maturity Models erreicht werden soll.
- REST-APIs folgen dem Backend-for-Frontend Pattern und unterstützen die einfache Kommunikation des Frontend mit dem Backend.
- IT-Sicherheit unterstützen, indem verschiedene operative Maßnahmen umgesetzt werden.
- Die Verfügbarkeit / Resilienz der REST-API soll durch Anwendung der Eclipse Microprofile „Fault Tolerance“ verbessert werden
- Die Einhaltung von Business-Regeln und die Konsistenz des Anwendungszustands ist u.a. über die Bean Validation API sicherzustellen
- Die Korrektheit der Funktionalität soll durch Unit- und Integrationstests nachgewiesen werden. Integrationstests sollen über RestAssured umgesetzt werden.
- Zur Spezifikation / Dokumentation der REST-API soll die OpenAPI eingesetzt werden.
- Zur Reduzierung der Abhängigkeiten zwischen Software-Bausteinen sollen geeignete Ansätze gewählt werden.
- Zur Anbindung externer Systeme soll das „Anti-Corruption Layer“-Pattern umgesetzt werden.

- Die Persistierung erfolgt in einer relationalen Datenbank, unter Verwendung der JPA- und JTA-Spezifikationen, die proprietäre Hibernate Panache Lösung kann eingesetzt werden.

4 Technologieauswahl

[beryildi]

Im Fokus des vierten Kapitels steht die Auswahl von genutzten Technologien.

4.1 JetBrains Code With me

Die Pokecollect Applikation wurde in 100% Pair-Programming implementiert. Als Hilfsmittel für die Remote-Arbeit wurde das JetBrains Plugin Code With Me verwendet. Code With Me ist ein kollaborativer Codierungs- und Pair-Programming-Service. Es ermöglicht, andere in die IDE-Projekt einzuladen und gemeinsam in Echtzeit daran zu arbeiten. [4]

4.2 Quarkus

Java Quarkus ist ein Framework zur Entwicklung von Microservices und Cloud-nativen Anwendungen mit Java, mit besonderem Fokus auf REST-basierte Anwendungen. Es verwendet Technologien wie GraalVM und Kubernetes, um eine schnelle, ressourceneffiziente und einfach skalierbare REST-API bereitzustellen. Quarkus vereinfacht die Entwicklung von REST-basierten Anwendungen durch seine integrierten Funktionen und konzentriert sich auf Cloud-native Entwicklung. [5]

4.3 REST-Client externe API

Der Quarkus Rest-Client ist eine Bibliothek für die Entwicklung von REST-basierten Clients in Java. Er ermöglicht es Entwicklern, einfach und effizient REST-APIs aufzurufen.

4.4 Keycloak

Keycloak ist eine Open-Source-Lösung für SSO und IAM. Es ermöglicht ein zentralisiertes System bereitzustellen, um Benutzeridentitäten und den Zugriff auf Anwendungen und Dienste zu verwalten. Keycloak bietet eine Verwaltungskonsole und unterstützt die Integration mit vielen verschiedenen Anwendungen und Protokollen, einschließlich SAML, OAuth und OpenID Connect. [6]

4.5 Jakarta Persistence API

Das Zuordnen von Java-Objekten zu Datenbanktabellen ist eine Technik, die als ORM bezeichnet wird. Die JPA ist ein ORM-Format. Durch JPA kann ein Entwickler Daten aus einer relationalen Datenbank Java-Objekten zuordnen, speichern, manipulieren

und abrufen und umgekehrt. JPA kann in Java EE- und Java SE-Anwendungen verwendet werden. [7]

4.6 Hibernate Validator

Hibernate Validator ist eine Implementierung der Java Bean Validation API, die die Möglichkeit bietet, die Datenintegrität in Java-Anwendungen zu überprüfen. Es ermöglicht Entwicklern, Validierungsregeln für Java-Beans festzulegen und sie automatisch ausführen zu lassen, wenn Dateneingaben validiert werden. [8]

4.7 JSON-B

JSON-B ist eine Standardbindungsschicht zum Konvertieren von Java-Objekten in und aus JSON-Nachrichten. Es definiert einen Standard-Mapping-Algorithmus zum Konvertieren vorhandener Java-Klassen in JSON und ermöglicht Entwicklern, den Mapping-Prozess mithilfe von Java-Annotationen anzupassen. [9]

4.8 Rest-Assured

REST Assured ist eine Java-Bibliothek, mit der Tests für REST-APIs mithilfe einer flexiblen domänenspezifischen Sprache (DSL) geschrieben werden können, die sogar nicht Domain Experten verstehen würden.

5 Proprietär: Hibernate Panache

[beryildi]

Die Verwendung von Panache Hibernate hat einige Auswirkungen auf die Java-Softwarearchitektur im Allgemeinen.

Panache Hibernate bietet eine höhere Abstraktionsebene. Dadurch können Entwickler beispielsweise Datenbankoperationen durchführen, ohne die Feinheiten des Hibernate-Frameworks zu kennen. Ebenso bietet sich eine einfache und benutzerfreundliche API, die Entwicklern hilft, Datenbankoperationen zu implementieren.

Es sollte jedoch beachtet werden, dass die Verwendung proprietärer Lösungen wie Hibernate Panache in der Flexibilität und Anpassung der Lösung eingeschränkt sein kann.

Beispiele für Einschränkungen, die mit der Verwendung proprietärer Lösungen wie Hibernate Panache verbunden sein können, sind:

- Abhängigkeit: Da Hibernate Panache eine proprietäre Lösung ist, sind Entwickler von der Verfügbarkeit und dem Support des Anbieters abhängig.
- Flexibilität: Es können Einschränkungen bei der Anpassbarkeit und Flexibilität geben. Entwickler müssen sich möglicherweise an die Vorgaben des Anbieters halten.
- Integrität: Proprietäre Lösungen können möglicherweise Schwachstellen aufweisen, da sie nicht öffentlich überprüft werden können. [10]

6 Entwicklung

In diesem Kapitel wird der Entwicklungsprozess von „PokeCollect“ erklärt. Dabei wird auf die Grundstruktur und konkreten Code eingegangen. Es wurde sich beim Aufbau der Software stark an dem ECB-Pattern orientiert.

6.1 ECB-Pattern

[jbelasch]

Das ECB-Pattern schreibt die Grundstruktur der Anwendung vor. Gegliedert wird der Aufbau der Software in einem Schichten-System: Boundary, Control, Entity und Gateway. Auf die einzelnen Schichten und deren Beschaffenheit wird in den folgenden Kapiteln eingegangen.

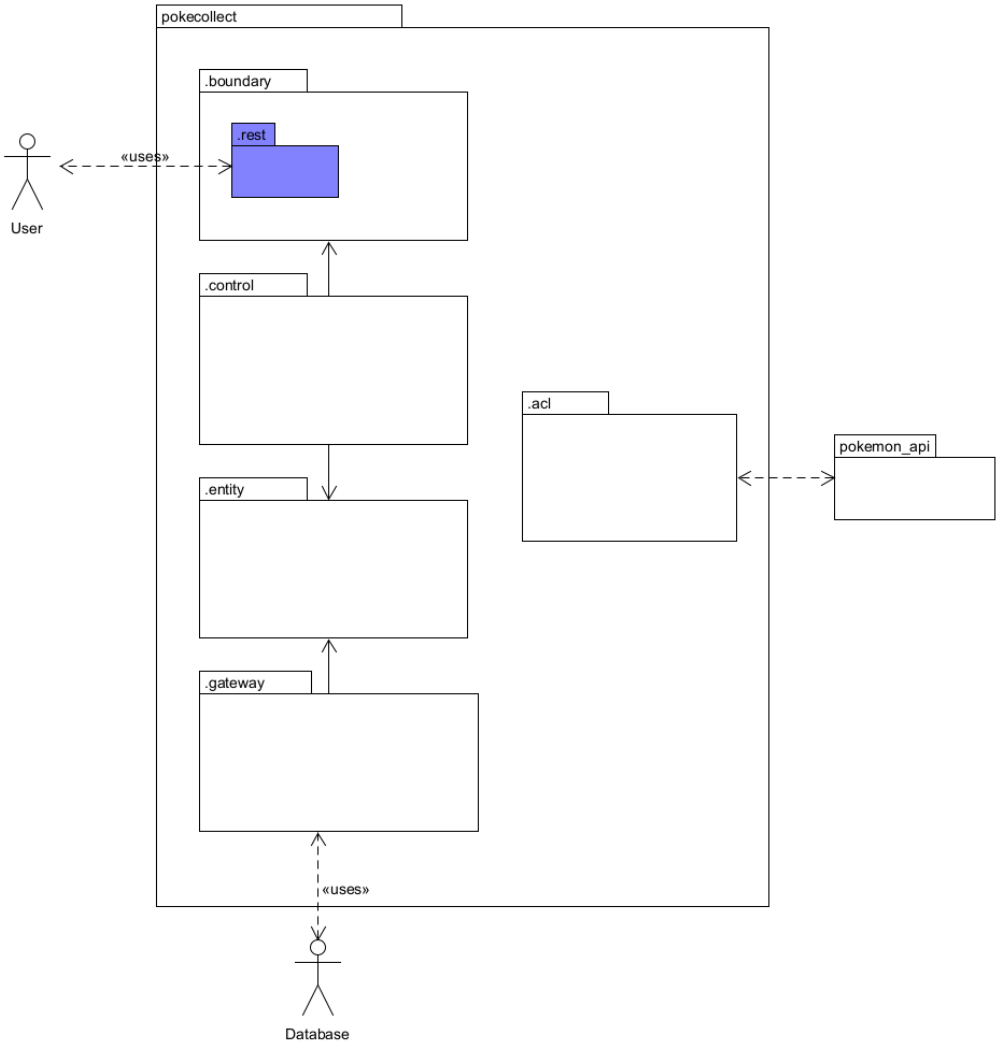

Abbildung 2 Paketdiagramm: Gesamtsystem

6.1.1 Boundary

[jbelasch]

Das folgende Unterkapitel befasst sich mit der Präsentationsschicht und deren Implementation.

Die Boundary stellt die Schnittstelle dar mit der von Außerhalb interagiert werden kann. Alle eingehen Request werden hier entgegengenommen und verarbeitet.

Die verschiedenen Zuständigkeiten der API wurden aufgeteilt auf vier Ressourcen: Admin-Ressource, User-Ressource, Pokémon-Ressource und eine Trade-Ressource. Diese vier Ressourcen wurden entsprechend auf vier Packages aufgeteilt.

Zu jeder Ressource gibt eine passendes Manager Interface, welches in die zugehörige Ressource per CDI mit der Annotation @Inject, injected wird und so eine Kommunikation mit tieferliegenden Schichten ermöglicht. Dies ist möglich, weil die Interfaces von den Manager Klassen in der Control-Schicht implementiert werden.

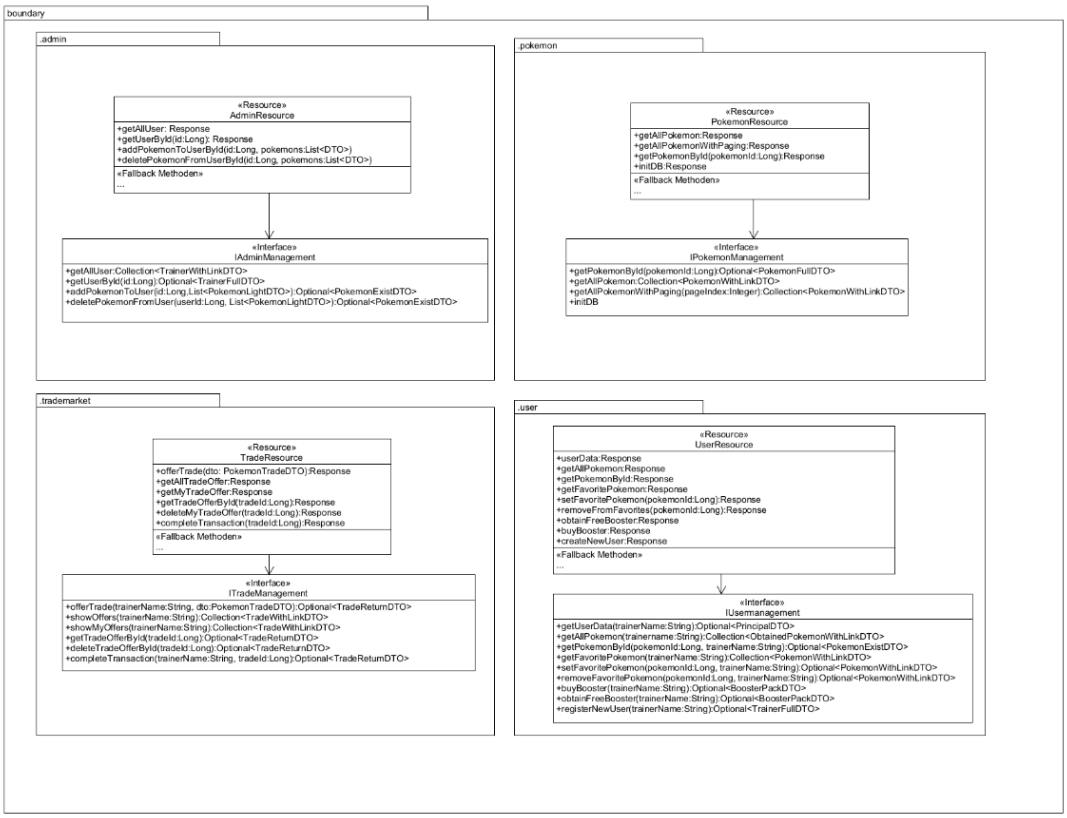

Abbildung 3 ECB-Boundary

Nachdem der Grundaufbau der Boundary erläutert wurde, gehen wir nun auf die einzelnen Zuständigkeiten und Funktionalitäten der Ressourcen ein.

Die erste Ressource ist die des Admins. Dieser verfügt über 4 Funktionen.

Tabelle 1 Admin-Resource

http-Methode	**Beschreibung**	**Eingabe**	**Statuscode**	**Ausgabe**
{application_path}/admins/user				
GET	Gibt eine Auflistung aller vorhandenen Trainer zurück		200	Collection aus TrainerWithLinkDTO im JSON-Format
			404	Fehlermeldung
{application_path}/admins/user/{userId}				
GET	Gibt den Trainer mit der angegebenen ID zurück.	Long Wert der größer gleich 1 ist	200	TrainerFullDTO im JSON-Format
{application_path}/admins/{id}				
POST	Erstellt ein neues Pokémon mit den bereitgestellten Daten und fügt es einem Trainer hinzu.	Long Wert der größer gleich 1 ist, Liste aus PokemonLightDTO im JSON-Format	200	PokemonExistDTO im JSON-Format
			404	Fehlermeldung
{application_path}/admins/{id}				
DELETE	Löschen Sie Pokemon mit den bereitgestellten Daten und entfernen Sie es von einem Trainer.	Long Wert der größer gleich 1 ist, Liste aus PokemonLightDTO im JSON-Format	200	PokemonExistDTO Im JSON-Format
			404	Fehlermeldung

Die ersten zwei Methoden ermöglichen es eine Liste von allen Usern oder einen spezifischen User mit einer ID zu bekommen. Der Admin soll dadurch immer einen Überblick über alle existierenden User halten können.

Mit den letzten zwei Methoden bekommt der Admin die Möglichkeit an die Hand von einem User Karten zu entfernen oder hinzufügen zu können. Dafür benötigt der Admin die ID des Users und die Liste an Pokémon, die es zu verändert gibt.

Die zweite Ressource ist die Pokémon-Ressource. Diese stellt dem User alle wichtigen Informationen zu den verfügbaren Pokémon bereit, wie Name, Typ und Bilder-URL. Sie gibt jedoch keine Auskunft darüber, ob man das Pokémon bereit in seinem Sammelalbum besitzt oder nicht. Für diese Information wird die User-Ressource zuständig sein, auf die gleich eingegangen wird. Folgende Methoden lassen sich in der Pokémon-Ressource finden:

Tabelle 2 Pokemon-Resource

http-Methode	**Beschreibung**	**Eingabe**	**Statuscode**	**Ausgabe**
{application_path}/pokemon				
GET	Gibt eine Sammlung aller vorhandenen Pokémon zurück.		200	Collection aus TrainerWithLinkDTO im JSON-Format
{application_path}/pokemon/page/{pageIndex}				
GET	Gibt eine Sammlung aller vorhandenen Pokémon mit Paging zurück.	Integer Wert der größer gleich 0 ist	200	Collection aus TrainerWithLinkDTO im JSON-Format
{application_path}/pokemon/{pokemonId}				
GET	Gibt das Pokémon mit der angegebenen ID zurück.	Long Wert der größer gleich 1 ist	200	PokemonFullDTO im JSON-Format
			404	Fehlermeldung

Jeder User ist in der Lage eine Liste von allen Pokémon abzurufen. Diese Liste hält alle wichtigen Informationen rund um die Pokémon, die verfügbar sind (Name, Typ, Bild-URL). Des Weiteren gibt es Möglichkeit die Pokémon mit Paging abzurufen. Sprich das nur 30 Pokémon auf einmal abgerufen werden. Pokémon 1-30 lassen sich auf der Page 0 finden. Pokémon 31-60 auf Page 1 usw.

Zuletzt gibt es auch die Möglichkeit Daten zu einzelnen Pokémon zu bekommen. Dafür benötigt man die zugehörige Pokémon ID des Pokémons.

Als nächstes haben wir die User-Ressource. Diese ist zuständig für eine Vielzahl an Funktionalität. Diese Ressource nimm Request entgegen, die für folgende Aufgaben zuständig sind:

Tabelle 3 User-Resource

http-Methode	Beschreibung	Eingabe	Statuscode	Ausgabe
{application_path}/users/whoami				
GET	Gibt den aktuell angemeldeten Benutzer zurück		200	PrincipalDTO im JSON-Format
{application_path}/users				
GET	Gibt alle vom Trainer erhaltenen Pokémon zurück		200	Collection aus ObtainedPokemonWithLinkDTO im JSON-Format
			404	Fehlermeldung
{application_path}/users /{pokemonId}				
GET	Geben Sie das erhaltene Pokémon mit der angegebenen ID zurück.	Long Wert der größer gleich 1 und kleiner gleich 151 ist	200	PokemonExistDTO im JSON-Format
			404	Fehlermeldung
{application_path}/users /favorite				
GET	Gibt eine Sammlung aller Pokémon zurück, die als Favorit festgelegt wurden.		200	Collection aus PokemonWithLinkDTO im JSON-Format
{application_path}/users /favorite/{pokemonId}				
PUT	Aktualisiert den Favoritenstatus des Pokémon mit der angegebenen ID.	Long Wert der größer gleich 1 und kleiner gleich 151 ist	200	PokemonWithLinkDTO im JSON-Format
			404	Fehlermeldung
{application_path}/users /favorite/{pokemonId}				
DELETE	Entfernt den Favoritenstatus des Pokémon mit der angegebenen ID.	Long Wert der größer gleich 1 und kleiner gleich 151 ist	200	PokemonWithLinkDTO im JSON-Format
			404	Fehlermeldung
{application_path}/users /freeBooster				
GET	Öffnen Sie ein kostenloses Booster-Pack und fügen Sie		200	BoosterPackDTO im JSON-Format
			204	Kein Inhalt

	das neue Pokémon der erhaltenen Pokémon-Liste hinzu.			

Zuletzt haben wir die Trade-Ressource. Diese stellt die Funktionalität rund um die Tausch-Börse zur Verfügung. Unser User soll Tausch-Angebote erstellen, einsehen und annehmen können.

Tabelle 4 Trade-Resource

http-Methode	**Beschreibung**	**Eingabe**	**Statuscode**	**Ausgabe**
{application_path}/trade				
POST	Erstellt ein neues Handelsangebot mit den bereitgestellten Daten.	PokemonTradeDTO im JSON-Format	200	Collection aus TrainerWithLinkDTO im JSON-Format
			404	Fehlermeldung
{application_path}/trade/allOffers				
GET	Gibt eine Sammlung aller bestehenden Handelsangebote zurück, mit Ausnahme der selbst erstellten.		200	Collection aus TradeWithLinkDTO
{application_path}/trade/allOffers				
GET	Gibt eine Sammlung von selbst erstellten Handelsangeboten zurück.		200	Collection aus TradeWithLinkDTO
{application_path}/trade/offer/{offerId}				
GET	Gibt ein Handelsangebot mit der angegebenen ID zurück.	Long Wert der größer gleich 1 ist	200	TradeReturnDTO im JSON-Format
			404	Fehlermeldung
{application_path}/trade/{tradeId}				
DELETE	Löschen Sie ein Handelsangebot mit der angegebenen ID.	Long Wert der größer gleich 1 ist	200	TradeReturnDTO im JSON-Format
			404	Fehlermeldung
{application_path}/trade/{tradeId}				

GET	Schließen Sie die Handels-angebotstransaktion mit der angegebenen ID ab.	Long Wert der größer gleich 1 ist	200	TradeReturnDTO im JSON-Format
			404	Fehlermeldung

Im Folgenden wird auf ein Code Beispiel aus der Boundry eingegangen. Im genauen wird hier auf die Annotationen und die Instanzvariablen der Ressource eingegangen.

```
@Inject
IUserManagement userManagement;

@Inject
SecurityIdentity securityIdentity;

@Inject
ResourceUriBuilder uriBuilder;
```

Abbildung 4 Ressource Instanzvariablen

Die IUserManagement wird per Dependency Inversion in die Ressource injected und dient dazu die Anfrage weiter in den Control-Layer zu reichen.

Die SecurityIdentity wird von Keycloak zur Verfügung gestellt. In unserem Use-Case nutzen wir SecurityIdentity, um den Namen des Users zu bekommen, der den Request an die API geschickt hat. Im Authentication-Header des Requests muss dafür der Bearer-Token des Users mit angehangen sein. Mit diesem wird der Request-Absender mit Hilfe von Keycloak identifiziert und alle relevanten Informationen per SecurityIdentity verfügbar gemacht. Beispielsweise kann man den Namen des Request-Absenders bekommen:

```
String trainerName = securityIdentity.getPrincipal().getName();
```

Abbildung 5 Name per SecurityIdentity

Mit Hilfe des ResourceUriBuilders werden Links generiert. Beispielsweise zu Detail Ansichten von einzelnen Pokemon:

Abbildung 6 getAllPokemon Response - URL Links

6.1.2 Control

[beryildi]

Das folgende Unterkapitel befasst sich mit der Steuerungsschicht und deren Implementation.

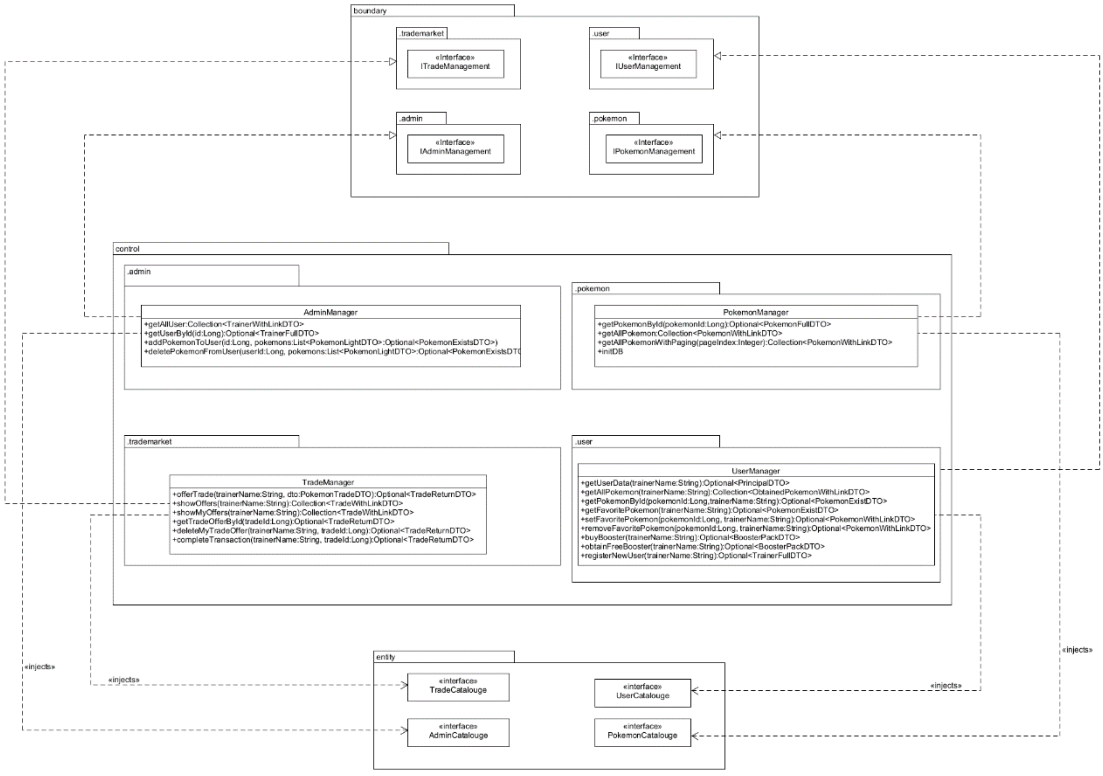

Abbildung 7 ECB-Control

Die Steuerungsschicht fungiert als Schnittstelle für die Kommunikation zwischen der Boundary- und der Gateway-Schicht. Wie in Abbildung 3 dargestellt, wurde die Präsentationslogik in vier Pakete aufgeteilt. Auch hier wird dieses Muster verfolgt.

Die Steuerungsschicht ist zustandsfrei und wird im Klassenkopf mit dem @Request-Scoped Annotation versehen. Um die Abhängigkeiten nach dem SOLID- Prinzip zu invertieren, werden die Schnittstellen IAdminManagement, IUserManagement, ITrade-Management und IPokemonManagement von den Kontrollklasse implementiert. Diese Schnittstellen werden schließlich in der Boundary in den entsprechenden Klassen durch CDI mittels @Inject injiziert. Die Manager-Klassen injizieren ebenfalls die Admin-, User-, Trade- und Pokemon-Katalog-Schnittstellen, die dazu beitragen sollen, die Abhängigkeiten zwischen der Entität-Schicht umzukehren.

Abbildung 8 AdminManager

6.1.3 Entity

[jbelasch]

Das folgende Unterkapitel befasst sich mit der Entitätsschicht und deren Implementation.

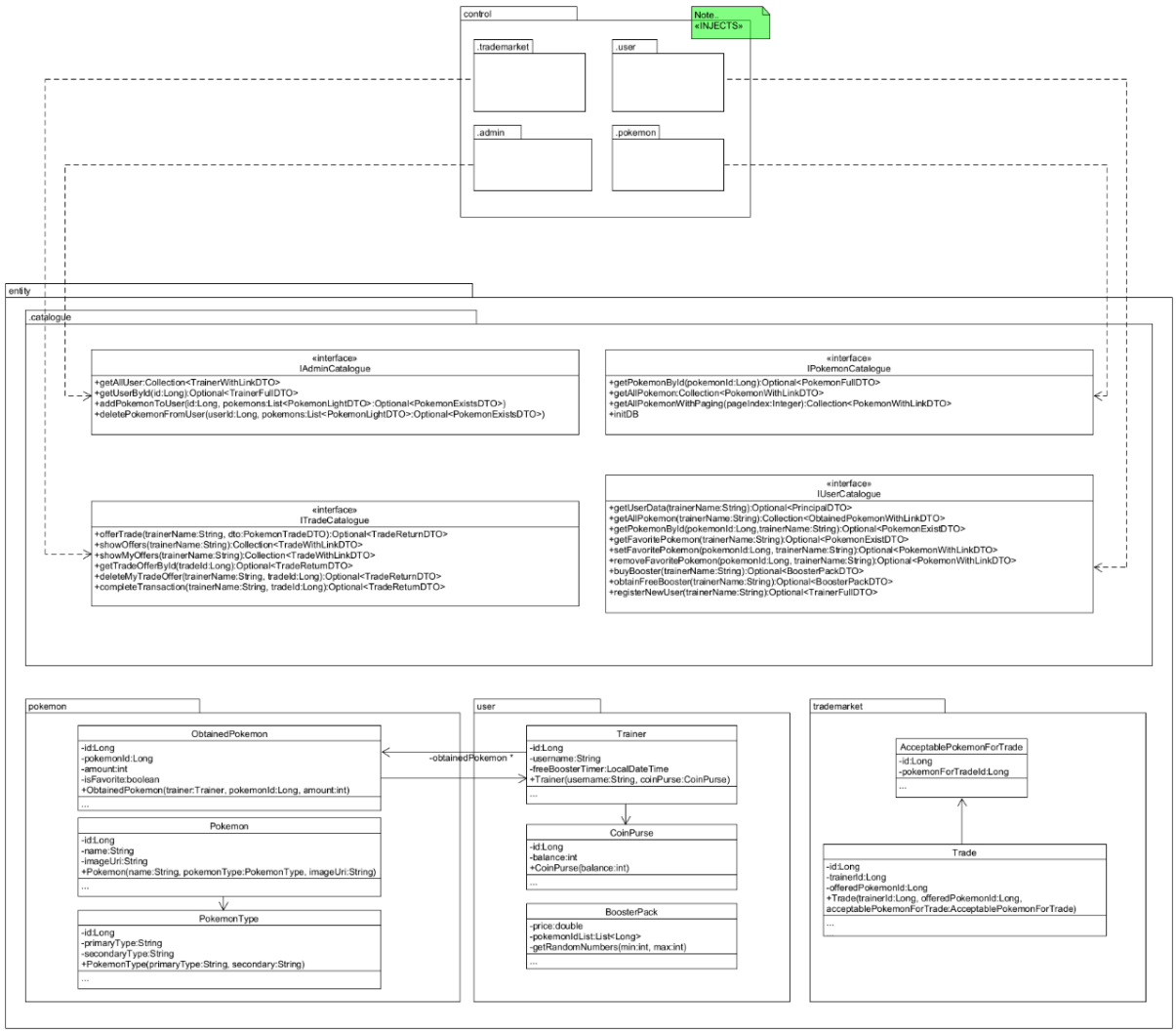

Abbildung 9 ECB-Entity

Das Entity-Package besteht aus zwei verschiedenen Arten von Komponenten.

Einmal, wie der Name es schon andeutet, werden hier alle relevanten Datentypen (Entitäten) für unser Programm gehalten. Alle diese Datentypen werden in der einen oder anderen Form in der Panache Repository persistiert.

Der andere Teil des Entity-Packages sind die „Catalogue“-Interfaces. Diese werden von den Repository-Klassen im Gateway-Layer implementiert und dann durch eine Dependency Inversion in die Manager-Klassen im Control-Layer injected.

Im weiteren Verlauf des Kapitels gehen wir auf konkrete Beispiele zu den Entitäten und den „Catalogues“ ein.

Zunächst kommen die wichtigsten Entitäten der Anwendung. Diese wurden gegliedert in drei Packages.

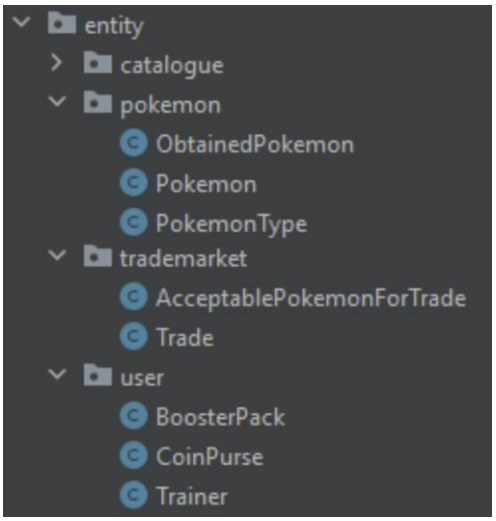

Abbildung 10 Entity Layer - Entitäten

Im User-Package haben wir die Klassen Trainer, BoosterPack und CoinPurse. Der Trainer stellt den User in der Anwendung dar. Dieser hat eine ID, Namen, CoinPurse (Währung), eine Liste von ObtainedPokemon, auf die wir noch zu sprechen kommen, und eine LocalDateTime. Letztere dient dazu es dem User zu ermöglichen in regelmäßigen Abstünden, kostenlose Kartenpackungen zu bekommen. In LocalDateTime wird dann die genaue Uhrzeit gespeichert, wann die letzte Kartenpackung geöffnet wurde. Die CoinPurse ist lediglich ein komplexer Container, der eine Menge an der „In-Game“-Währung halten soll. Die BoosterPack-Klasse stellt die Kartenpackungen dar, die vom User gekauft werden können. Dahinter liegen ein Preis für die Kartenpackung und eine ArrayList aus 3 Zahlen, die beim Kauf einer Kartenpackung zufällig generiert werden. Diese Zahlen bestimmen, dann die Pokémon-Karten, die der User aus seiner Kartenpackung bekommt.

Im Pokemon-Package finden sich alle wichtigen Entitäten rund um Pokémon. Die Pokemon-Klasse hält alle relevanten Informationen aus der externen API. Diese sind die Pokémon-ID, Name, Bilder-URL und der Pokémon-Typ. Beim Pokemon-Typen handelt es sich um einen komplexen Container. Dieser ist nur dazu da den Primär-Typen und Sekundär-Typen zu halten, falls einer vorhanden ist. Bei beiden Attributen handelt es sich um Strings. Die ObtainedPokemon halten alle wichtigen Informationen zu den Pokemon, die ein User besitzt. Diese haben ebenfalls eine Pokémon-ID. Des Weiteren wird der Trainer (User) angegeben zu dem das ObtainedPokemon gehoert, die

Amount, sprich wie oft der User Trainer diese Karte schon besitzt und zuletzt isFavorite, was darüber aussagt, ob es sich bei dieser Pokémon-Karte, um eine Favoriten-Karte handelt.

Zuletzt gib es noch das Trademarket-Package. Hier findet sich die Trade-Klasse. Jeder User wird die Möglichkeit haben Pokémon-Karten tauschen zu können. Dafuer kann er sich eine Karte aussuchen und einen Trade erstellen. Dieser Trade besteht aus einer eindeutigen ID, der Trainer-ID, die den Trade erstellt hat, der offeredID, welches Pokemon man zu tauschen besitzt und AccaptablePokemonForTrade, ein komplexer Container, der die Pokémon-Karte hält, welches man gerne haben möchte. Wir haben uns an dieser Stelle für einen komplexen Container entschieden, weil wir uns die Möglichkeit offenlassen wollten, für eine Pokémon-Karte, die man anbietet, eine Vielzahl an Pokémon-Karten zu akzeptieren. Bei unserer aktuellen Implementation haben wir lediglich einen 1 zu 1-Tausch.

Beispielhaft wird nun Code demonstriert.

```
beryildi
@Entity
public class Pokemon extends PanacheEntityBase {

    @Id
    @SequenceGenerator(
            name = "PokemonSequence",
            sequenceName = "Pokemon_id_seq",
            allocationSize = 1,
            initialValue = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PokemonSequence")
    @Column(name = "id", nullable = false)
    private Long id;
    private String name;

    @OneToOne(cascade = CascadeType.ALL)
    private PokemonType pokemonType;
    private String imageUri;
```

Abbildung 11 Pokémon-Klasse

Über der Klasse findet sich die *@Entity* Annotation. Diese markiert die Klasse als Entität, die in unserer Datenbank persistiert wird. Des Weiteren erweitert die Klasse Pokemon von PanacheEntityBase. Dies ermöglicht es später in dem PokemonRepository (Gateway-Layer) die Klasse Pokemon, als PanacheRepository<Pokemon> zu implementieren. Dazu gleich mehr im Gateway Kapitel.

Weiterhin sieht man die *@Id* Annotation. Diese erzeugt beim Erstellen eines Pokemon Objektes eine Primay Key. Die Annotation *@SequenceGenerator* definiert, den Namen des Sequencers und mit welchem Wert die erste ID anfangen soll. In unserem Fall bei 1. Die Zahlensequenz wird jetzt nur inkrementiert, wenn ein neues Objekt vom Typ Pokemon erzeugt wird.

Die letzte wichtige Annotation mit der hier gearbeitet wird ist *@OneToOne.* Diese sagt aus, dass die Instanz vom Typ Pokemon, im Kontext von relationalen Datenbanken, in einer 1 zu 1 Beziehung steht zu einem PokemonType. *(cascade = CascadeType.ALL)* besagt nur, dass wenn ein Pokemon in unserer PanacheRepository persistiert wird, auch der PokemonType persistiert werden soll, ohne dass eine zusätzliche Aufforderung dazu notwendig ist.

Zu guter Letzt kommen die „Catalogues“.

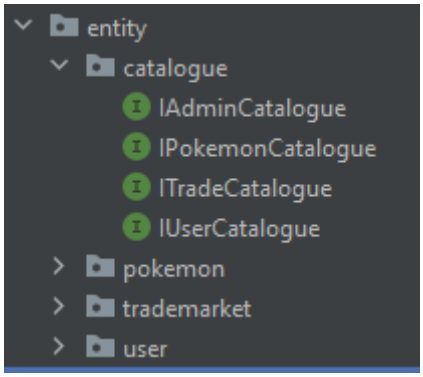

Abbildung 12 Entity Layer – Catalogue

Bereitgestellt wurden vier verschiedene “Catalogues” für vier verschiedene Zuständigkeitsbereiche aufgeteilt. Dabei handelt es sich um Interfaces, die von den Repositories implementiert werden. Diese werden dann per Dependency Inversion in den Manager-Klassen im Control-Layer injected. Folgende “Catalogues“ liegen vor:

- IAdminCatalogue
- IPokemonCatalogue
- ITradeCatalogue
- IUserCatalogue

Für jedes Repository werden hier die Methoden vordefiniert. Beispielhaft betrachten wir den IUserCatalogue:

```java
public interface IUserCatalogue {

    Optional<PrincipalDTO> getUserData(String trainerName);
    Collection<ObtainedPokemonWithLinkDTO> getAllPokemon(String trainerName);
    Optional<PokemonExistsDTO> getPokemonById(Long pokemonId, String trainerName);
    Collection<PokemonWithLinkDTO> getFavoritePokemon(String trainerName);
    Optional<PokemonWithLinkDTO> setFavoritePokemon(Long pokemonId, String trainerName);
    Optional<PokemonWithLinkDTO> removeFavoritePokemon(Long pokemonId, String trainerName);
    Optional<BoosterPackDTO> buyBooster(String trainerName);
    Optional<BoosterPackDTO> obtainFreeBooster(String trainerName);
    Optional<TrainerFullDTO> registerNewUser(String trainerName);


}
```

6.1.4 Gateway

[beryildi]

Das folgende Unterkapitel befasst sich mit der Datenbankschicht und deren Implementation.

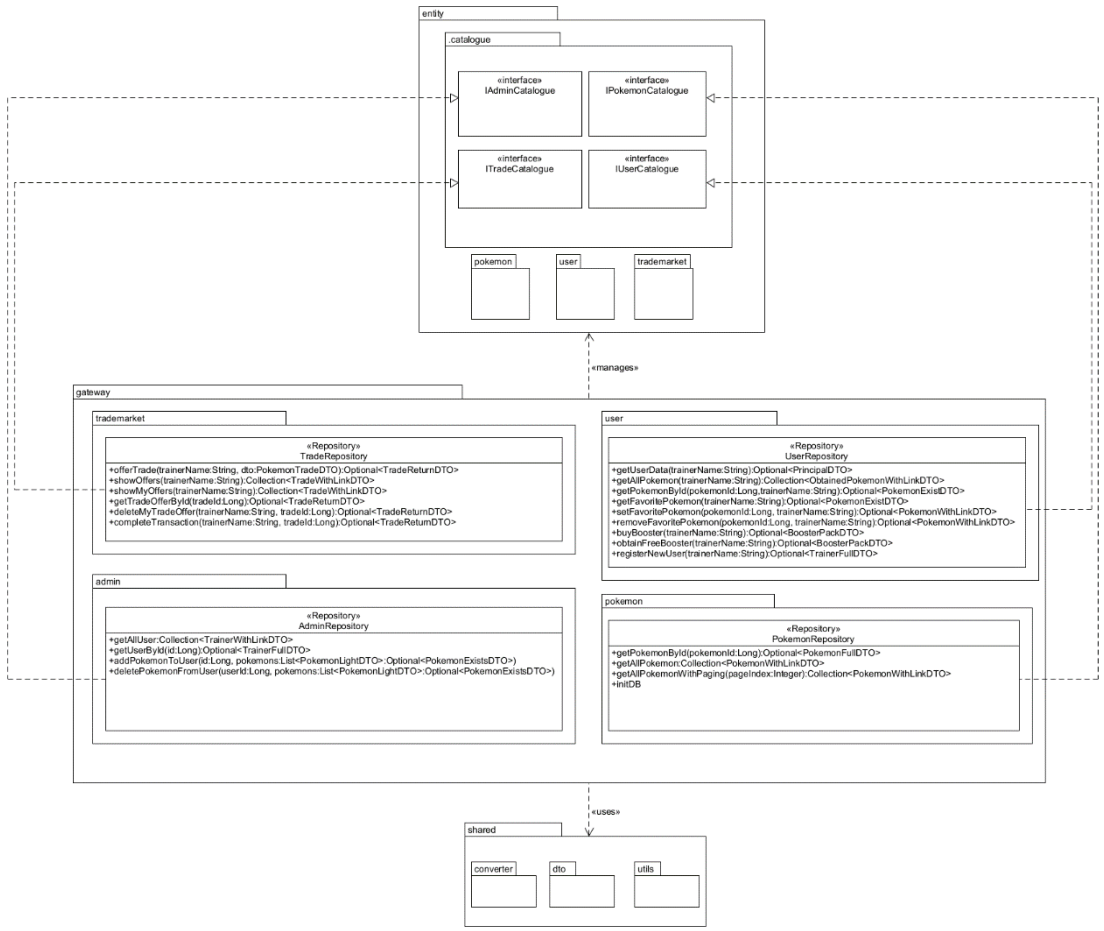

Abbildung 13 ECB-Gateway

Die Repository-Schicht ist zustandslos und wird im Klassenkopf mit der @Request-Scoped Annotation versehen. Wie in Kapitel 6.1.3 beschrieben, implementieren die Repository-Klassen auch die Admin-, User-, Trade- und Pokemon-Katalog-Schnittstellen aus der Entitäts-Schicht. Auch hier wird dasselbe Muster für das Aufteilen der Repository-Logik in vier Pakete weiterverfolgt.

Da die proprietäre Lösung Hibernate Panache verwendet wird, implementieren die Repository-Klassen auch das PanacheRepository<E>, wobei E für eine beliebige Entität steht, die die PanacheEntityBase mittels Erweiterung (extends) erweitert, wie in Kapitel 6.1.3 beschrieben.

Durch diese Umsetzung wird es dem Anwender ermöglicht, Datenbankoperationen mittels der Hibernate Panache API auszuführen. Zum Beispiel kann die Entität "Trainer", die die PanacheEntityBase erweitert, mithilfe der Hibernate Panache API unkompliziert persistiert werden.

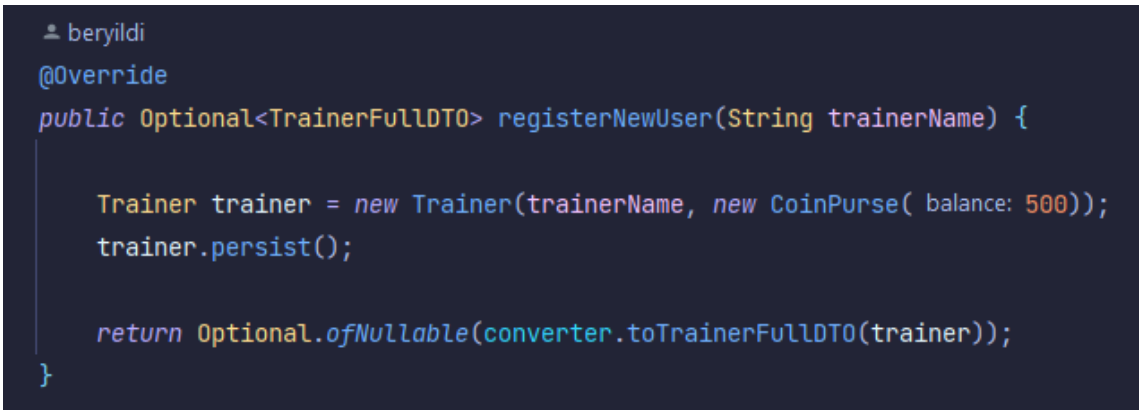

Abbildung 14 Hibernate Panache persist()

Gemäß dem selben Konzept können auch Daten aus der Datenbank einfach abgerufen und verwendet werden. Die Java Stream API wird hierfür genutzt, die im Modul "Design von Programmierschnittstellen" vertieft wurde. Der Vorteil der Verwendung der Java Stream API besteht darin, dass Boilerplate-Code durch Functional Interfaces ersetzt werden und somit der Quellcode lesbarer gestaltet wird.

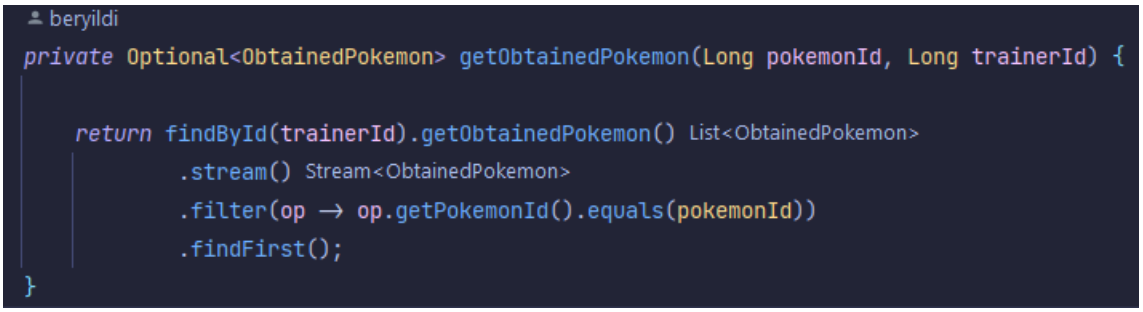

Abbildung 15 Hibernate Panache findById

Auch Update Operationen können einfach mit der Hibernate Panache API durchgeführt werden.

```java
@Override
public Optional<PokemonWithLinkDTO> setFavoritePokemon(Long pokemonId, String trainerName) {

    Optional<Trainer> oTrainer = getTrainer(trainerName);

    if (oTrainer.isPresent()) {
        Long trainerId = oTrainer.get().getId();

        Optional<ObtainedPokemon> obtainedPokemon = this.getObtainedPokemon(pokemonId, trainerId);

        if (obtainedPokemon.isPresent()) {
            obtainedPokemon.get().setFavorite(true);
            return Optional.of(converter.toPokemonWithLinkDTO(obtainedPokemon.get()));
        }
    }

    return Optional.empty();
}
```

Abbildung 16 Hibernate Panache Update

Laut den Abbildungen 14, 15 und 16 wurde die Verwendung des Typs Optional<E> als Rückgabewert gewählt, wobei E eine beliebige Java Bean repräsentiert. Es ist allgemein bekannt, dass das Rückgeben von null nicht zu einem guten Design führt, da es den Code semantisch unklarer macht und die Gefahr besteht, dass es zu NullPointerExceptions kommt, wenn der Entwickler dessen nicht bewusst ist. Die Verwendung von Optional ist eine elegantere Lösung, da der Entwickler so gezwungen ist, auf einen möglichen null-Wert zu reagieren. [11]

Durch den Zwang nun auf null-Werten zu reagieren, wird in der Boundary-Schicht entsprechend reagiert.

Der zurückgegebene Statuscode ist anhand der Prüfung vom Optional entweder 200 OK oder 404 No Content.

```
public Response setFavoritePokemon(@PathParam("pokemonId") @Min(ConfigSetting.MIN_POKEMON) @Max(ConfigSetting.MAX_POKEMON) Long pokemonId) {
    String trainerName = securityIdentity.getPrincipal().getName();

    Optional<PokemonWithLinkDTO> oPokemonWithLinkDTO = this.userManagement.setFavoritePokemon(pokemonId, trainerName);
    if (oPokemonWithLinkDTO.isPresent()) {
        this.addSelfLinkToPokemon(oPokemonWithLinkDTO.get());
        return Response.ok(oPokemonWithLinkDTO).build();
    }
    return Response.status(Response.Status.NOT_FOUND)
            .entity(new ErrorHandler( errorMessage: "Favorite Pokemon cannot be set because Pokemon ID with " + pokemonId + " does not exist"))
            .build();
}
```

Abbildung 17 Use Case von Optional Werten

[jbelasch]

In der Pokemon Ressource werden Informationen zu den einzelnen Pokémon aus einer externen API gefetcht und lokal in der Projekt Datenbank persistiert. Dies geschieht immer, wenn das Back-End gestartet wird. Dazu wir ein @Observes StartupEvent genutzt:

```
beryildi
public void initPokemonDB(@Observes StartupEvent event) {
    this.initDB();
}
```

Abbildung 18 StartupEvent - Pokemon Datenbank

Die Methode initPokemonDB wird also beim Start der Anwendung einmalig ausgeführt.

6.1.5 ACL

[beryildi]

Das folgende Unterkapitel befasst sich mit dem Anti-Corruption-Layer und deren Implementation.

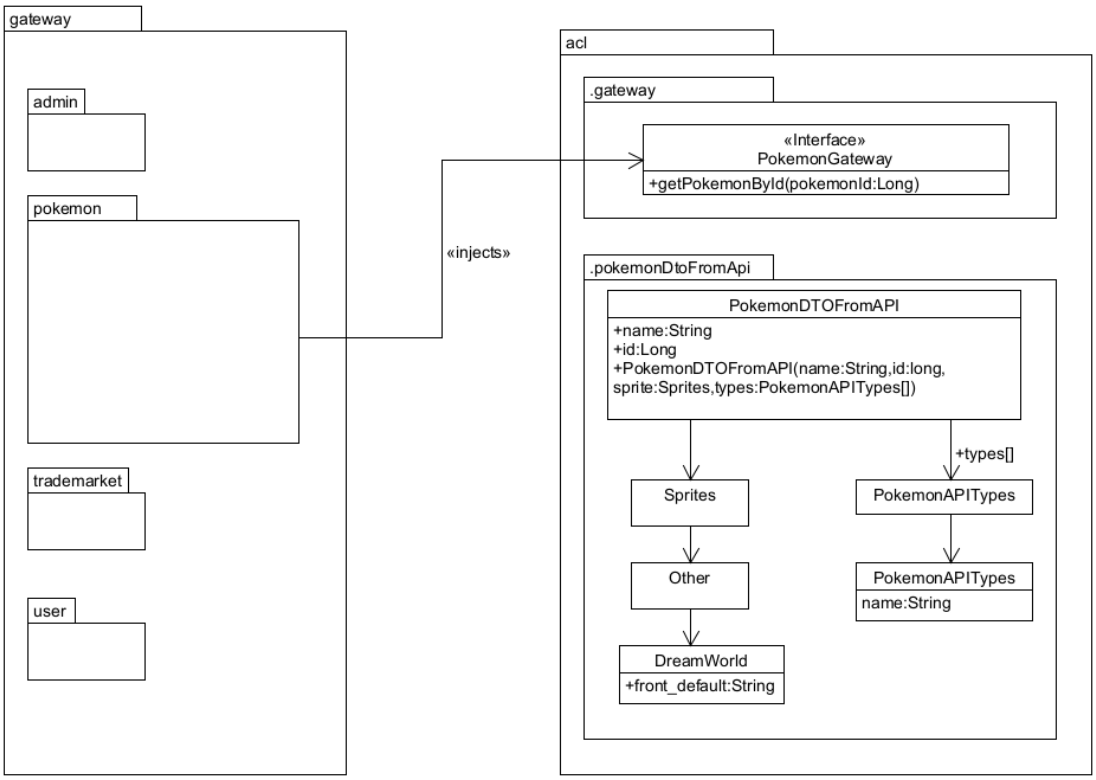

Abbildung 19 ACL

Das Anti-Corruption Layer eine Schicht, die zwischen einer Anwendung und einer externen Komponente eingesetzt wird. Dies stellt sicher, dass die Daten zwischen den Komponenten in einem konsistenten und standardisierten Format vorliegen. In der Pokecollect-Anwendung ist die externe Komponente die Pokemon API. Das Paketdiagramm in Abbildung 20 repräsentiert die JSON-Datei, die von der externen Komponente zur Verfügung gestellt wird. Die Pokemon API liefert umfassendere Informationen, aber diese werden in der Klasse PokemonDTOFromAPI entsprechend beschränkt.

Abbildung 20 JSON aus externer API

Um die Kommunikation mit der externen API zu ermöglichen, wird in der Pokecollect Applikation ein Rest-Client genutzt, welcher durch die Extension bereitgestellt wird, die von der Quarkus Umgebung angeboten wird. Hierbei wird das erworbene Wissen aus dem zweiten Praktikum angewendet.

Abbildung 21 Quarkus Rest-Client Extension

Zur Konfiguration der API-Adresse ist es erforderlich, dass diese in der Datei "application.properties" hinterlegt wird.

Abbildung 22 API Config

Schließlich kann die externe API verwendet werden. Dies wird durch das Interface PokemonGateway realisiert.

```
beryildi *
@Path("/pokemon")
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient(configKey = "pokemonAPI")
public interface PokemonGateway {

    beryildi
    @GET
    @Path("/{pokemonId}")
    PokemonDTOFromAPI getPokemonById(@PathParam("pokemonId") Long pokemonId);
}
```

Abbildung 23 Externe PokemonAPI Gateway

6.1.6 Shared

[beryildi]

Das folgende Unterkapitel befasst sich mit dem Shared-Layer.

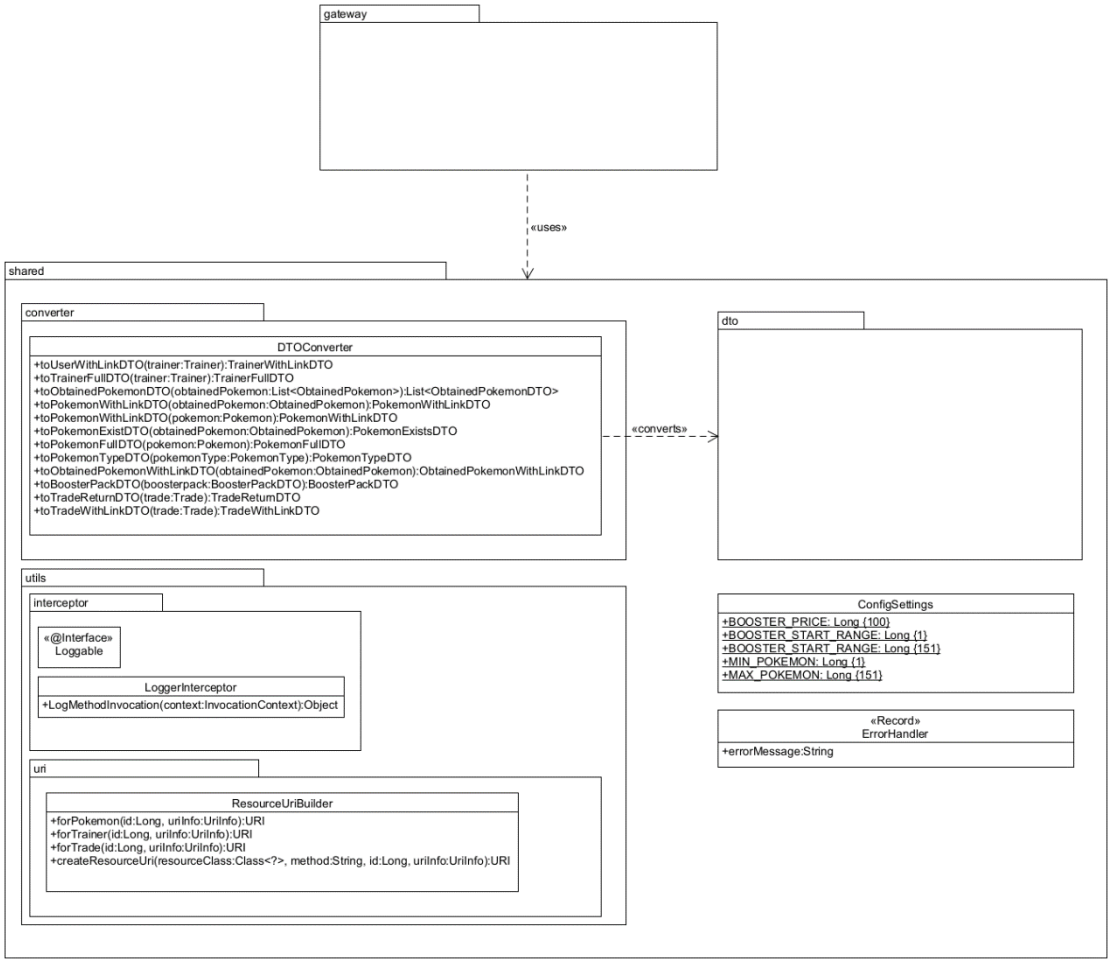

Abbildung 24 ECB-Shared

Das Package Shared stellt eine zentralisierte Umgebung bereit, in der öffentliche Klassen für externe Systeme zusammengefasst werden. Innerhalb dieses Packages befinden sich sowohl eingehende als auch ausgehende Daten-Transfer-Objekte (DTOs), ein DTO-Konverter, der die Umwandlung dieser DTOs unterstützt, ein Logger-Interceptor, ein Ressourcen-Builder zur Generierung von Links und Konfigurationsklasse.

6.1.7 Interceptor

[beryildi]

In Java Quarkus werden Interceptoren oft verwendet, um bestimmte Aktionen auszuführen, während eine Anwendung ausgeführt wird. Einer der häufigen Einsatzbereiche für Interceptoren ist das Loggen.

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Loggable {}
```

Abbildung 25 Interceptor

Abbildung 25 definiert eine benutzerdefinierte Annotation in Java. Diese Annotation gibt an, dass die Annotation "Loggable" als Interception-Binding verwendet werden kann. Die Annotation "@Retention (RUNTIME)" gibt an, dass die Annotation zur Laufzeit verfügbar sein soll, während die Annotation "@Target ({METHOD, TYPE})" angibt, dass die Annotation auf Methoden und Typen angewendet werden kann.

```
@Interceptor
@Loggable
public class LoggerInterceptor {

    private static final Logger logger = Logger.getLogger(LoggerInterceptor.class.getName());

    @AroundInvoke
    public Object logMethodInvocation(InvocationContext context) throws Exception {

        // TODO: Beobachten wie das mit der performance ist wegen reflections...

        String methodName = context.getMethod().getName();
        String className = context.getTarget().getClass().getName();


        long startTime = System.currentTimeMillis();
        logger.info(className + " : " + methodName + " - Method invocation started at: " + TimeUnit.MILLISECONDS.toSeconds(startTime));

        Object result = context.proceed();

        long endTime = System.currentTimeMillis();
        logger.info(className + " : " + methodName + " - Method invocation ended at: " + TimeUnit.MILLISECONDS.toSeconds(endTime));
        logger.info(className + " : " + methodName + " - Total time taken: " + (endTime - startTime) + "ms");
        return result;
    }
}
```

Abbildung 26 Interceptor Implementation

Abbildung 26 definiert einen Java-Interceptor namens "LoggerInterceptor", der mit der Annotation "Loggable" gekennzeichnet ist. Der Interceptor misst die Dauer der Methode und gibt Informationen zu Klassenname, Methode, Startzeit und Dauer in Millisekunden an und gibt diese in der Konsole aus.

7 Zusammenfassung und Fazit

[beryildi, jbelasch]

Wir möchten am Ende unserer Ausarbeitung noch kurz die Zeit nehmen und uns einem kleinen Fazit widmen.

7.1 Bewertung und Reflexion

Die Durchführung der Aufgabe stellte für alle Beteiligten eine äußerst positive Erfahrung dar. Dieses Projekt ermöglichte es uns, erstmalig eine Backend-Architektur zu konzipieren und diese mittels einer REST-API mit einem echten Web-Frontend zu verbinden. Die von uns gesetzten Ziele konnten erfolgreich mit Wunsch- und Kann-Kriterien umgesetzt werden. Die im Rahmen der Vorlesungen "Softwarearchitektur" und "Design von Programmierschnittstellen" vermittelten Inhalte konnten wir umsetzen.

Während des Projekts hatten wir auch die Möglichkeit, neue Techniken kennenzulernen. Dank den genannten Vorlesungen, insbesondere "Design von Programmierschnittstellen", durften wir erstmals mit der Java Stream API arbeiten, die das Verwalten von Listen ungemein erleichterte. Auch die Arbeit mit Keycloak brachte uns wertvolle Erfahrungen. Die Zusammenarbeit im Team verlief einwandfrei. Die Verknüpfung unseres Backends mit einem Frontend stellte für uns die größte Herausforderung, die jedoch bewältigt wurde. Wir sind dankbar dafür, diese Erfahrung sammeln zu dürfen.

7.2 Erweiterung und Ausbaumöglichkeiten

Im Folgenden möchten wir ein paar Dinge ansprechen, die ausbaufähig sind:

- Aufgrund von Zeitmangel und immensem Zeitdruck anderer Module (SWE, SP3DA, Webanwendungen) konnten wir auch hier nicht viele Integrationstest mit RestAssured schreiben.
- Tests decken nur erfolgsfälle ab.
- Konfiguration vom Keycloak Client (RootUrl, Valid Redirect URL) konnten wir nicht automatisieren.

7.3 Verbesserungsvorschläge

Im folgenden Abschnitt möchten wir einige Vorschläge zur Verbesserung ansprechen.

- Tipps beziehungsweise Beispiele für die Verwendung der Java Keycloak API. Die Verwendung der Java Keycloak API erwies sich als besonders herausfordernd aufgrund des Mangels an umfassenden Beispielen im Internet oder in der Vorlesung. (Wie nutzt man die Keycloak API und legt beispielhaft einen User an)

- Ein Praktikum, das sich auf die Arbeit mit JSON-Webtokens für die Authentifizierung konzentriert. (Wäre das Video nicht im Opencast verfügbar gewesen sein, wäre es sehr unwahrscheinlich, dass wir die Authentifizierung mit Keycloak implementiert bekommen hätten.)

- Ich hätte mir ein ausführliches Praktikum zum Testen mit RestAssured oder allgemein JUnit Tests gewünscht. (Welche Best Pracices gibt es? wie teste ich effektiv, was soll ich überhaupt testen)

7.4 Schlusswort

Insgesamt haben wir das Projekt gut geführt und konnten viel für die Zukunft mitnehmen. Wir konnten auch hier, wertvolle Erfahrungen sammeln. Die einzigen kritischen Punkte, die wir aufzählen können, sind einerseits die im 7.3 genannten Aspekte und andererseits das sehr eng getaktete Semester (mit 5 Hausarbeiten, die seit Anfang Februar alle wöchentlich abgegeben werden müssen).

8 Literaturverzeichnis

[1] E. Evans, „Domain-Driven Design: Tackling Complexity in the Heart of Software“.

[2] H. Sevinc, „adesso,“ 11 01 2021. [Online]. Available: https://www.adesso.de/de/news/blog/die-solid-design-prinzipien.jsp. [Zugriff am 10 02 2023].

[3] F. Listing, „MicroConsult,“ [Online]. Available: https://www.microconsult.de/blog/2019/05/fl_solid-prinzipien/#:~:text=Das%20Single%2DResponsibility%2DPrinzip%20besagt,desto%20h%C3%B6her%20ist%20das%20Fehlerrisiko.. [Zugriff am 14 02 2023].

[4] JetBrains, 11 02 2023. [Online]. Available: https://www.jetbrains.com/code-with-me/.

[5] „Quarkus,“ 09 02 2023. [Online]. Available: https://quarkus.io/about/.

[6] „login.master,“ 09 02 2023. [Online]. Available: https://login-master.com/keycloak-open-source-identity-und-access-management/.

[7] vogella GmbH, Lars Vogel, Simon Scholz, „vogella,“ 09 02 2023. [Online]. Available: https://www.vogella.com/tutorials/JavaPersistenceAPI/article.html.

[8] „Hibernate,“ 09 02 2023. [Online]. Available: https://hibernate.org/validator/.

[9] 09 02 2023. [Online]. Available: https://javaee.github.io/jsonb-spec/.

[10] „thalesgroup,“ [Online]. Available: https://cpl.thalesgroup.com/de/software-monetization/proprietary-software-license#:~:text=Trotz%20dieser%20Vorteile%20hat%20auch,an%20ihren%20Bedarf%20anpassen%20k%C3%B6nnen.. [Zugriff am 09 02 2023].

[11] K. Spichale, „API-Design,“ in *Praxishandbuch für Java- und Webservice-Entwickler*, dpunkt.verlag, 2019, p. 382.